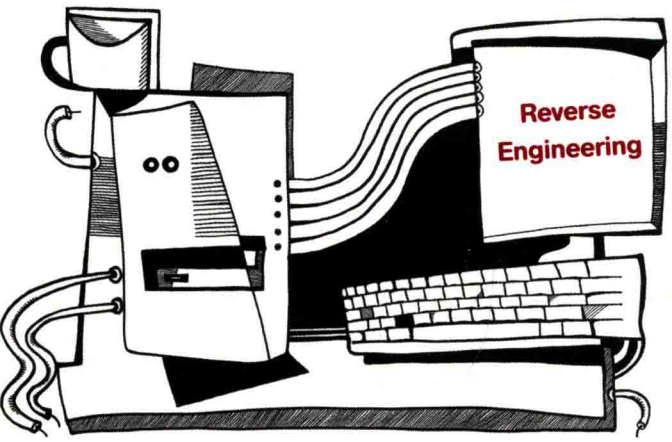


安天 ANTIY

异步图书 www.epubit.com.cn

- 了解逆向工程的权威指南
- 初学者必备的大百科全书
- 安天网络安全工程师培训必读书目



Reverse Engineering for Beginners

逆向工程权威指南 下册

[乌克兰] Dennis Yurichev◎著

Archer 安天安全研究与应急处理中心◎译

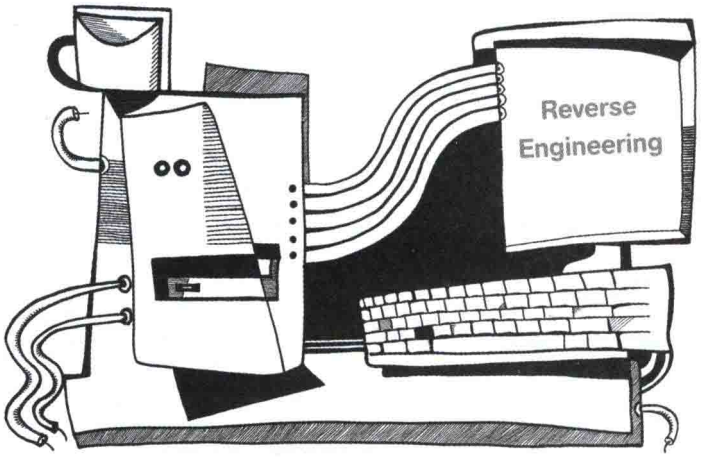




中国工信出版集团

人民邮电出版社 POSTS & TELECOM PRESS





Reverse Engineering for Beginners

逆向工程权威指南 下册

[乌克兰] Dennis Yurichev◎著
Archer 安天安全研究与应急处理中心◎译

人民邮电出版社
北京

目　录

第 47 章　字符串剪切……485

47.1　x64 下的 MSVC 2013 优化……486
47.2　x64 下采用编辑器 GCC 4.9.1 进行非优化操作……487
47.3　x64 下的 GCC 4.9.1 优化……488
47.4　ARM64：非优化的 GCC（Linaro）4.9……489
47.5　ARM64:优化 GCC（Linaro）4.9……490
47.6　ARM: Keil 6/2013 优化（ARM 模式）……491
47.7　ARM:Keil 6/2013（Thumb 模式）优化……492
47.8　MIPS……493

第 48 章　toupper()函数……495

48.1　x64……495
48.1.1　两个比较操作……495
48.1.2　一个比较操作……496
48.2　ARM……497
48.2.1　ARM64 下的 GCC……497
48.3　总结……498

第 49 章　不正确的反汇编代码……499

49.1　x86 环境下的从一开始错误的反汇编……499
49.2　随机噪音，怎么看起来像反汇编指令？……500

第 50 章　混淆技术……505

50.1　字符串变换……505
50.2　可执行代码……506
50.2.1　插入垃圾代码……506
50.2.2　用多个指令组合代替原来的一个指令……506
50.2.3　始终执行或者从来不会执行的代码……506
50.2.4　把指令序列搞乱……507
50.2.5　使用间接指针……507
50.3　虚拟机以及伪代码……507
50.4　一些其他的事情……507
50.5　练习题……508
50.5.1　练习 1……508

第 51 章　C++……509

51.1　类……509
51.1.1　一个简单的例子……509
51.1.2　类继承……515
51.1.3　封装……518
51.1.4　多重继承……520
51.1.5　虚拟方法……523
51.2　ostream 输出流……526
51.3　引用……527
51.4　STL/标准模板库（Standard Template Library）……528
51.4.1　std::string（字符串）……528
51.4.2　std::list……535
51.4.3　std::vector 标准向量……545
51.4.4　std::map()和 std::set()……552

第 52 章　数组与负数索引……563

第 53 章　16 位的 Windows 程序……566

53.1　例子#1……566
53.2　例子#2……566
53.3　例子#3……567
53.4　例子#4……568
53.5　例子#5……571
53.6　例子#6……574
53.6.1　全局变量……576

第四部分　Java

第 54 章　Java……581

54.1　简介……581
54.2　返回一个值……581
54.3　简单的计算函数……586
54.4　JVM 的内存模型……588
54.5　简单的函数调用……588
54.6　调用函数 beep()（蜂鸣器）……590
54.7　线性同余随机数产生器（PRNG）…591
54.8　条件转移……592
54.9　传递参数……594
54.10　位操作……595

54.11 循环……596
54.12 switch()语句……598
54.13 数组……599
54.13.1 简单的例子……599
54.13.2 数组元素求和……601
54.13.3 输入变量为数组的主函数 main()……601
54.13.4 预设初始值的的数组……602
54.13.5 可变参数函数……604
54.13.6 二维数组……606
54.13.7 三维数组……606
54.13.8 小结……607
54.14 字符串……607
54.14.1 第一个例子……607
54.14.2 第二个例子……608
54.15 异常处理……609
54.16 类……612
54.17 简单的补丁……614
54.17.1 第一个例子……614
54.17.2 第二个例子……616
54.18 总结……618

第五部分 在代码中发现重要而有趣的内容

第 55 章 编译器产生的文件特征……621

55.1 Microsoft Visual C++……621
55.1.1 命名规则……621
55.2 GCC 编译器……621
55.2.1 命名规则……621
55.2.2 Cygwin……621
55.2.3 MinGW……621
55.3 Intel FORTRAN……621
55.4 Watcom 以及 OpenWatcom……622
55.4.1 命名规则……622
55.5 Borland 编译器……622
55.5.1 Dephi 编程语言……622
55.6 其他的已知 DLL 文件……623

第 56 章 Win32 环境下与外部通信……624

56.1 在 Windows API 中最经常使用的函数……624
56.2 tracer:解析指定模块的所有函数……624

第 57 章 字符串……626

57.1 字符串……626
57.1.1 C/C++中的字符串……626
57.1.2 Borland Delphi……626
57.1.3 Unicode 编码……626
57.1.4 Base64……628
57.2 错误/调试信息……629
57.3 可疑的魔数字符串……629

第 58 章 调用宏 assert()（中文称为断言）……630

第 59 章 常数……631

59.1 魔数……631
59.1.1 动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）……632
59.2 寻找常数……632

第 60 章 检索关键指令……633

第 61 章 可疑的代码模型……635

61.1 XOR 异或指令……635
61.2 手写汇编代码……635

第 62 章 魔数与程序调试……637

第 63 章 其他的事情……638

63.1 总则……638
63.2 C++……638
63.3 部分二进制文件的特征……638
63.4 内存“快照”对比……638
63.4.1 Windows 注册表……639
63.4.2 瞬变比较器 Blink-comparator……639

第六部分 操作系统相关

第 64 章 参数的传递方法（调用规范）……643

64.1 cdecl [C Declaration 的缩写]……643
64.2 stdcall [Standard Call 的缩写]……643
64.2.1 带有可变参数的函数……644
64.3 fastcall……644
64.3.1 GCC regparm……645
64.3.2 Watcom/OpenWatcom……645
64.4 thiscall……645
64.5 64 位下的 x86……646
64.5.1 Windows x64……646
64.5.2 64 位下的 Linux……648

64.6 单/双精度数型返回值……649
64.7 修改参数……649
64.8 指针型函数参数……649

第 65 章 线程本地存储 TLS……652

65.1 线性同余发生器（改）……652
65.1.1 Win32 系统……652
65.1.2 Linux 系统……656

第 66 章 系统调用（syscall-s）……658

66.1 Linux……658
66.2 Windows……659

第 67 章 Linux……660

67.1 位置无关的代码……660
67.1.1 Windows……662
67.2 在 Linux 下的 LD_PRELOAD……662

第 68 章 Windows NT……666

68.1 CRT (Win32)……666
68.2 Win32 PE 文件……669
68.2.1 术语……670
68.2.2 基地址……670
68.2.3 子系统……670
68.2.4 操作系统版本……670
68.2.5 段……671
68.2.6 重定向段 Relocations(relocs)……672
68.2.7 导出段和导入段……672
68.2.8 资源段……674
68.2.9 .NET……675
68.2.10 TLS 段……675
68.2.11 工具……675
68.2.12 更进一步……675
68.3 Windows SEH……675
68.3.1 让我们暂时把 MSVC 放在一边……675
68.3.2 让我们重新回到 MSVC……680
68.3.3 Windows x64……693
68.3.4 关于 SEH 的更多信息……697
68.4 Windows NT：临界区段……697

第七部分 常用工具

第 69 章 反汇编工具……703

69.1 IDA……703

第 70 章 调试工具……704

70.1 tracer……704
70.2 OllyDbg……704
70.3 GDB……704

第 71 章 系统调用的跟踪工具……705

71.1 strace/dtruss……705

第 72 章 反编译工具……706

第 73 章 其他工具……707

第八部分 更多范例

第 74 章 修改任务管理器（Vista）……711

74.1 使用 LEA 指令赋值……713

第 75 章 修改彩球游戏……715

第 76 章 扫雷（Windows XP）……717

76.1 练习题……721

第 77 章 人工反编译与 Z3 SMT 求解法……722

77.1 人工反编译……722
77.2 Z3 SMT 求解法……725

第 78 章 加密狗……730

78.1 例 1：PowerPC 平台的 MacOS Classic 程序……730
78.2 例 2: SCO OpenServer……737
解密错误信息……745
78.3 例 3: MS-DOS……747

第 79 章 “QR9”：魔方态加密模型……753

第 80 章 SAP……782

80.1 关闭客户端的网络数据包压缩功能……782
80.2 SAP 6.0 的密码验证函数……793

第 81 章 Oracle RDBMS……797

81.1 V$VERSION 表……797
81.2 X$KSMLRU 表……805
81.3 V$TIMER 表……806

第 82 章 汇编指令与屏显字符……811
82.1 EICAR……811
第 83 章 实例演示……813
83.1 10PRINT CHR$(205.5+RND(1));: GOTO 10……813
83.1.1 Trixter 的 42 字节程序……813
83.1.2 笔者对 Trixter 算法的改进：27 字节……814
83.1.3 从随机地址读取随机数……814
83.1.4 其他……815
83.2 曼德博集合……815
83.2.1 理论……816
83.2.2 demo 程序……820
83.2.3 笔者的改进版……822
第九部分 文件分析
第 84 章 基于 XOR 的文件加密……827
84.1 Norton Guide：单字节 XOR 加密实例……827
信息熵……828
84.2 4 字节 XOR 加密实例……828
84.3 练习题……830
第 85 章 Millenium 游戏的存档文件……831
第 86 章 Oracle 的.SYM 文件……835
第 87 章 Oracle 的.MSDB 文件……842
总结……845
第十部分 其他
第 88 章 npad……849
第 89 章 修改可执行文件……851
89.1 文本字符串……851
89.2 x86 指令……851
第 90 章 编译器内部函数……852
第 91 章 编译器的智能短板……853
第 92 章 OpenMP……854
92.1 MSVC……856
92.2 GCC……858
第 93 章 安腾指令……860
第 94 章 8086 的寻址方式……863
第 95 章 基本块重排……864
95.1 PGO 的优化方式……864
第十一部分 推荐阅读
第 96 章 参考书籍……869
96.1 Windows……869
96.2 C/C++……869
96.3 x86/x86-64……869
96.4 ARM……869
96.5 加密学……869
第 97 章 博客……870
97.1 Windows 平台……870
第 98 章 其他内容……871
第十二部分 练习题
第 99 章 初等难度练习题……875
99.1 练习题 1.4……875
第 100 章 中等难度练习题……876
100.1 练习题 2.1……876
100.1.1 Optimizing MSVC 2010 x86……876
100.1.2 Optimizing MSVC 2012 x64……877
100.2 练习题 2.4……877
100.2.1 Optimizing MSVC 2010……877
100.2.2 GCC 4.4.1……878
100.2.3 Optimizing Keil（ARM mode）……879
100.2.4 Optimizing Keil（Thumb mode）……880
100.2.5 Optimizing GCC 4.9.1（ARM64）……880
100.2.6 Optimizing GCC 4.4.5

（MIPS）……881
100.3 练习题 2.6……882
100.3.1 Optimizing MSVC 2010……882
100.3.2 Optimizing Keil（ARM mode）……883
100.3.3 Optimizing Keil（Thumb mode）……884
100.3.4 Optimizing GCC 4.9.1（ARM64）……884
100.3.5 Optimizing GCC 4.4.5（MIPS）……885
100.4 练习题 2.13……885
100.4.1 Optimizing MSVC 2012……886
100.4.2 Keil（ARM mode）……886
100.4.3 Keil（Thumb mode）……886
100.4.4 Optimizing GCC 4.9.1（ARM64）……886
100.4.5 Optimizing GCC 4.4.5（MIPS）……887
100.5 练习题 2.14……887
100.5.1 MSVC 2012……887
100.5.2 Keil（ARM mode）……888
100.5.3 GCC 4.6.3 for Raspberry Pi（ARM mode）……888
100.5.4 Optimizing GCC 4.9.1（ARM64）……889
100.5.5 Optimizing GCC 4.4.5（MIPS）……890
100.6 练习题 2.15……891
100.6.1 Optimizing MSVC 2012 x64……892
100.6.2 Optimizing GCC 4.4.6 x64……894
100.6.3 Optimizing GCC 4.8.1 x86……895
100.6.4 Keil（ARM 模式）：面向 Cortex-R4F CPU 的代码……896
100.6.5 Optimizing GCC 4.9.1（ARM64）……897
100.6.6 Optimizing GCC 4.4.5（MIPS）……898
100.7 练习题 2.16……899
100.7.1 Optimizing MSVC 2012 x64……899
100.7.2 Optimizing Keil（ARM mode）……899
100.7.3 Optimizing Keil（Thumb mode）……900
100.7.4 Non-optimizing GCC 4.9.1（ARM64）……900
100.7.5 Optimizing GCC 4.9.1（ARM64）……901
100.7.6 Non-optimizing GCC 4.4.5（MIPS）……904
100.8 练习题 2.17……905
100.9 练习题 2.18……905
100.10 练习题 2.19……905
100.11 练习题 2.20……905

第 101 章 高难度练习题……906

101.1 练习题 3.2……906
101.2 练习题 3.3……906
101.3 练习题 3.4……906
101.4 练习题 3.5……906
101.5 练习题 3.6……907
101.6 练习题 3.8……907

第 102 章 Crackme/Keygenme……908

附录 A x86……909

A.1 数据类型……909
A.2 通用寄存器……909
A.2.1 RAX/EAX/AX/AL……909
A.2.2 RBX/EBX/BX/BL……910
A.2.3 RCX/ECX/CX/CL……910
A.2.4 RDX/EDX/DX/DL……910
A.2.5 RSI/ESI/SI/SIL……910
A.2.6 RDI/EDI/DI/DIL……910
A.2.7 R8/R8D/R8W/R8L……911
A.2.8 R9/R9D/R9W/R9L……911
A.2.9 R10/R10D/R10W/R10L……911
A.2.10 R11/R11D/R11W/R11L……911
A.2.11 R12/R12D/R12W/R12L……911
A.2.12 R13/R13D/R13W/R13L……911
A.2.13 R14/R14D/R14W/R14L……912
A.2.14 R15/R15D/R15W/R15L……912
A.2.15 RSP/ESP/SP/SPL……912
A.2.16 RBP/EBP/BP/BPL……912
A.2.17 RIP/EIP/IP……912
A.2.18 段地址寄存器 CS/DS/ES/SS/FS/GS……913
A.2.19 标识寄存器……913
A.3 FPU 寄存器……913
A.3.1 控制字寄存器（16 位）……914

A.3.2 状态字寄存器（16位）……914
A.3.3 标记字寄存器（16位）……915
A.4 SIMD寄存器……915
A.4.1 MMX寄存器……915
A.4.2 SSE与AVX寄存器……915
A.5 FPU调试寄存器……915
A.5.1 DR6规格……916
A.5.2 DR7规格……916
A.6 指令……917
A.6.1 指令前缀……917
A.6.2 常见指令……917
A.6.3 不常用的汇编指令……922
A.6.4 FPU指令……927
A.6.5 可屏显的汇编指令（32位）……928

附录B ARM……931

B.1 术语……931
B.2 版本差异……931
B.3 32位ARM（AArch32）……931
B.3.1 通用寄存器……931
B.3.2 程序状态寄存器/CPSR……931
B.3.3 VFP（浮点）和NEON寄存器……932
B.4 64位ARM（AArch64）……932
通用寄存器……932
B.5 指令……933
Conditional codes速查表……933

附录C MIPS……934

C.1 寄存器……934
C.1.1 通用寄存器GPR……934
C.1.2 浮点寄存器FPR……934
C.2 指令……934
转移指令……935

附录D 部分GCC库函数……936

附录E 部分MSVC库函数……937

附录F 速查表……938

F.1 IDA……938
F.2 OllyDbg……939
F.3 MSVC选项……939
F.4 GCC……939
F.5 GDB……940

附录G 练习题答案……941

G.1 各章练习……941
第3章……941
第5章……941
第7章……941
第13章……942
第14章……942
第15章……942
第16章……942
第17章……943
第18章……943
第19章……944
第21章……945
第41章……946
第50章……946
G.2 初级练习题……947
G.2.1 练习题1.1……947
G.2.2 练习题1.4……947
G.3 中级练习题……947
G.3.1 练习题2.1……947
G.3.2 练习题2.4……948
G.3.3 练习题2.6……949
G.3.4 练习题2.13……949
G.3.5 练习题2.14……949
G.3.6 练习题2.15……949
G.3.7 练习题2.16……950
G.3.8 练习题2.17……950
G.3.9 练习题2.18……950
G.3.10 练习题2.19……950
G.3.11 练习题2.20……950
G.4 高难度练习题……950
G.4.1 练习题3.2……950
G.4.2 练习题3.3……950
G.4.3 练习题3.4……950
G.4.4 练习题3.5……950
G.4.5 练习题3.6……951
G.4.6 练习题3.8……951
G.5 其他练习题……951
G.5.1 “扫雷（Windows XP）”……951

参考文献……952

第 47 章　字符串剪切

我们经常需要去除字符串里的开始符号或者结束符号。

本章介绍的程序，专门用于去除字符串的结尾部分的回车和换行字符 CR/LF。

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
        char c;
        size_t str_len;

        // work as long as \r or \n is at the end of string
        // stop if some other character there or its an empty string'
        // (at start or due to our operation)
        for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
        {
                if (c=='\r' || c=='\n')
                        s[str_len-1]=0;
                else
                        break;
        };
        return s;
};

int main()
{
        // test

        // strdup() is used to copy text string into data segment,
        // because it will crash on Linux otherwise,
        // where text strings are allocated in constant data segment,
        // and not modifiable.

        printf ("[%s]\n", str_trim (strdup("")));
        printf ("[%s]\n", str_trim (strdup("\n")));
        printf ("[%s]\n", str_trim (strdup("\r")));
        printf ("[%s]\n", str_trim (strdup("\n\r")));
        printf ("[%s]\n", str_trim (strdup("\r\n")));
        printf ("[%s]\n", str_trim (strdup("test1\r\n")));
        printf ("[%s]\n", str_trim (strdup("test2\n\r")));
        printf ("[%s]\n", str_trim (strdup("test3\n\r\n\r")));
        printf ("[%s]\n", str_trim (strdup("test4\n")));
        printf ("[%s]\n", str_trim (strdup("test5\r")));
        printf ("[%s]\n", str_trim (strdup("test6\r\r\r")));
};
```

输入参数总能正常返回并退出。当你需要对串进行批量处理时会非常方便，就像这里的 main()函数一样。

在该程序循环体的 for()语句里有两个判断条件表达式：一个条件表达式是 str_len>0（字符串的长度大于零）；另外一个是 c=s[str_len-1]（意思是取出的值不为 0、不是终止符）。循环判断语句“str_len>0 && (c=s[str_len-1])”实际上利用了所谓“逻辑短路”的执行特性，因此可以这样书写第二个判断表达式（可以参考 Yur13:p.1.3.8）。C/C++编译器自左至右的逐一检测判断条件。因为逻辑操作符是“&&”（与），所以

一旦第一个条件表达式的值为假，计算机就不用再判断（执行）第二个条件判断表达式。实际上，第二个条件表达式是一种只能在相应的条件下才可以运行的语句。笔者综合利用了第一个条件表达式、逻辑操作符“&&”的逻辑含义以及“逻辑短路”的特性，为第二个条件判断表达式限定了执行条件。

47.1 x64 下的 MSVC 2013 优化

指令清单 47.1 x64 下的 MSVC 2013 优化

```
s$ = 8
str_trim PROC

; RCX is the first function argument and it always holds pointer to the string

; this is strlen() function inlined right here:
; set RAX to 0xFFFFFFFFFFFFFFFF (-1)
        or      rax, -1
$LL14@str_trim:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT $LL14@str_trim
; is string length zero? exit then
        test    eax, eax
$LN18@str_trim:
        je      SHORT $LN15@str_trim
; RAX holds string length
; here is probably disassembler (or assembler printing routine) error,
; LEA RDX... should be here instead of LEA EDX...
        lea     edx, DWORD PTR [rax-1]
; idle instruction: EAX will be reset at the next instructions execution'
        mov     eax, edx
; load character at address s[str_len-1]
        movzx   eax, BYTE PTR [rdx+rcx]
; save also pointer to the last character to R8
        lea     r8, QWORD PTR [rdx+rcx]
        cmp     al, 13 ; is it '\r'?
        je      SHORT $LN2@str_trim
        cmp     al, 10 ; is it '\n'?
        jne     SHORT $LN15@str_trim
$LN2@str_trim:
; store 0 to that place
        mov     BYTE PTR [r8], 0
        mov     eax, edx
; check character for 0, but conditional jump is above...
        test    edx, edx
        jmp     SHORT $LN18@str_trim
$LN15@str_trim:
; return "s"
        mov     rax, rcx
        ret     0
str_trim ENDP
```

第一个特征就是 MSVC 编译器对字符串长度函数 strlen()进行了内联（inline）式的展开和嵌入处理。编译器认为，内联处理后的执行效率会比常规的函数调用（call）的效率更高 。有关内联函数可以参考本书的第 43 章。

内嵌处理之后，strlen()函数的第一个指令是：OR RAX,0xffffffffffffffff。我们不清楚为何 MSVC 采用 OR（或）指令，而没有采用 MOV RAX, 0xffffffffffffffff 指令直接赋值。当然，这两条指令执行的是相同操作：将所有位设置为 1。因此这个数值就是赋值为−1。可以参考本书的第 30 章。

你也可能会问，为什么 strlen()函数会用到−1 这个数。当然是出于优化的目的。这里我们列出了 MSVC

的编译代码。

指令清单 47.2　x64 下的 MSVC 2013 的内嵌函数 strlen()

```
; RCX = pointer to the input string
; RAX = current string length
        or      rax, -1
label:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT label
; RAX = string length
```

如果把这个变量的初始值设置为 0，那么是否可以把代码压缩得更短一些呢？我们可以来试试。

指令清单 47.3　我们的 strlen()字符串长度函数版本

```
; RCX = pointer to the input string
; RAX = current string length
        xor     rax, rax
label:
        cmp     byte ptr [rcx+rax], 0
        jz      exit
        inc     rax
        jmp     label
exit:
; RAX = string length
```

我们没能成功。因为我们不得不加入了一个额外的指令：JMP 跳转指令。

在使用“－1”作为初始值之后，MSVC 2013 随即在加载字符的指令之前分配了一个 INC 指令。如果第一个字符是 0（终止符），那么 RAX 就直接为 0，返回的字符串长度还会是 0。

函数中的其余部分还是很好懂的。当然在程序的最后有另外一个技巧。除去那些内联之后的 strlen()函数的展开代码，整个函数就只有 3 个条件指令。其实，从道理上讲这里应该有 4 个转移指令：第 4 个应当位于函数的结尾部分，用于检查当前字符是不是 0。然后这段代码使用了一个跳转到“$LN18@str_trim”标签的无条件转移指令，而这个标签后面的第一个指令就是条件转移指令 JE。编译器用这种指令组用来判断输入的字符串是不是空字符串（当前字符是不是终止符），而且直接就是在 strlen()执行结束后。所以这里使用 JE 指令有两个目的。这也许杀鸡用宰牛刀，但是无论如何，MSVC 就是这样做的。

要想提示程序性能，就应当尽量脱离条件转移指令进行程序作业。有关详情请参阅本书第 33 章。

47.2　x64 下采用编辑器 GCC 4.9.1 进行非优化操作

```
str_trim:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
; for() first part begins here
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    strlen
        mov     QWORD PTR [rbp-8], rax    ; str_len
; for() first part ends here
        jmp     .L2
; for() body begins here
.L5:
        cmp     BYTE PTR [rbp-9], 13      ; c=='\r'?
        je      .L3
        cmp     BYTE PTR [rbp-9], 10      ; c=='\n'?
        jne     .L4
.L3:
```

```
        mov     rax, QWORD PTR [rbp-8]    ; str_len
        lea     rdx, [rax-1]              ; EDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]   ; s
        add     rax, rdx                  ; RAX=s+str_len-1
        mov     BYTE PTR [rax], 0         ; s[str_len-1]=0
; for() body ends here
; for() third part begins here
        sub     QWORD PTR [rbp-8], 1      ; str_len--
; for() third part ends here
.L2:
; for() second part begins here
        cmp     QWORD PTR [rbp-8], 0      ; str_len==0?
        je      .L4                       ; exit then
; check second clause, and load "c"
        mov     rax, QWORD PTR [rbp-8]    ; RAX=str_len
        lea     rdx, [rax-1]              ; RDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]   ; RAX=s
        add     rax, rdx                  ; RAX=s+str_len-1
        movzx   eax, BYTE PTR [rax]       ; AL=s[str_len-1]
        mov     BYTE PTR [rbp-9], al      ; store loaded char into "c"
        cmp     BYTE PTR [rbp-9], 0       ; is it zero?
        jne     .L5                       ; yes? exit then
; for() second part ends here
.L4:
; return "s"
        mov     rax, QWORD PTR [rbp-24]
        leave
        ret
```

笔者在程序中增加了注释。执行完长度计算函数 strlen()后，控制权将传递给标号为 L2 的语句。接着注意检查两个条件表达式。如果第一判断条件表达式为真，也就是说如果长度为 0（str_len 的值为 0），那么计算机将不再检测第二个条件判断表达式。这种特性又称为“逻辑短路”。

概括地说，这个函数的执行流程如下：

- 运行 for()语句的第一部分，也就是调用 strlen()函数的循环初始化指令。
- 跳转到标号 L2；检测循环条件是否成立。
- 跳转到标号 L5，进入循环体；
- 再执行 for()语句，如果条件不成立，则直接退出。
- 执行 for()语句的第三部分，将变量 str_len 递减。
- 再次跳转到标号 L2，检测循环条件是否成立、进入循环……周而复始，直到循环条件不成立。
- 跳转到 L4 标号，准备退出。
- 制备返回值，即变量 s。

47.3 x64 下的 GCC 4.9.1 优化

```
str_trim:
        push    rbx
        mov     rbx, rdi
; RBX will always be "s"
        call    strlen
; check for str_len==0 and exit if its so'
        test    rax, rax
        je      .L9
        lea     rdx, [rax-1]
; RDX will always contain str_len-1 value, not str_len
; so RDX is more like buffer index variable
        lea     rsi, [rbx+rdx]        ; RSI=s+str_len-1
        movzx   ecx, BYTE PTR [rsi]   ; load character
        test    cl, cl
```

```
        je      .L9                     ; exit if its zero'
        cmp     cl, 10
        je      .L4
        cmp     cl, 13                  ; exit if its not' '\n' and not '\r'
        jne     .L9
.L4:
; this is weird instruction. we need RSI=s-1 here.
; its possible to get it by' MOV RSI, EBX / DEC RSI
; but this is two instructions instead of one
        sub     rsi, rax
; RSI = s+str_len-1-str_len = s-1
; main loop begin
.L12:
        test    rdx, rdx
; store zero at address s-1+str_len-1+1 = s-1+str_len = s+str_len-1
        mov     BYTE PTR [rsi+1+rdx], 0
; check for str_len-1==0. exit if so.
        je      .L9
        sub     rdx, 1                  ; equivalent to str_len--
; load next character at address s+str_len-1
        movzx   ecx, BYTE PTR [rbx+rdx]
        test    cl, cl                  ; is it zero? exit then
        je      .L9
        cmp     cl, 10                  ; is it '\n'?
        je      .L12
        cmp     cl, 13                  ; is it '\r'?
        je      .L12
.L9:
; return "s"
        mov     rax, rbx
        pop     rbx
        ret
```

GCC 的实现方式更为复杂。在循环体执行前的代码只执行一次，而且它还会检查结束符是不是回车和换行 CR/LF。这难道不是多此一举吗？

一般来说，实现主循环体的流程是这样的：

① 循环开始，检查 CR/LF 结束符，进行判断。

② 保存零字符。

但是，GCC 编译器会将这两步逆序执行。因此，第一步肯定不会是保存零字符，而是进行下述判断：

① 看看第一个字符是不是 CR/LF,如果不是的话，就会退出。

② 循环开始，保存零字符。

③ 根据检查字符是不是 CR/LF 来决定程序的执行。

这样处理之后，主循环体就小了很多，更适用于目前的 CPU 了。这种代码的中间变量不是 str_len，而是 str_len-1。或许是因为后者更适用于用作缓冲区型数据的索引标号（数组下标）。很明显，GCC 注意到了，str_len-1 使用了两次。因此最好的办法是分配一个变量，其值总是比目前的字符串的长度小 1，然后再将其递减（按照变量 str_len 的递减方式递减）。

47.4　ARM64：非优化的 GCC（Linaro）4.9

它将生成近乎“一一对应”的汇编指令。具体代码如下所示。

指令清单 47.4　非优化的 GCC（Linaro[1]）4.9

```
str_trim:
        stp     x29, x30, [sp, -48]!
```

① Linaro 是一家开源的基于 ARM 操作平台的组织，由多家业内公司联合成立。其开发了 ARM 开发工具、Linux 内核以及 Linux 发行版（包括 Android 及 Ubuntu）的主要自动建构系统。

```
        add     x29, sp, 0
        str     x0, [x29,24] ; copy input argument into local stack
        ldr     x0, [x29,24] ; s
        bl      strlen
        str     x0, [x29,40] ; str_len variable in local stack
        b       .L2
; main loop begin
.L5:
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 13          ; is it '\r'?
        beq     .L3
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10          ; is it '\n'?
        bne     .L4             ; goto exit if it is not
.L3:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
        strb    wzr, [x0]       ; write byte at s+str_len-1
; decrement str_len:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        str     x0, [x29,40]
; save X0 (or str_len-1) to local stack
.L2:
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; goto exit then
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
; load byte at address s+str_len-1 to W0
        ldrb    w0, [x0]
        strb    w0, [x29,39] ; store loaded byte to "c"
        ldrb    w0, [x29,39] ; reload it
; is it zero byte?
        cmp     w0, wzr
; goto exit, if its zero or to L5 if its not''
        bne     .L5
.L4:
; return s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

47.5 ARM64:优化 GCC（Linaro）4.9

编译器采用了更为高级的优化技术。在程序开始时，首先调入第一个字符，并和十进制数 10 进行比对

（也就是换行 LF 的数值）。此后，后续字符相继被调入主循环。这种处理方法和本书第 47 章第 3 节中的例子类似。

指令清单 47.5　GCC（Linaro）4.9 的优化

```
str_trim:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     x19, x0
; X19 will always hold value of "s"
        bl      strlen
; X0=str_len
        cbz     x0, .L9         ; goto L9 (exit) if str_len==0
        sub     x1, x0, #1
; X1=X0-1=str_len-1
        add     x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]    ; load byte at address X19+X1=s+str_len-1
; W2=loaded character
        cbz     w2, .L9         ; is it zero? jump to exit then
        cmp     w2, 10          ; is it '\n'?
        bne     .L15
.L12:
; main loop body. loaded character is always 10 or 13 at this moment!
        sub     x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
        add     x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
        strb    wzr, [x2,1]     ; store zero byte at address s+str_len-2+1=s+str_len-1
        cbz     x1, .L9         ; str_len-1==0? goto exit, if so
        sub     x1, x1, #1      ; str_len--
        ldrb    w2, [x19,x1]    ; load next character at address X19+X1=s+str_len-1
        cmp     w2, 10          ; is it '\n'?
        cbz     w2, .L9         ; jump to exit, if its zero'
        beq     .L12            ; jump to begin loop, if its' '\n'
.L15:
        cmp     w2, 13          ; is it '\r'?
        beq     .L12            ; yes, jump to the loop body begin
.L9:
; return "s"
        mov     x0, x19
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
```

47.6　ARM: Keil 6/2013 优化（ARM 模式）

这里我们会再次看到，编译器分配了 ARM 模式下的条件指令，使整个代码更为紧凑。

指令清单 47.6　Keil 6/2013 优化（ARM 模式）

```
str_trim PROC
        PUSH    {r4,lr}
; R0=s
        MOV     r4,r0
; R4=s
        BL      strlen          ; strlen() takes "s" value from R0
; R0=str_len
        MOV     r3,#0
; R3 will always hold 0
|L0.16|
```

```
        CMP     r0,#0           ; str_len==0?
        ADDNE   r2,r4,r0        ; (if str_len!=0) R2=R4+R0=s+str_len
        LDRBNE  r1,[r2,#-1]     ; (if str_len!=0) R1=load byte at address R2-1=s+str_len-1
        CMPNE   r1,#0           ; (if str_len!=0) compare loaded byte against 0
        BEQ     |L0.56|        ; jump to exit if str_len==0 or loaded byte is 0
        CMP     r1,#0xd         ; is loaded byte '\r'?
        CMPNE   r1,#0xa         ; (if loaded byte is not '\r') is loaded byte '\r'?
        SUBEQ   r0,r0,#1        ; (if loaded byte is '\r' or '\n') R0-- or str_len--
        STRBEQ  r3,[r2,#-1]     ; (if loaded byte is '\r' or '\n') store R3 (zero) at
    address R2-1=s+str_len-1
        BEQ     |L0.16|        ; jump to loop begin if loaded byte was '\r' or '\n'
|L0.56|
; return "s"
        MOV     r0,r4
        POP     {r4,pc}
        ENDP
```

47.7 ARM:Keil 6/2013（Thumb 模式）优化

在 Thumb 模式指令集的条件执行指令比 ARM 模式指令集的少，因此这种代码更接近 x86 的指令。但是在程序的 22 和 23 行处的偏移量 0x20 和 0x1f，会令多数人感到匪夷所思。为什么 Keil 6 编译器会分配这些指令？老实说，很难讲。也许这就是 Keil 6 优化进程的诡异之处。不管这种代码多么令人费解，整个程序的功能确实忠实于我们的源代码。

指令清单 47.7 Keil 6/2013（Thumb 模式）优化

```
1     str_trim PROC
1           PUSH    {r4,lr}
2         MOVS r4,r0
4     ; R4=s
5           BL          strlen          ; strlen() takes "s" value from R0
6     ; R0=str_len
7           MOVS    r3,#0
8     ; R3 will always hold 0
9           B       |L0.24|
10    |L0.12|
11          CMP     r1,#0xd         ; is loaded byte '\r'?
12          BEQ     |L0.20|
13          CMP     r1,#0xa         ; is loaded byte '\n'?
14          BNE     |L0.38|         ; jump to exit, if no
15    |L0.20|
16          SUBS    r0,r0,#1        ; R0-- or str_len--
17          STRB    r3,[r2,#0x1f]   ; store 0 at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
18    |L0.24|
19          CMP     r0,#0           ; str_len==0?
20          BEQ     |L0.38|         ; yes? jump to exit
21          ADDS    r2,r4,r0        ; R2=R4+R0=s+str_len
22          SUBS    r2,r2,#0x20     ; R2=R2-0x20=s+str_len-0x20
23          LDRB    r1,[r2,#0x1f]   ; load byte at
24       address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 to R1
25          CMP     r1,#0           ; is loaded byte 0?
26          BNE     |L0.12|         ; jump to loop begin, if its not 0'
27    |L0.38|
28    ; return "s"
29          MOVS    r0,r4
30          POP     {r4,pc}
31          ENDP
```

47.8 MIPS

指令清单 47.8　(IDA)GCC 4.4.5 优化

```
str_trim:
; IDA is not aware of local variable names, we gave them manually:
saved_GP          = -0x10
saved_S0          = -8
saved_RA          = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+saved_RA($sp)
                sw      $s0, 0x20+saved_S0($sp)
                sw      $gp, 0x20+saved_GP($sp)
; call strlen(). input string address is still in $a0, strlen() will take it from there:
                lw      $t9, (strlen & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jalr    $t9
; input string address is still in $a0, put it to $s0:
                move    $s0, $a0 ; branch delay slot
; result of strlen() (i.e, length of string) is in $v0 now
; jump to exit if $v0==0 (i.e., if length of string is 0):
                beqz    $v0, exit
                or      $at, $zero ; branch delay slot, NOP
                addiu   $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
                addu    $a1, $s0, $a1
; $a1 = input string address + $a1 = s+strlen-1
; load byte at address $a1:
                lb      $a0, 0($a1)
                or      $at, $zero ; load delay slot, NOP
; loaded byte is zero? jump to exit if its so':
                beqz    $a0, exit
                or      $at, $zero ; branch delay slot, NOP
                addiu   $v1, $v0, -2
; $v1 = str_len-2
                addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
                li      $a2, 0xD
; skip loop body:
                b       loc_6C
                li      $a3, 0xA ; branch delay slot
loc_5C:
; load next byte from memory to $a0:
                lb      $a0, 0($v1)
                move    $a1, $v1
; $a1=s+str_len-2
; jump to exit if loaded byte is zero:
                beqz    $a0, exit
; decrement str_len:
                addiu   $v1, -1     ; branch delay slot
loc_6C:
; at this moment, $a0=loaded byte, $a2=0xD (CR symbol) and $a3=0xA (LF symbol)
; loaded byte is CR? jump to loc_7C then:
                beq     $a0, $a2, loc_7C
                addiu   $v0, -1     ; branch delay slot
; loaded byte is LF? jump to exit if its not LF':
                bne     $a0, $a3, exit
                or      $at, $zero  ; branch delay slot, NOP
loc_7C:
; loaded byte is CR at this moment
```

```
; jump to loc_5c (loop body begin) if str_len (in $v0) is not zero:
                bnez    $v0, loc_5C
; simultaneously, store zero at that place in memory:
                sb      $zero, 0($a1) ; branch delay slot
; "exit" label was named by me manually:
exit:
                lw      $ra, 0x20+saved_RA($sp)
                move    $v0, $s0
                lw      $s0, 0x20+saved_S0($sp)
                jr      $ra
                addiu   $sp, 0x20     ; branch delay slot
```

S-字头的寄存器就是保存寄存器（saved temporaries）。在过程调用过程中，保存寄存器的值需要保留（被调用方函数保存和恢复），因此$S0 的值保存在局部栈里，在完成任务后恢复其初始状态。

第 48 章　toupper()函数

本章讨论的是将小写字符转换成大写字符的 toupper()函数。这是一个比较常见的函数。

```
char toupper (char c)
{
        if(c>='a' && c<='z')
                return c-'a'+'A';
        else
                return c;
}
```

这里的程序代码中，我们可以看到'a'+'A'这个表达式。这种写法旨在增强程序的可读性。在程序的实际编译过程中，它会被优化为相应的计算指令。[①]

我们知道，小写的英文首字母 a 的 ASCII 码十六进制是 61，也就是十进制的 97；而大写的 A 的 ASCII 码的十六进制则是 41，相当于十进制的 65。很显然，在 ASCII 码表中，两者的差值是 32（0x20）。

实际上，我们从图 48.1 所示的这个 7 位标准 ASCII 码表格就能看得更清楚了。



图 48.1　EMACS 中的 7 位 ASCII 码表

48.1　x64

48.1.1　两个比较操作

非优化的 MSVC 非常直接：为了执行转换到大写字母的目的，程序首先检查输入的符号的十进制数值是否在 97～122 这个范围，也就是小写字母 a～z 的范围内。如果输入值满足这个条件，那么就将它的数值减去十进制的 32，从而得到相应的大写字母所对应的值了。非优化编译时，编译器同样会对代码进行小幅度的优化处理。

指令清单 48.1　x64 下的 MSVC 2013 非优化程序

```
 1 c$ = 8
 2 toupper PROC
 3         mov    BYTE PTR [rsp+8], cl
 4         movsx  eax, BYTE PTR c$[rsp]
 5         cmp    eax, 97
 6         jl     SHORT $LN2@toupper
 7         movsx  eax, BYTE PTR c$[rsp]
 8         cmp    eax, 122
 9         jg     SHORT $LN2@toupper
10         movsx  eax, BYTE PTR c$[rsp]
11         sub    eax, 32
12         jmp    SHORT $LN3@toupper
```

① 然而，如果小心翼翼的话，还是有些编译器即使不用优化的办法，也能输出正确的结果。

```
13          jmp    SHORT $LN1@toupper ; compiler artefact
14  $LN2@toupper:
15          movzx  eax, BYTE PTR c$[rsp] ; unnecessary casting
16  $LN1@toupper:
17  $LN3@toupper: ; compiler artefact
18           ret   0
19  toupper ENDP
```

需要注意的是：程序第 3 行把输入的字节装载到 64 位栈的那条指令。输入值显然是占据了 64 位的低 8 位，而其高位（从第 8 位到第 63 位）则保持不变。我们可以在调试器 debugger 里看到这高 56 位是随机的噪音数据（也就是随机数）。因为所有指令都是以字节为操作对象，因此这种存储方案不会引发问题。第 15 行的最后一个 MOVZX 指令从本地栈里取出一个字节，把它无符号扩展为 32 位数据中，因此这高 56 位的噪音数据都会被填充为零。

而非优化的 GCC 生成的代码几乎相同。

指令清单 48.2 x64 下的 GCC 4.9 非优化代码

```
toupper:
        push    rbp
        mov     rbp, rsp
        mov     eax, edi
        mov     BYTE PTR [rbp-4], al
        cmp     BYTE PTR [rbp-4], 96
        jle     .L2
        cmp     BYTE PTR [rbp-4], 122
        jg      .L2
        movzx   eax, BYTE PTR [rbp-4]
        sub     eax, 32
        jmp     .L3
.L2:
        movzx   eax, BYTE PTR [rbp-4]
.L3:
        pop     rbp
        ret
```

48.1.2 一个比较操作

优化的 MSVC 表现得更好，它只分配一个比较操作指令。

指令清单 48.3 x64 下的 MSVC 2013 优化程序

```
toupper PROC
        lea     eax, DWORD PTR [rcx-97]
        cmp     al, 25
        ja      SHORT $LN2@toupper
        movsx   eax, cl
        sub     eax, 32
        ret     0
$LN2@toupper:
        movzx   eax, cl
        ret     0
toupper ENDP
```

前面已经解释过了，如何用一个比较操作来代替两个（参见本书 42.2.1 节）。

如果用 C/C++重写，相应的源代码如下：

```
int tmp=c-97;

if (tmp>25)
        return c;
else
        return c-32;
```

这里的变量 tmp 应当是一个有符号数。这里的转换过程采用了两个减法指令以及一个比较指令。而原

来的算法是通过两个比较指令加一个减法指令来实现的。

优化的 GCC 算法更好，它不用跳转指令 JUMP，转而采用 CMOVcc 指令（参见 33.1 节）。

指令清单 48.4　x64 下的 GCC 4.9 优化程序

```
1 toupper:
2         lea     edx, [rdi-97] ; 0x61
3         lea     eax, [rdi-32] ; 0x20
4         cmp     dl, 25
5         cmova   eax, edi
6         ret
```

第三行的指令代码首先准备了要减去的数值 0x20（十六进制的 20，十进制的 32），这会给人一种印象，好像减法总会发生似的。而第五行的指令，在不应进行转换（也就是不执行减法）的时候，把原始的输入数据复制给 EAX 寄存器。可见它能在条件不成立的情况下舍弃错误的值。也就是说，编译器为了避免使用条件转移指令而预先进行了减法运算。

48.2　ARM

ARM 模式下的 Keil 优化也只有一个比较指令。

指令清单 48.5　ARM 模式下的 Keil 6/2013 优化程序

```
toupper PROC
        SUB     r1,r0,#0x61
        CMP     r1,#0x19
        SUBLS   r0,r0,#0x20
        ANDLS   r0,r0,#0xff
        BX      lr
        ENDP
```

从这里的程序我们可以看到，如果 R1 的值小于或等于 0x19（十六进制的 19），那么就会执行 SUBLS 指令和 ANDLS 指令。这两条指令实际执行的是转换大写字符的操作。

而 Thumb 模式下的 Keil 优化也可以通过单条比较指令进行大写转换。

指令清单 48.6　Thumb 模式下的 Keil 6/2013 的优化指令

```
toupper PROC
        MOVS    r1,r0
        SUBS    r1,r1,#0x61
        CMP     r1,#0x19
        BHI     |L0.14|
        SUBS    r0,r0,#0x20
        LSLS    r0,r0,#24
        LSRS    r0,r0,#24
|L0.14|
        BX      lr
        ENDP
```

最后的两条指令（LSLS 和 LSRS）的整体作用其实就是“AND reg,0xFF”。用 C/C++表达式书写的话，它就可以表示为“(i<<24)>>24”，也就是先左移动 24 位，然后右移 24 位。显然，Thumbe 模式下的 Keil 认为，它们这里的 2 个双字节的指令比“AND reg,0xFF”的运行效率更高（后者要首先将 0xFF 载入寄存器，然后做 And 与操作）。

48.2.1　ARM64 下的 GCC

指令清单 48.7　ARM64 下的非优化 GCC 4.9

```
toupper:
```

```
        sub     sp, sp, #16
        strb    w0, [sp,15]
        ldrb    w0, [sp,15]
        cmp     w0, 96
        bls     .L2
        ldrb    w0, [sp,15]
        cmp     w0, 122
        bhi     .L2
        ldrb    w0, [sp,15]
        sub     w0, w0, #32
        uxtb    w0, w0
        b       .L3
.L2:
        ldrb    w0, [sp,15]
.L3:
        add     sp, sp, 16
        ret
```

指令清单 48.8 ARM64 下的优化 GCC 4.9

```
toupper:
        uxtb    w0, w0
        sub     w1, w0, #97
        uxtb    w1, w1
        cmp     w1, 25
        bhi     .L2
        sub     w0, w0, #32
        uxtb    w0, w0
.L2:
        ret
```

48.3 总结

本章演示的优化技术已经属于十分常见的编译器优化技术。逆向工程分析人员会频繁遇到这种类型的汇编指令。

第 49 章　不正确的反汇编代码

在实际工程中，反编译工程师们经常会遇到一些不正确的反汇编代码。我们来看看如何处理。

49.1　x86 环境下的从一开始错误的反汇编

与 opcode 等长的 ARM 以及 MIPS 指令集（它们每个指令的 opcode 无非就是 2 个字节或 4 个字节长）不同，x86 构架下的指令长度不尽相同。因此，若从 x86 程序的中间开始解析指令，无论什么分析工具都会分析出错误的结果。

比如说：

```
add     [ebp-31F7Bh], cl
dec     dword ptr [ecx-3277Bh]
dec     dword ptr [ebp-2CF7Bh]
inc     dword ptr [ebx-7A76F33Ch]
fdiv    st(4), st
db 0FFh
dec     dword ptr [ecx-21F7Bh]
dec     dword ptr [ecx-22373h]
dec     dword ptr [ecx-2276Bh]
dec     dword ptr [ecx-22B63h]
dec     dword ptr [ecx-22F4Bh]
dec     dword ptr [ecx-23343h]
jmp     dword ptr [esi-74h]
xchg    eax, ebp
clc
std
db 0FFh
db 0FFh
mov     word ptr [ebp-214h], cs ; <- disassembler finally found right track here
mov     word ptr [ebp-238h], ds
mov     word ptr [ebp-23Ch], es
mov     word ptr [ebp-240h], fs
mov     word ptr [ebp-244h], gs
pushf
pop     dword ptr [ebp-210h]
mov     eax, [ebp+4]
mov     [ebp-218h], eax
lea     eax, [ebp+4]
mov     [ebp-20Ch], eax
mov     dword ptr [ebp-2D0h], 10001h
mov     eax, [eax-4]
mov     [ebp-21Ch], eax
mov     eax, [ebp+0Ch]
mov     [ebp-320h], eax
mov     eax, [ebp+10h]
mov     [ebp-31Ch], eax
mov     eax, [ebp+4]
mov     [ebp-314h], eax
call    ds:IsDebuggerPresent
mov     edi, eax
lea     eax, [ebp-328h]
push    eax
call    sub_407663
pop     ecx
```

```
test    eax, eax
jnz     short loc_402D7B
```

这里我们可以看到，解析出来的前几条指令毫无道理。但是后来反汇编工具逐渐走上了正轨。

49.2 随机噪音，怎么看起来像反汇编指令？

一般来说，我们可以从以下三点来判断，一段程序是由随机代码组成的还是正常的程序段：

- 不寻常的指令集合。一般来说，x86 最常见的指令是 PUSH、MOV、CALL 等。如果遇到了大杂烩式的稀有指令大拼盘（浮点运算 FPU 指令、输入输出 IN/OUT 指令和很少见的一些系统指令混在一起），多数情况就是反汇编过程出问题了。
- 又大又像是随机数的数值、偏移量以及立即数。
- 转移指令的偏移量不合逻辑，经常跳转到其他指令块的中间。

指令清单 49.1 x86 模式下的随机数噪声

```
    mov     bl, 0Ch
    mov     ecx, 0D38558Dh
    mov     eax, ds:2C869A86h
    db      67h
    mov     dl, 0CCh
    insb
    movsb
    push    eax
    xor     [edx-53h], ah
    fcom    qword ptr [edi-45A0EF72h]
    pop     esp
    pop     ss
    in      eax, dx
    dec     ebx
    push    esp
    lds     esp, [esi-41h]
    retf
    rcl     dword ptr [eax], cl
    mov     cl, 9Ch
    mov     ch, 0DFh
    push    cs
    insb
    mov     esi, 0D9C65E4Dh
    imul    ebp, [ecx], 66h
    pushf
    sal     dword ptr [ebp-64h], cl
    sub     eax, 0AC433D64h
    out     8Ch, eax
    pop     ss
    sbb     [eax], ebx
    aas
    xchg    cl, [ebx+ebx*4+14B31Eh]
    jecxz   short near ptr loc_58+1
    xor     al, 0C6h
    inc     edx
    db      36h
    pusha
    stosb
    test    [ebx], ebx
    sub     al, 0D3h ; 'L'
    pop     eax
    stosb

loc_58: ; CODE XREF: seg000:0000004A
    test    [esi], eax
```

```
    inc     ebp
    das
    db      64h
    pop     ecx
    das
    hlt

    pop     edx
    out     0B0h, al
    lodsb
    push    ebx
    cdq
    out     dx, al
    sub     al, 0Ah
    sti
    outsd
    add     dword ptr [edx], 96FCBE4Bh
    and     eax, 0E537EE4Fh
    inc     esp
    stosd
    cdq
    push    ecx
    in      al, 0CBh
    mov     ds:0D114C45Ch, al
    mov     esi, 659D1985h
```

指令清单 49.2　x86-64 下的随机数噪声

```
    lea     esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
    lea     ecx, cs:0FFFFFFFFB2A6780Fh
    mov     al, 96h
    mov     ah, 0CEh
    push    rsp
    lods    byte ptr [esi]

    db 2Fh ; /

    pop     rsp
    db      64h
    retf    0E993h

    cmp     ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]

    db 9Ah

    rcr     byte ptr [rax+1Dh], cl
    lodsd
    xor     [rbp+6CF20173h], edx
    xor     [rbp+66F8B593h], edx
    push    rbx
    sbb     ch, [rbx-0Fh]
    stosd
    int     87h
    db      46h, 4Ch
    out     33h, rax
    xchg    eax, ebp
    test    ecx, ebp
    movsd
    leave
    push    rsp
```

```
    db 16h

    xchg    eax, esi
    pop     rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
    out     dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
    pop     rbp
```

指令清单 49.3 ARM 模式下的随机数噪声

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC],#-0x964
LDCGE   p6, c14, [R0],#0x168
STCCSL  p9, c9, [LR],#0x14C
CMNHIP  PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2,c15,c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA   R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB    SP, [R8],PC,ROR#18
LDCCS   p9, c13, [R6,#0x1BC]
LDRGE   R8, [R9,#0x66E]
STRNEB  R5, [R8],#-0x8C3
STCCSL  p15, c9, [R7,#-0x84]
RSBLS   LR, R2, R11,ASR LR
SVCGT   0x9B0362
SVCGT   0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR     R0, [R3],#-0xCE4
LDCGT   p15, c8, [R1,#0x2CC]
LDRCCB  R1, [R11],-R7,ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
USATNE  R10, #7, SP,LSL#11
LDRGEB  LR, [R1],#0xE56
STRPLT  R9, [LR],#0x567
LDRLT   R11, [R1],#-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP,LSL#25
LDCL    p8, c14, [R12,#-0x288]
STCNEL  p2, c6, [R6,#-0xBC]!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4,LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7,ASR R0
LDRMI   R7, [LR],R3,LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
```

```
EORMI   R12, R6, #0xC5
MCRRCS  p1, 0xF, R1,R3,c2
```

指令清单 49.4　ARM 的 Thumb 模式下的随机数噪声

```
    LSRS    R3, R6, #0x12
    LDRH    R1, [R7,#0x2C]
    SUBS    R0, #0x55 ; 'U'
    ADR     R1, loc_3C
    LDR     R2, [SP,#0x218]
    CMP     R4, #0x86
    SXTB    R7, R4
    LDR     R4, [R1,#0x4C]
    STR     R4, [R4,R2]
    STR     R0, [R6,#0x20]
    BGT     0xFFFFFF72
    LDRH    R7, [R2,#0x34]
    LDRSH   R0, [R2,R4]
    LDRB    R2, [R7,R2]

    DCB 0x17
    DCB 0xED

    STRB    R3, [R1,R1]
    STR     R5, [R0,#0x6C]
    LDMIA   R3, {R0-R5,R7}
    ASRS    R3, R2, #3
    LDR     R4, [SP,#0x2C4]
    SVC     0xB5
    LDR     R6, [R1,#0x40]
    LDR     R5, =0xB2C5CA32
    STMIA   R6, {R1-R4,R6}
    LDR     R1, [R3,#0x3C]
    STR     R1, [R5,#0x60]
    BCC     0xFFFFFF70
    LDR     R4, [SP,#0x1D4]
    STR     R5, [R5,#0x40]
    ORRS    R5, R7

loc_3C ; DATA XREF: ROM:00000006
    B       0xFFFFFF98
```

指令清单 49.5　MIPS（小端）下的随机数噪声

```
    lw      $t9, 0xCB3($t5)
    sb      $t5, 0x3855($t0)
    sltiu   $a2, $a0, -0x657A
    ldr     $t4, -0x4D99($a2)
    daddi   $s0, $s1, 0x50A4
    lw      $s7, -0x2353($s4)
    bgtzl   $a1, 0x17C5C

    .byte 0x17
    .byte 0xED
    .byte 0x4B # K
    .byte 0x54 # T

    lwc2    $31, 0x66C5($sp)
    lwu     $s1, 0x10D3($a1)
    ldr     $t6, -0x204B($zero)
    lwc1    $f30, 0x4DBE($s2)
    daddiu  $t1, $s1, 0x6BD9
    lwu     $s5, -0x2C64($v1)
    cop0    0x13D642D
    bne     $gp, $t4, 0xFFFF9EF0
    lh      $ra, 0x1819($s1)
```

```
sdl     $fp, -0x6474($t8)
jal     0x78C0050
ori     $v0, $s2, 0xC634
blez    $gp, 0xFFFEA9D4
swl     $t8, -0x2CD4($s2)
sltiu   $a1, $k0, 0x685
sdc1    $f15, 0x5964($at)
sw      $s0, -0x19A6($a1)
sltiu   $t6, $a3, -0x66AD
lb      $t7, -0x4F6($t3)
sd      $fp, 0x4B02($a1)
```

我们必须注意的一点是：一些编写良好的解压包或者加密程序代码（也包括一些变形代码），从代码来看也很像是随机数指令序列，然而一旦运行起来则是非常正确的。

第50章 混淆技术

代码混淆技术是一种用于阻碍逆向工程分析人员解析程序代码（或功能）的指令处理技术。

50.1 字符串变换

我们在第57章可看到，在逆向工程的过程中字符串经常起到路标的作用。注意到这个问题的编程人员就会着手解决这个问题。他们会采用一些变换的手法，让他人不能直接通过IDA或者16进制编辑器直接搜索到字符串原文。

这里我们举一个简单的例子。

比方说，我们可以这样构造一个字符串：

```
mov     byte ptr [ebx], 'h'
mov     byte ptr [ebx+1], 'e'
mov     byte ptr [ebx+2], 'l'
mov     byte ptr [ebx+3], 'l'
mov     byte ptr [ebx+4], 'o'
mov     byte ptr [ebx+5], ' '
mov     byte ptr [ebx+6], 'w'
mov     byte ptr [ebx+7], 'o'
mov     byte ptr [ebx+8], 'r'
mov     byte ptr [ebx+9], 'l'
mov     byte ptr [ebx+10], 'd'
```

当然还有更为复杂的构造方法：

```
mov     ebx, offset username
cmp     byte ptr [ebx], 'j'
jnz     fail
cmp     byte ptr [ebx+1], 'o'
jnz     fail
cmp     byte ptr [ebx+2], 'h'
jnz     fail
cmp     byte ptr [ebx+3], 'n'
jnz     fail
jz      it_is_john
```

不管是以上的哪种情况，我们用十六进制的文本编译器都不能直接搜索到字符串原文。

实际上这两种方法适用于那些无法利用数据段构造数据的情景。因为它们可以在文本段直接构造数据，所以也常见于各种PIC和shellcode。

另外，笔者还见过这样使用sprintf()函数的：

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

代码看起来很诡异，但是作为一个简单的反编译技巧来说，也不失为一个好办法。

加密存储字符串是另一种常见的处理方法。只是这样一来，就要在每次使用前对字符串解密。相关的例子可以参看第78章第2节。

50.2 可执行代码

50.2.1 插入垃圾代码

在正常执行指令序列中插入一些虽然可被执行但是没有任何作用的指令，本身就是一种代码混淆技术。

我们可以看一个简单的例子。

指令清单 50.1 源代码

```
add     eax, ebx
mul     ecx
```

指令清单 50.2 采用混淆技术后的代码

```
xor     esi, 011223344h ; garbage
add     esi, eax        ; garbage
add     eax, ebx
mov     edx, eax        ; garbage
shl     edx, 4          ; garbage
mul     ecx
xor     esi, ecx        ; garbage
```

在程序代码中插入的混淆指令，调用了源程序不会使用的 ESI 和 EDX 寄存器。混淆代码利用了源程序的中间之后，大幅度地增加了反编译的难度，何乐不为呢？

50.2.2 用多个指令组合代替原来的一个指令

- MOV op1,op2 这条指令，可以使用组合指令代替：PUSH op2, POP op1。
- JMP label 指令可以用 PUSH label, RET 这个指令对代替。反编译工具 IDA 不能识别出这种 label 标签的调用结构。
- CALL label 指令则可以用以下三个指令代替：PUSH {call 指令后面的那个 label}、PUSH label 和 RET 指令。
- PUSH op 可以用以下的指令代替。SUB ESP,4 或 8; MOV [ESP],操作符。

50.2.3 始终执行或者从来不会执行的代码

在下面的代码中，假定此处 ESI 的值肯定是 0，那么我们可以在 fake luggage 处插入任意长度和复杂度的指令，以达到混淆的目的。这种混淆技术称为不透明谓词（opaque predicate）。

```
mov     esi, 1
...     ; some code not touching ESI
dec     esi
...     ; some code not touching ESI
cmp     esi, 0
jz      real_code
; fake luggage
real_code:
```

我们还可以看看其他的例子（同样，我们假定 ESI 始终会是零）。

```
add     eax, ebx          ; real code
mul     ecx               ; real code
add     eax, esi          ; opaque predicate. XOR, AND or SHL, etc, can be here instead of ADD.
```

50.2.4　把指令序列搞乱

```
instruction 1
instruction 2
instruction 3
```

上面的 3 行正常执行的指令序列可以用如下所示的复杂结构代替：

```
begin:          jmp     ins1_label

ins2_label:     instruction 2
                jmp     ins3_label

ins3_label:     instruction 3
                jmp     exit:

ins1_label:     instruction 1
                jmp     ins2_label
exit:
```

50.2.5　使用间接指针

```
dummy_data1     db      100h dup (0)
message1        db      'hello world',0

dummy_data2     db      200h dup (0)
message2        db      'another message',0

func            proc
                ...
                mov     eax, offset dummy_data1 ; PE or ELF reloc here
                add     eax, 100h
                push    eax
                call    dump_string
                ...
                mov     eax, offset dummy_data2 ; PE or ELF reloc here
                add     eax, 200h
                push    eax
                call    dump_string
                ...
func            endp
```

这个程序执行时，我们只能在 IDA 编译工具中看到 dummy_data1 和 dummy_data2 的 reference(调用信息)。它不能正常反馈字符串正体 message1 和 message2 的调用信息。

全局变量或者函数也可以这样混淆。

50.3　虚拟机以及伪代码

编程人员可以构建其自身的 PL 或者 ISA 解释器（类似 VB.NET 或者 Java）。这样的话，反编译者就得花很多时间来理解这些解释器指令的意义以及细节。当然，他们基本上必须开发一种专用的反汇编或者反编译工具了。

50.4　一些其他的事情

笔者对 Tiny C 编译器做了一些修改，然后用它编译了一个小程序（参见 url: http://go.yurichev.com/17220）。

请分析该程序的具体功能（参见 G.1.13）。

50.5 练习题

50.5.1 练习 1

这是一个很短的程序，采用打了补丁的 Tiny C 编译器编译。看看它能做什么？
答案请参见 G.1.15。

第 51 章　C++

51.1　类

51.1.1　一个简单的例子

从汇编层面看，C++类（class）的组织方式和结构体数据完全一致。
我们演示一个含有两个变量、两个结构体以及一个方法的类型数据：

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

MSVC –x86

使用 MSVC 编译上述程序，可得到如下所示的代码。

指令清单 51.1　MSVC

```
_c2$ = -16 ; size = 8
_c1$ = -8 ; size = 8
_main PROC
    push ebp
```

```
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

我们来看看程序是如何实现的。程序为每个对象（类的实例）分配了 8 个字节内存，正好能存储 2 个变量。

在初始化 c1 时，编译器调用了无参构造函数??0c@@QAE@XZ。在初始化另一个实例（即 c2）时，编译器向有参构造函数??0c@@QAE@HH@Z 传递了 2 个参数。

在传递整个类对象（C++的术语是 this）的指针时，this 指针通过 ECX 寄存器传递给被调用方函数。这种调用规范应当符合 thiscall 规范，详细讲解请参阅本书的 51.1.1 节。

MSVC 通过 ECX 寄存器传递 this 指针。不过，这种调用约定并没有统一的技术规范。GCC 编译器以传递第一个函数的参数的方式传递 this 指针，其他的编译器多数都遵循了 GCC 的 thiscall 规范。

为什么这些函数有这些很奇怪的名字（见上面）？其实这是编译器对函数名称进行的名称改编（name mangling）的结果。

C++的类可能包含同名的但是参数不同的方法（即类成员函数）。这就是所谓的多态性。当然，不同的类可以有重名却不同的方法。

名称改编（name mangling）是一种在编译过程中，用 ASCII 字符串将函数、变量的名称重新改编的机制。改编后的方法（类成员函数）名称就被用作该程序内部的函数名。这完全是因为编译器的 Linker 和加载 DLL 的 OS 装载器均不能识别 C++或 OOP（面向对象的编程语言）的数据结构。

函数 dump()调用了两次。我们再来看看构造函数的指令代码。

指令清单 51.2 MSVC

```
_this$ = -4        ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8     ; size = 4
_b$ = 12    ; size = 4
```

```
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _a$[ebp]
    mov  DWORD PTR [eax], ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR _b$[ebp]
    mov  DWORD PTR [edx+4], eax
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c
```

构造函数本身就是一种函数，它们使用 ECX 寄存器存储结构体的指针，然后将指针复制到其自己的局部变量里。当然，第二步并不是必须的。

从 C++的标准（ISO13, P.12.1）可知，构造函数不必返回返回值。事实上，从指令层面来来看，构造函数的返回值是一个新建立的对象的指针，即 this 指针。

现在我们来看看 dump()。

指令清单 51.3　MSVC

```
_this$ = -4             ; size = 4
?dump@c@@QAEXXZ PROC  ; c::dump, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax+4]
    push ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR [edx]
    push eax
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    mov  esp, ebp
    pop  ebp
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

很简单，dump()函数从 ECX 寄存器读取一个指向数据结构（这个结构体含有 2 个 int 型数据）的指针，然后再把这两个整型数据传递给 printf()函数。

如果指定优化编译参数/Ox 的话，那么 MSVC 能够生成更短的可执行程序。

指令清单 51.4　MSVC （优化编译）

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    mov  eax, ecx
    mov  DWORD PTR [eax], 667
    mov  DWORD PTR [eax+4], 999
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8  ; size = 4
_b$ = 12 ; size = 4
```

```
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov  edx, DWORD PTR _b$[esp-4]
    mov  eax, ecx
    mov  ecx, DWORD PTR _a$[esp-4]
    mov  DWORD PTR [eax], ecx
    mov  DWORD PTR [eax+4], edx
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

优化编译生产的代码就这么短。我们需要注意的是：在调用构造函数之后，栈指针不是通过“add esp, x”指令到恢复其初始状态的。另一方面，构造函数的最后一条指令是指令 ret 8 而不是 RET。

这是因为此处不仅遵循了 thiscall 调用规范（参见 51.1.1 节），而且还同时遵循 stdcall 调用规范（64.2 节）。Stdcall 规范约定：应当由被调用方函数（而不是由调用方函数）恢复参数栈的初始状态。构造函数（也是本例中的被调用方函数）使用“add ESP，x”的指令把本地栈释放 x 字节，然后把程序控制权传递给调用方函数。

读者还可以参考本书第 64 章，了解各调用规范的详细约定。

必须指出的是，编译器自身能决定调用构造函数和析构函数。我们则可以通过 C++语言的编程基础找到程序中的相应指令。

MSVC-x86-64

在 x86-64 环境里的 64 位应用程序使用 RCX、RDX、R8 以及 R9 这 4 个寄存器传递函数的前 4 项参数，而其他的参数则通过栈传递。然而，在调用那些涉及类成员函数的时候，编译器会通过 RCX 寄存器传递类对象的 this 指针，用 RDX 寄存器传递函数的第一个参数，依此类推。我们可以在类成员函数 c(int a,int b) 中看到这一点。

指令清单 51.5 x64 下的 MSVC 2012 优化

```
; void dump()

?dump@c@@QEAAXXZ PROC ; c::dump
    mov     r8d, DWORD PTR [rcx+4]
    mov     edx, DWORD PTR [rcx]
    lea     rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ; '%d; %d'
    jmp     printf
?dump@c@@QEAAXXZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov     DWORD PTR [rcx], edx ; 1st argument: a
    mov     DWORD PTR [rcx+4], r8d ; 2nd argument: b
    mov     rax, rcx
    ret     0
??0c@@QEAA@HH@Z ENDP ; c::c

; default ctor
```

```
??0c@@QEAA@XZ PROC ; c::c
    mov     DWORD PTR [rcx], 667
    mov     DWORD PTR [rcx+4], 999
    mov     rax, rcx
    ret     0
??0c@@QEAA@XZ ENDP ; c::c
```

64 位环境下[①]的 int 型数据依然是 32 位数据。因此，上述程序仍然使用 32 位寄存器传递整型数据。

类成员函数 dump()还使用了 JMP printf 指令取代了 RET 指令。我们在 13.1.1 节中已经见过这个 hack 了。

GCC-x86

除了个别不同之处以外，GCC 4.4.1 的编译方式和 MSVC 2012 的编译手段几乎一样。

指令清单 51.6 GCC 4.4.1

```
    public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Ev
    mov  [esp+20h+var_18], 6
    mov  [esp+20h+var_1C], 5
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Eii
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    mov  eax, 0
    leave
    retn
main endp
```

这里我们可以看到另外一种风格的名称改编方法，当然这应当是 GNU[②]的专用风格。必须注意的是，类对象的指针是以函数的第一个参数的方式传递的。当然，编程人员看不到这些技术细节。

第一个构造函数是：

```
                public _ZN1cC1Ev ; weak
_ZN1cC1Ev       proc near               ; CODE XREF: main+10

arg_0           = dword ptr 8

                push    ebp
                mov     ebp, esp
```

① 很明显，这是为了使 64 位系统向下兼容 32 位的 C/C++应用程序。

② 这里有一个比较好的文档，它描述了各种编译器的不同的命名混淆规则。

```
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax], 667
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax+4], 999
                pop     ebp
                retn
_ZN1cC1Ev       endp
```

它通过外部传来的第一个参数获取结构体的指针，然后在相应地址修改了 2 个数值。

第二个构造函数是：

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = dword ptr 10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_4]
                mov     [eax], edx
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_8]
                mov     [eax+4], edx
                pop     ebp
                retn
_ZN1cC1Eii      endp
```

上述函数的程序逻辑与下面的 C 语言代码大致相当：

```
void ZN1cC1Eii (int *obj, int a, int b)
{
  *obj=a;
  *(obj+1)=b;
};
```

结果是完全可以预期的。

现在我们来看看 dump()函数：

```
                public _ZN1c4dumpEv
_ZN1c4dumpEv    proc near

var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
arg_0           = dword ptr 8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+arg_0]
                mov     edx, [eax+4]
                mov     eax, [ebp+arg_0]
                mov     eax, [eax]
                mov     [esp+18h+var_10], edx
                mov     [esp+18h+var_14], eax
                mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
                call    _printf
                leave
                retn
_ZN1c4dumpEv    endp
```

这个函数在其内部表征中只有一个参数。这个参数就是这个对象的 this 指针。

本函数可以用 C 语言重写如下：

```
void ZN1c4dumpEv (int *obj)
{
  printf ("%d; %d\n", *obj, *(obj+1));
};
```

综合本节的各例可知，MSVC 和 GCC 的区别在于函数名的名称编码风格以及传递 this 指针的具体方式（MSVC 通过 ECX 传递，而 GCC 以函数的第一个参数的方式传递）。

GCC-x86-64

在编译 64 位应用程序的时候，GCC 通过 RDI、RSI、RDX、RCX、R8 以及 R9 这几个寄存器传递函数的前 6 个参数。它通过 RDI 寄存器，以第一个函数参数的形式传递 this 指针。另外，整数型 int 数据依然是 32 位数据。它还会不时使用转移指令 JMP 替代 RET 指令。

指令清单 51.7　x64 下的 GCC 4.4.6

```
; default ctor

_ZN1cC2Ev:
    mov  DWORD PTR [rdi], 667
    mov  DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_ZN1cC2Eii:
    mov  DWORD PTR [rdi], esi
    mov  DWORD PTR [rdi+4], edx
    ret

; dump()

_ZN1c4dumpEv:
    mov  edx, DWORD PTR [rdi+4]
    mov  esi, DWORD PTR [rdi]
    xor  eax, eax
    mov  edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp  printf
```

51.1.2　类继承

继承而来的类与前文的简单结构体相似，但是它可以对父类进行扩展。

我们先来看一个简单的例子：

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
    private:
        int width, height, depth;
```

```
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, ↙
    ↳ height, depth);
        };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};
```

我们共同关注 dump()函数（又称方法）以及 object::print_color()的指令代码，重点分析 32 位环境下有关数据类型的内存存储格局。

下面所示的是几个不同的类的 dump()方法，它们是在启用优化编译选项/Ox 和/Ob0 后，由 MSVC 2008 产生的代码。

指令清单 51.8 MSVC 2008 带参数/Ob0 的优化

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; `string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax

; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
    call _printf
    add  esp, 8
    ret  0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

指令清单 51.9 MSVC 2008 带参数/Ob0 的优化

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

指令清单 51.10 MSVC 2008 带参数/Ob0 的优化

```
?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
    push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5radius@
    call _printf
    add esp, 12
    ret 0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

因此，这里是内存的基本排列：

① 父类 object 对象的存储格局如下所示。

offset	description
+0x0	int color

② 继承类对象：box 和 sphere（分别为盒子和球体）的存储格局分别如下面两张表所示。

box

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere

offset	description
+0x0	int color
+0x4	int radius

下面分析一下函数主体 main()。

指令清单 51.11 MSVC 2008 带参数/Ob0 的优化

```
PUBLIC _main
_TEXT SEGMENT
```

```
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub  esp, 24
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+40]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+32]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?dump@box@@QAEXXZ ; box::dump
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?dump@sphere@@QAEXXZ ; sphere::dump
    xor  eax, eax
    add  esp, 24
    ret  0
_main ENDP
```

继承类必须在其基（父）类字段的后面加入自己的字段，因此基类和继承类的类成员函数可以共存。

当程序调用类成员对象 object::print_color()时，指向对象 box 和 sphere 的指针是通过 this 指针传递的。由于在所有继承类和基类中 color 字段的偏移量固定为 0（offset+0x0），所有类对象的类成员函数 object::print_color 都可以正常运行。

因此，无论是基类还是继承类调用 object::print_color()，只要该方法所引用的字段的相对地址固定不变，那么该方法就可以正常运行。

假如基于 box 类创建一个继承类，那么编译器就会在变量 depth 的后面追加您所添加新的变量，以确保基类 box 的各字段的相对地址在其继承类中固定不变。

因此，当父类为 box 类的各继承类在调用各自的方法 box::dump()时，它们都能检索到 color、width、height 以及 depths 字段的正确地址。因为各字段的相对地址不会发生变化。

GCC 生成的指令代码与 MSVC 生成的代码几乎相同。唯一的区别是：GCC 不会使用 ECX 寄存器传递 this 指针，它会以函数的第一个参数的传递方式传递 this 指针。

51.1.3 封装

封装（encapsulation）的作用是：把既定的数据和方法限定为类的私有信息，使得其他调用方只能访问类所定义的公共方法和公共数据、不能直接访问被封装起来的私有对象。

在指令层面，到底有没有划分私有对象和公开对象的界限呢？

其实完全没有。

我们来看看这一个简单的例子：

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
```

```
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width,height, depth);
        };
};
```

我们启用 MSVC 2008 的优化选项/Ox 和/Ob0 编译上述程序，再查看类函数 box::dump()的代码。

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

下面这个表格显示了类的变量在内存中的偏移量的分布情况。

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

所有字段都是无法被其他函数直接访问的私有变量。但是，既然我们知道了这个对象的内存存储格局，能不能写出一个修改这些变量的程序呢？

为此，我们可以构造一个名称为 hack_oop_encapsulation()的函数。如果不做调整的话，直接访问既定字段的源程序大致会是：

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code cant be compiled':
                // "error C2248: 'box::width' : cannot access private member declared in class 'box'"
};
```

当然，上述代码不可能被成功编译出来。然而，只要把 box 的数据类型强制转换为整型数组的话，我们就可以通过编译并且直接修改相应字段。

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

上述函数的功能十分简单：它将输入数据视为整型数组，然后将数组的第二个元素、一个整型 int 值修改为 123 。

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
```

```
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

接下来，我们验证一下它的功能。

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

运行的结果为：

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

我们可以看到，封装只能够在编译阶段保护类的私有对象。虽然C++编译器禁止外部代码直接访问那些被明确屏蔽的内部对象，但是通过适当的hack技术，我们确实能够突破编译器的限制策略。

51.1.4 多重继承

多重继承，指的是一个类可以同时继承多个父类的字段和方法。

我们还是写个简单的例子：

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
```

```
            return density;
        };
        void dump()
        {
            printf ("this is solid_object. density=%d\n", density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
        void dump()
        {
            printf ("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, ↙
    ↳ height, depth, density);
        };
        int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};
```

在启用其优化选项（/Ox 和/Ob0）后，我们使用 MSVC 编译上述程序，重点观察 box::dump()、solid_object::dump()以及 solid_box::dump()这 3 个类成员函数。

指令清单 51.12　带/Ob0 参数的 MSVC 2008 优化程序

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    mov  edx, DWORD PTR [ecx+4]
    push eax
    mov  eax, DWORD PTR [ecx]
    push edx
    push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5height?$DN?$CFd@
    call _printf
    add  esp, 16
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

指令清单 51.13　带/Ob0 参数的 MSVC 2008 优化程序

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
```

```
    mov  eax, DWORD PTR [ecx]
    push eax
; 'this is solid_object. density=%d', 0aH
    push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?$CFd@
    call _printf
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

指令清单 51.14 带/Ob0 参数的 MSVC 2008 优化程序

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
    push OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd?0?5hei@
    call _printf
    add  esp, 20
    ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

上述 3 个类对象的内存分布如下：

① 类 box。如下表所示。

offset	description
+0x0	width
+0x4	height
+0x8	depth

② 类 solid_object。如下表所示。

offset	description
+0x0	density

③ 类 solid_box，可以看成是以上两个类的联合体。如下表所示。

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

以上图表采用了偏移量与对应变量的方式展现 3 个类对象的内存存储结构。图中一共出现了 4 个变量，即长 width、高 height、宽 depth 以及密度 density。

体积函数 get_volume()的代码如下所示。

指令清单 51.15 带/Ob0 参数的 MSVC 2008 优化程序

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
```

```
    mov  eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret  0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

密度函数 get_density()的代码如下所示。

指令清单 51.16　带/Ob0 参数的 MSVC 2008 优化程序

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

最有意思的是 solod_box::get_weight()重量函数。

指令清单 51.17　带/Ob0 参数的 MSVC 2008 优化程序

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    push edi
    lea  ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov  ecx, esi
    mov  edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop  edi
    pop  esi
    ret  0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

函数 get_weight()（计算重量）只调用了两个方法。在调用 get_volume()（计算体积）时，它传递了 this 指针。而在调用 get_density()（密度）函数时，它传递的地址是“this 指针＋12 个字节”。后面这个地址对应的是 solid_box 类的 solid_object 字段。

因此，solid_object::get_density()方法认为，它处理的是常规的 solid_object 类，而 box::get_volume()则可以正常访问原有数据类型的 3 个变量，如同直接操作 box 类一样。

因此，我们可以相信：继承了其他的、多个类而生成的类对象，在内存之中就是一种联合体型的数据结构。它继承了原有父类的全部字段和方法。在这种继承类对象调用某个具体方法时，它传递的是与该方法原有基类相对地址相应的 this 指针。

51.1.5　虚拟方法

我们再来看看一个简单点的例子：

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};
```

```
class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

类 object 定义一个虚拟函数 dump()，它被继承类 box 和 sphere 中的同名函数覆盖了。

在调用虚拟函数时，编译器阶段可能无法确定对象的类型情况。当类中含有虚函数时，其基类的指针就可以指向任何派生类的对象，这时就有可能不知道基类指针到底指向的是哪个对象的情况。这时就要根据实时类型信息，确定应当调用的相应函数。

在启用优化编译选项/Ox 和/Ob0 后，我们再用 MSVC 2008 编译主函数 main():

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub  esp, 32
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
```

```
    lea  ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov  eax, DWORD PTR _b$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _b$[esp+32]
    call edx
    mov  eax, DWORD PTR _s$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _s$[esp+32]
    call edx
    xor  eax, eax
    add  esp, 32
    ret  0
_main ENDP
```

指向 dump()函数的函数指针应当位于类对象 object 中的某个地方。我们在哪里去找新方法的函数地址呢？它必定由构造函数定义： main()函数没用调用其他函数，因此这个指针肯定由构造函数定义。

box 类实例的构造函数为：

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box 'RTTI Type Descriptor'
    DD    00H
    DB    '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@@8 DD FLAT:??_R0?AVbox@@@8 ; box::'RTTI Base Class Descriptor at (0,-1,0,64)'
    DD    01H
    DD    00H
    DD    0ffffffffH
    DD    00H
    DD    040H
    DD    FLAT:??_R3box@@8

??_R2box@@8 DD    FLAT:??_R1A@?0A@EA@box@@8 ; box::'RTTI Base Class Array'
    DD    FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD    00H ; box::'RTTI Class Hierarchy Descriptor'
    DD    00H
    DD    02H
    DD    FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::'RTTI Complete Object Locator'
    DD    00H
    DD    00H
    DD    FLAT:??_R0?AVbox@@@8
    DD    FLAT:??_R3box@@8

??_7box@@6B@ DD    FLAT:??_R4box@@6B@ ; box::`vftable'
    DD    FLAT:?dump@box@@UAEXXZ

_color$ = 8    ; size = 4
_width$ = 12   ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20   ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    call ??0object@@QAE@XZ ; object::object
    mov  eax, DWORD PTR _color$[esp]
    mov  ecx, DWORD PTR _width$[esp]
    mov  edx, DWORD PTR _height$[esp]
    mov  DWORD PTR [esi+4], eax
    mov  eax, DWORD PTR _depth$[esp]
    mov  DWORD PTR [esi+16], eax
    mov  DWORD PTR [esi], OFFSET ??_7box@@6B@
```

```
    mov  DWORD PTR [esi+8], ecx
    mov  DWORD PTR [esi+12], edx
    mov  eax, esi
    pop  esi
    ret  16
??0box@@QAE@HHHH@Z ENDP ; box::box
```

它的内存布局略有不同：第一个字段是某个 box::'vftable'（虚拟函数表）的指针（具体名称由 MSVC 编译器设置）。

在这个表中，我们看到一个指向数据表 box::RTTI Complete Object Locator 的链接和一个指向类成员函数 box::dump()的链接。它们的正规名称分别是虚拟方法表和 RTTI[①]。虚拟方法表存储着各方法的地址，RTTI 表存储着类型的信息。另外，RTTI 表为 C++程序提供了“强制转换运算符”dynamic_cast（将基类类型的指针或引用安全地转换为派生类型的指针或引用）和“类型查询操作符”typeid。在上述指令调用类成员函数时，它所使用的类名称仍然是文本型字符串。基于代码中 dump()函数的实例情况可知，在通过调用指向基类的指针（或引用）调用其虚拟函数（类实例：：虚方法）时，指针最终会指向派生类实例的同名虚拟方法——构造函数会把指针实际指向的对象实例的类型信息存储在数据结构之中。

在内存数据表里检索虚拟方法的内存地址必定要消耗额外的 CPU 时间。因此虚拟方法的运行速度比一般的方法要慢一些。

在 GCC 生成的相应代码中，RTTI 表的构造稍微有些不同。

51.2 ostream 输出流

我们来看一个经典的例子“hello world!”，这里我们试图采用输出流的方式重新实现它。

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

几乎所有的教科书都写明，运算重载符“<<”的作用是“重载”其他类型的数据，主要用于对象之间的运算。我们来看看操作符“<<”的输出流用法。

指令清单 51.18 精简版的 MSVC 2012 的程序代码

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU? ↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

我们把源程序稍微修改一下：

```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

① RTTI 是 Run-time type information 的缩写，意思是实时类型信息。

教科书都说这种运算会从左至右依次进行。实际上确实如此：

指令清单 51.19　MSVC 2012 下的程序实现

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU? ↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax             ; result of previous function execution
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU? ↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8

    xor  eax, eax
    ret  0
_main ENDP
```

如果我们用 f()函数来表示“<<”的运算功能的话，那么上述程序就可以表示为：

```
f(f(std::cout, "Hello, "), "world!");
```

GCC 生成的代码 和 MSVC 生成的代码基本一样。

51.3　引用

C++中，引用也是指针（可以参考本书第 10 章），但是调用其是安全的，因为在编译它们时，是很难出现失误的（参见 ISO13，p.8.3.2）。比如说，引用永远指向某个类型的既定对象，而且引用不可以是空（参见 Cli,p.8.6）。更进一步讲，引用不能改变指向，不可能把某个引用重新指向其他的对象（参见 Cli,p.8.5）。

接下来，我们稍微改写一下第 10 章的第一个源程序，把 f1 函数的指针全部替换为引用：

```
void f2 (int x, int y, int & sum, int & product)
{
        sum=x+y;
        product=x*y;
};
```

我们可以看到编译之后的代码和那个使用指针的代码完全一样（参见第 10 章）。

指令清单 51.20　采用 MSVC 2010 优化的程序代码

```
_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
_sum$ = 16                                              ; size = 4
_product$ = 20                                          ; size = 4
?f2@@YAXHHAAH0@Z PROC                                   ; f2
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul eax, ecx
        mov ecx, DWORD PTR _product$[esp-4]
        push esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
```

```
        ret     0
?f2@@YAXHHAAH0@Z ENDP                                   ; f2
```

至于为什么这里的程序使用了一些奇怪的名字，可以参见本书的 51.1.1 节。

51.4 STL/标准模板库（Standard Template Library）

请注意：本书仅保证本节内容在 32 位系统里有效，未对 x64 系统做过验证。

51.4.1 std::string（字符串）

内部

多数字符串库（Yur13,p.2.2）都把 std::string 定义为以下几个组件：缓冲区指针、字符串缓冲区、当前字符串的长度（便于函数操作；请参阅［Yur13］的 2.2.1 节）以及当前缓冲区的容量。在缓冲区里，字符串通常用 0 作字符串结束符，以兼容常规的 C 语言 ASCIIZ 字符串格式。

C++的标准 ISO13 没有定义 std::string 的实现方法。然而，它们通常用上述方法定义的这种数据结构。

在 C++的标准数据结构中，字符串不是类对象（不像 Qt 那样，把 QString 声明为标准类）。它把字符串定义为模板型数据（基本字符串 basic_string）。这是为了兼容各种类型的字符元素。现在，它至少支持 char（标准字符）以及 wchar_t（宽字符）。

因此，std::string 是以 8 位字符 char 为基本类型的类，而 std::wstring 则是以 16/32 位宽字符 wchar_t 为基本类型的类。

MSVC

当字符串长度在 16 字符以内时，MSVC 将字符串数据直接存储在缓冲区里，不再使用“指针＋缓冲区”的复杂结构。

这就意味着：在 32 位系统里，字符串最少会占用 24（16+4+4=24）个字节；而在 64 位环境下，字符串至少占用 32（16+8+8=32）个字节。如果字符串的长度超过 16 个字符的话，这 16 个字节空间就算白白浪费了。

指令清单 51.21 MSVC 的例子程序

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
    size_t size;      // AKA 'Mysize' in MSVC
    size_t capacity;  // AKA 'Myres' in MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, p->size, p-> capacity);
};

int main()
{
```

```
    std::string s1="short string";
    std::string s2="string longer that 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // that works without using c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};
```

源代码的功能应当不需要解释。

需要注意的有以下几点：

只要字符串长度没有超过 16 个字符，编译器就不会使用堆（heap）存储字符串的缓冲区。在实际的程序中，多数字符串确实是短字符串。很明显，微软研发人员将 16 字符串作为长短字符串的分割点。

虽然我们没有在主程序 main()的最后调用成员函数 c_str()，但是这个程序可以通过编译，也能在屏幕上显示出字符串的内容。

这就是为什么这个程序能运转起来。

第一个例子的字符串不足 16 个字符，因此它会被保存到字符串缓冲区。这个缓冲区实际位于 std:string 对象的起始地址（可视为结构体型数据）。这个区域里的数据，采取了标准的 ASCIIZ 的数据结构。因此 printf()函数在处理指针时直接处理了相应的 ASCIIZ 字符串。所以，源程序可以显示第一个字符串的内容。

第二个字符串的长度大于 16 字节，更危险。编程人员很容易在此疏忽大意、忘记此时应当使用 string 对象的成员函数 c_str()。之所以本例这样还能正常工作，是因为指向缓冲区的指针正好位于结构体的开始部分。某些人可能在相当长的时间里一直这么写代码，而一直不觉得会有问题；直到他们遇到超长字符串引发程序崩溃的时候，他们才会开始意识到问题的存在。

GCC

GCC 在实现 std::string 的时候使用了 MSVC 里没有的变量——reference count。

另外一个有趣的事情是：指向 std::string 实例的指针，并不是指向结构体的起始地址，而是指向了字符串缓冲区的指针。文件 libstdc++-v3/include/bits/ basic_string.h 解释了这一问题。它说，这是为了便于调试程序：

```
* The reason you want _M_data pointing to the character %array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to get
* the _Rep for the debugger to use, so users can check the actual
* string length.)
```

- 之所以使用_M_data（而不采用_Rep）指向字符串数列%array，原因在于调试人员能看到字符串的内容。也许我们可以增加一个非内驻的成员_Rep 以调试，但是使用者可以检查实际的字符串长度。

有兴趣的读者可以下载看看：http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01068.html。

考虑到有关问题之后，我们调整了上一个小节使用的源程序：

指令清单 51.22　GCC 例子

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
```

```
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string));
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer that 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // GCC type checking workaround:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

因为 GCC 的类型检查规则更为苛刻，所以我们不得不做很多针对性的修改。即使这个程序的 printf() 函数同样可以在不依赖 c_str()函数的情况下打印字符串内容。

一个更加复杂的例子

```
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

指令清单 51.23 MSVC 2012 编译的程序

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub  esp, 72

    push 7
    push OFFSET $SG39512
    lea  ecx, DWORD PTR _s1$[esp+80]
    mov  DWORD PTR _s1$[esp+100], 15
    mov  DWORD PTR _s1$[esp+96], 0
    mov  BYTE PTR _s1$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; ↙
    ↳std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    push 7
    push OFFSET $SG39514
    lea  ecx, DWORD PTR _s2$[esp+80]
    mov  DWORD PTR _s2$[esp+100], 15
    mov  DWORD PTR _s2$[esp+96], 0
```

```
    mov  BYTE PTR _s2$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; ↙
    ↳ std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    lea  eax, DWORD PTR _s2$[esp+72]
    push eax
    lea  eax, DWORD PTR _s1$[esp+76]
    push eax
    lea  eax, DWORD PTR _s3$[esp+80]
    push eax
    call ??$?HDU?$char_traits@D@std@@V?$allocator@D@1@@std@@YA?AV?$basic_string@DU? ↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@0@ABV10@0@Z ; std::operator+<char,std::char_traits< ↙
    ↳ char>,std::allocator<char> >

    ; inlined c_str() method:
    cmp  DWORD PTR _s3$[esp+104], 16
    lea  eax, DWORD PTR _s3$[esp+84]
    cmovae eax, DWORD PTR _s3$[esp+84]

    push eax
    push OFFSET $SG39581
    call _printf
    add  esp, 20

    cmp  DWORD PTR _s3$[esp+92], 16
    jb   SHORT $LN119@main
    push DWORD PTR _s3$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN119@main:
    cmp  DWORD PTR _s2$[esp+92], 16
    mov  DWORD PTR _s3$[esp+92], 15
    mov  DWORD PTR _s3$[esp+88], 0
    mov  BYTE PTR _s3$[esp+72], 0
    jb   SHORT $LN151@main
    push DWORD PTR _s2$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN151@main:
    cmp  DWORD PTR _s1$[esp+92], 16
    mov  DWORD PTR _s2$[esp+92], 15
    mov  DWORD PTR _s2$[esp+88], 0
    mov  BYTE PTR _s2$[esp+72], 0
    jb   SHORT $LN195@main
    push DWORD PTR _s1$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN195@main:
    xor  eax, eax
    add  esp, 72
    ret  0
_main ENDP
```

编译器没有采用原有模式构建字符串。使用堆来存储字符串缓冲区，数据结构自然就完全不同。ASCIIZ 字符串通常存储于程序的数据段，在执行程序的时候，assign 方式会构造字符串 s1 和 s2。而后程序通过运算符“＋”构造 s3。

请注意此处没有调用字符串的 c_str()方式。因为代码很紧凑，所以编译器把 c_str()的代码内联（内嵌）到了此处：如果字符串的长度不足 16 个字符，那么 EAX 寄存器将保留缓冲区的指针；否则它将提取堆里那个存储字符串的缓冲区指针。

最后，程序调用了 3 个析构函数，用于释放长字符串（长度大于 16 个字符的字符串）占用的堆空间。如果字符串长度小于 16，那么 std::string 对象全部存储于数据栈，会随函数结束而自动释放。

就性能而言，短字符串的处理速度更快，因为访问堆的操作要少一些。

GCC 的处理方式更为简单。GCC 不会把短字符串的文本缓冲区直接存储到 string 结构体里。

使用 GCC\ 4.8.1 编译上述程序，可得到：

指令清单 51.24 GCC 4.8.1 下的程序编译

```
.LC0:
    .string "Hello, "
.LC1:
    .string "world!\n"
main:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea  ebx, [esp+28]
    lea  edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea  esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

    ; inlined c_str():
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts

    mov  eax, DWORD PTR [esp+28]
    lea  ebx, [esp+19]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+24]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+20]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    lea  esp, [ebp-12]
```

```
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp
    ret
```

可见，传递给析构函数的指针不是对象的指针，而是指向 string 对象之前 12 个字节的地址的指针——那才是结构体真正的起始地址。

全局变量 std::string

虽然资深的 C++编程人员都不会把 std::string 当作全局变量使用，但实际上由 STL 定义的数据类型都可以用作全局变量，

确实如此，我们来看下面这段程序：

```
#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

实际上，在主函数 main()启动之前，全局变量就已经完成初始化操作了。

指令清单 51.25　MSVC 2012（这里我们可以看到这个全局变量是如何构造的以及一个析构体是如何注册的）

```
??__Es@@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov  ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; ↙
    ↳ std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??__Fs@@YAXXZ ; `dynamic atexit destructor for 's''
    call _atexit
    pop  ecx
    ret  0
??__Es@@YAXXZ ENDP
```

指令清单 51.26　MSVC 2012（从这个程序我们可以看到，主函数 main()是如何使用一个全局变量的）

```
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    mov  eax, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

指令清单 51.27　MSVC 2012 在退出之前，析构函数的调用过程

```
??__Fs@@YAXXZ PROC
    push ecx
```

```
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    jb   SHORT $LN23@dynamic
    push esi
    mov  esi, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    lea  ecx, DWORD PTR $T2[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    lea  ecx, DWORD PTR $T2[esp+12]
    call ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAEXPAPAD@Z
    lea  ecx, DWORD PTR $T1[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push esi
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    pop  esi
$LN23@dynamic:
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 15
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+16, 0
    mov  BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A, 0
    pop  ecx
    ret  0
??__Fs@@YAXXZ ENDP
```

实际上，主函数 main()没有调用那个创建全局变量的构造函数，这部分任务是 CRT 在启动 main()函数之前就已经完成的操作。不止如此，全局变量的析构函数由 stdlib 声明的 atexit()函数提供，只有在 main 结束之后才会被调用。

GCC 的处理方法十分相似。经 GCC 4.8.1 编译上述源程序，可得到：

指令清单 51.28 GCC 4.8.1 函数

```
main:
    push ebp
    mov  ebp, esp
    and  esp, -16
    sub  esp, 16
    mov  eax, DWORD PTR s
    mov  DWORD PTR [esp], eax
    call puts
    xor  eax, eax
    leave
    ret
.LC0:
    .string "a string"
_GLOBAL__sub_I_s:
    sub  esp, 44
    lea  eax, [esp+31]
    mov  DWORD PTR [esp+8], eax
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], OFFSET FLAT:s
    call _ZNSsC1EPKcRKSaIcE
    mov  DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
    mov  DWORD PTR [esp+4], OFFSET FLAT:s
    mov  DWORD PTR [esp], OFFSET FLAT:_ZNSsD1Ev
    call __cxa_atexit
    add  esp, 44
    ret
.LFE645:
    .size  _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long  _GLOBAL__sub_I_s
    .globl s
    .bss
    .align 4
```

```
    .type  s, @object
    .size  s, 4
s:
    .zero 4
    .hidden __dso_handle
```

但是 GCC 没有单独建立一个专用函数，而是把每个析构函数逐一传递给 atexit()函数。

51.4.2　std::list

std::list 是众所皆知的双向链表的容器类，它的每个数据元素可通过链表指针（链域）串接成逻辑意义上的线性表。这种数据结构的每个元素都有 2 个指针，一个指针指向前一个元素，另一个指针指向后一个元素。

链域的存在决定了，每个节点要比单纯的节点数据元素多占用 2 个 words 的空间，即 32 位系统下要多占用 8 个字节，而 64 位系统下会多占用 16 个字节。

C++的标准模板库只是给现有结构体扩充了“next”“previous”指针，使之形成语意上的有序序列。

我们以 2 个变量组成的结构体为例，演示 std::list 的链结构。

虽然 C++标准（ISO/IEC 14882:2011 (C++ 11 standard)）没有明确这种数据结构的具体实现方法，但是 MSVC 编译器和 GCC 编译器不约而同地选择几乎一致的实现方法。我们就以下源程序为例进行说明：

```
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
        n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // GCC implementation does not have "size" field
```

```
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
    printf ("node at .end:\n");
    tmp=l.end();
    dump_List_node ((struct List_node *)*(void**)&tmp);

    printf ("* let's count from the begin:\n");
    std::list<struct a>::iterator it=l.begin();
    printf ("1st element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("2nd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("3rd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

    printf ("* let's count from the end:\n");
    std::list<struct a>::iterator it2=l.end();
    printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

    printf ("removing last element...\n");
    l.pop_back();
    dump_List_val((unsigned int*)(void*)&l);
};
```

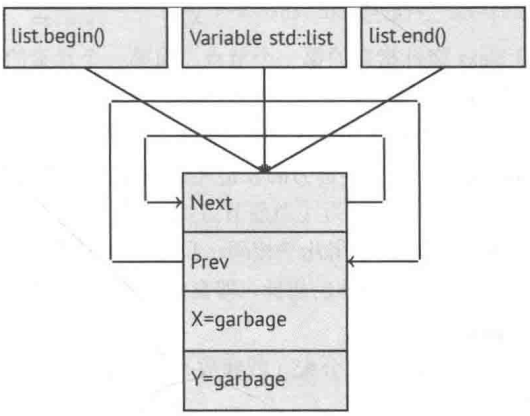
GCC

我们首先讲解 GCC 编译器的编译方式。

运行上述程序时，将会得到很多输出数据。我们进行分段解说：

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

此时它还是个空链。虽然我们还未对其进行赋值操作，但是变量 x 和变量 y 已经有数据了（也就是常人所说的虚结点/dummy node）。而且，“next”“prev”指针都指向该节点自己。



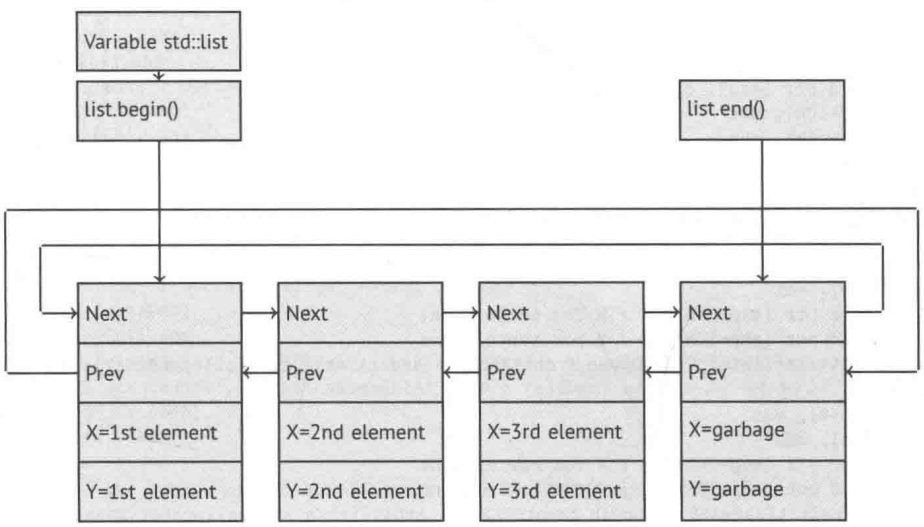
在这个时候，迭代器.begin（开始）和.end（结束）值相等。

然后程序创建了 3 个节点，此后整个链表的内部数据将会变为：

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

最后一个元素的地址还是刚才的 0x0028fe90。实际上在释放结构体之前它的地址不会发生变化。而且这个节点的两个字段 *x* 和 *y* 仍然还是随机的噪音数据，这时它们的值分别是 5 和 6。碰巧的是，这些值和最后一个元素的值相同。只是这种巧合并不会一直持续下去。

此时，这 3 个节点存储过程和内存分布结构如下所示。



变量 1 总是指向第一个节点。

实际上迭代器.begin()（开始）和.end()（结束）不是变量而是函数，它们的返回值是相应节点的指针。

双向链表通常都会采用这种虚节点（英文别称是 sentinel node/岗哨节点）的实现方法。这种节点具有

简化链结构和提升操作效率的作用。

迭代器其实就是一个指向节点的指针，而 list.begin()和 list.end()分别返回首/尾节点的指针。

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

事实上，最后一个节点的 Next 指针指向的第一个节点，而第一个元素的 Prev 指针指向的是最后一个元素。因此我们很容易就能想到这是一个循环链。

所以，可利用指向第一个节点的指针（即本例的实例 1）顺藤摸瓜地找到序列的最后一个节点，而不必遍历整个链。利用这种结构，我们还可毫不费力的在链尾之后插入其他节点。

运算符“++”、“--”就是把迭代器的值设为［当前节点->prev］或［当前节点->next］的值。逆向迭代器（reverse iterators，即.rbegin 和.rend）的作用几乎相同，只是方向相反。

迭代器的运算符“*”用于返回节点结构体的指针，即自定义的数据结构体的启始地址，换句话说是节点第一项数据（本例中是 x）的指针。

在链表里插入和删除节点时，我们只需要分配（或释放）节点的存储空间，,然后再更新所有的链域指针、即可保障整个链的有效性。

在删除节点之后，相邻节点的迭代器（链域）可能依然指向被删除的无效节点，此时迭代器就会失效。指向无效结点的迭代器又叫做“迷途指针”和“悬空指针”（dangling pointer）。当然，这种指针就不能再使用了。

在 GCC（以 4.8.1 版为例）编译 std::list 的实例时，它不会存储链中的节点总数。这就意味着内置函数.size()的运行速度十分缓慢，因为它必须遍历完整个数据链才能返回节点个数。因此，那些与节点总数有关的所有函数（例如 O(n)）的时间开销都和链表长度成正比。

指令清单 51.29 GCC4.8.1 带参数-fno 内置小函数的优化

```
main proc near
    push ebp
    mov  ebp, esp
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  ebx, [esp+10h]
    mov  dword ptr [esp], offset s ; "* empty list:"
    mov  [esp+10h], ebx
    mov  [esp+14h], ebx
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea  esi, [esp+18h]
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 1 ; X for new element
    mov  dword ptr [esp+1Ch], 2 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a ↙
    ↳ const&)
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 3 ; X for new element
    mov  dword ptr [esp+1Ch], 4 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a ↙
    ↳ const&)
    mov  dword ptr [esp], 10h
    mov  dword ptr [esp+18h], 5 ; X for new element
    mov  dword ptr [esp+1Ch], 6 ; Y for new element
    call _Znwj ; operator new(uint)
```

```
    cmp  eax, 0FFFFFFF8h
    jz   short loc_80002A6
    mov  ecx, [esp+1Ch]
    mov  edx, [esp+18h]
    mov  [eax+0Ch], ecx
    mov  [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
    mov  [esp+4], ebx
    mov  [esp], eax
    call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::__detail::_List_node_base::_M_hook ↙
    ↳ (std::__detail::_List_node_base*)
    mov  dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
    call puts
    mov  eax, [esp+10h]
    mov  [esp], eax
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
    mov  dword ptr [esp], offset aNodeAt_end ; "node at .end:"
    call puts
    mov  [esp], ebx
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
    mov  dword ptr [esp], offset aLetSCountFromT ; "* let's count from the begin:"
    call puts
    mov  esi, [esp+10h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi] ; operator++: get ->next pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the end:" ↙
    ↳ call puts
    mov  eax, [esp+1Ch]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+0Ch], eax
    mov  eax, [esp+18h]
```

```
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esp+14h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi+4] ; operator--: get ->prev pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi+4] ; operator--: get ->prev pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
    call puts
    mov  esi, [esp+14h]
    mov  [esp], esi
    call _ZNSt8__detail15_List_node_base9_M_unhookEv   ; std::__detail::_List_node_base:: ↙
    ↳ _M_unhook(void)
    mov  [esp], esi ; void *
    call _ZdlPv ; operator delete(void *)
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  [esp], ebx
    call _ZNSt10_List_baseIiaSaIiS0_EE8_M_clearEv ; std::_List_base<a,std::allocator<a>>:: ↙
    ↳ _M_clear(void)
    lea  esp, [ebp-8]
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  ebp
    retn
main endp
```

运行上面这个由 GCC 编译的程序，可以得到如下数据：

指令清单 51.30 整个输出

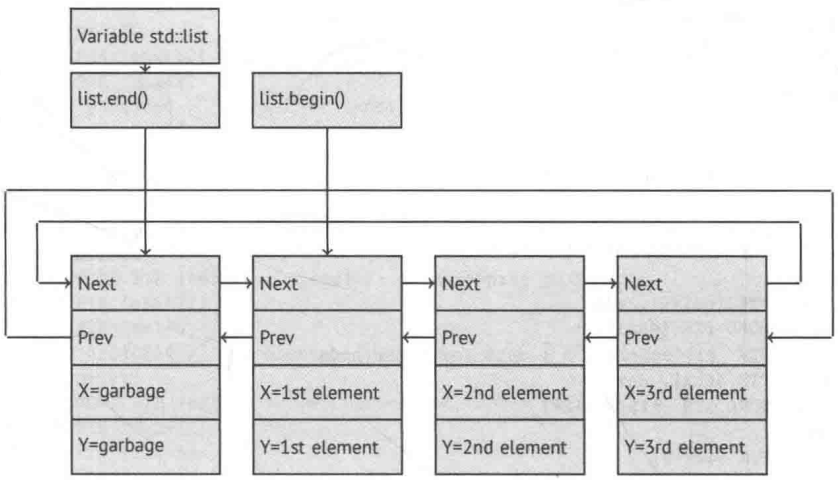
```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
```

```
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

MSVC

在编译 std::list 的时候，MSVC 2012 会存储链表的长度。除此以外，它的实现方法和 GCC 基本一致。这就是说，内置函数.size()只需要从内存中读取一个值就可以返回函数结果，其速度相当快。不过，这也意味着每次添增/删除节点的时候都需要调整这个值。

MSVC 的链存储结构也和 GCC 有所区别：



可见，GCC 构造的虚节点在链尾，而 MSVC 构造的虚节点在链首。

使用 MSVC 2012（启用/Fa2.asm/GS-/Ob1 选项）编译上述程序，可得到：

指令清单 51.31 MSVC 2012 带参数/Fa2.asm/GS-/Ob1 优化的程序

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main    PROC
    sub  esp, 16
    push ebx
    push esi
    push edi
    push 0
    push 0
    lea  ecx, DWORD PTR _l$[esp+36]
    mov  DWORD PTR _l$[esp+40], 0
    ; allocate first "garbage" element
    call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V? ↙
    ↳ $allocator@Ua@@@std@@@std@@@std@@QAEPAU?$_List_node@Ua@@PAX@2@PAU32@0@Z ; std:: ↙
    ↳ _List_alloc<0,std::_List_base_types<a,std::allocator<a> > >::_Buynode0
    mov  edi, DWORD PTR __imp__printf
    mov  ebx, eax
    push OFFSET $SG40685 ; '* empty list:'
    mov  DWORD PTR _l$[esp+32], ebx
    call edi ; printf
```

```
lea  eax, DWORD PTR _1$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov  esi, DWORD PTR [ebx]
add  esp, 8
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _1$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+40], 1 ; data for a new node
mov  DWORD PTR _t1$[esp+44], 2 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU? ↙
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
↳ const &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 3 ; data for a new node
mov  DWORD PTR [ecx], eax
mov  esi, DWORD PTR [ebx]
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _1$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+44], 4 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU? ↙
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
↳ const &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 5 ; data for a new node
mov  DWORD PTR [ecx], eax
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea  ecx, DWORD PTR _1$[esp+36]
push ebx
mov  DWORD PTR _t1$[esp+44], 6 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU? ↙
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
↳ const &>
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _1$[esp+36], 3
mov  DWORD PTR [ecx], eax
call edi ; printf
lea  eax, DWORD PTR _1$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx] ; get next field of node I variable points to
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call edi ; printf
push ebx ; pointer to the node $1$ variable points to!
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let''s count from the begin:'
call edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
```

```
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40846 ; '1st element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40848 ; '2nd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40850 ; '3rd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
    add  esp, 64
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40852 ; 'element at .end(): %d %d'
    call edi ; printf
    push OFFSET $SG40853 ; '* let''s count from the end:'
    call edi ; printf
    push DWORD PTR [ebx+12] ; use x and y fields from the node l variable points to
    push DWORD PTR [ebx+8]
    push OFFSET $SG40860 ; 'element at .end(): %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40862 ; '3rd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40864 ; '2nd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40866 ; '1st element: %d %d'
    call edi ; printf
    add  esp, 64
    push OFFSET $SG40867 ; 'removing last element...'
    call edi ; printf
    mov  edx, DWORD PTR [ebx+4]
    add  esp, 4

    ; prev=next?
    ; it is the only element, "garbage one"?
    ; if yes, do not delete it!
    cmp  edx, ebx
    je   SHORT $LN349@main
    mov  ecx, DWORD PTR [edx+4]
    mov  eax, DWORD PTR [edx]
    mov  DWORD PTR [ecx], eax
    mov  ecx, DWORD PTR [edx]
    mov  eax, DWORD PTR [edx+4]
    push edx
    mov  DWORD PTR [ecx+4], eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  DWORD PTR _l$[esp+32], 2
$LN349@main:
    lea  eax, DWORD PTR _l$[esp+28]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  eax, DWORD PTR [ebx]
```

```
    add  esp, 4
    mov  DWORD PTR [ebx], ebx
    mov  DWORD PTR [ebx+4], ebx
    cmp  eax, ebx
    je   SHORT $LN412@main
$LL414@main:
    mov  esi, DWORD PTR [eax]
    push eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  eax, esi
    cmp  esi, ebx
    jne  SHORT $LL414@main
$LN412@main:
    push ebx
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    xor  eax, eax
    pop  edi
    pop  esi
    pop  ebx
    add  esp, 16
    ret  0
_main ENDP
```

在构造数据链的时候，MSVC 通过“Buynode”函数分配了虚节点的空间；此后，这个函数同样用于分配其他节点的存储空间。相比之下，GCC 则会在局部栈里存储链表的第一个节点。

指令清单 51.32 整个程序的输出

```
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list 单向链表

std::forward_list 和 std::list 的结构基本相同，只是它是单向链，它的迭代器（链域）只有”next”字段而没有”prev”字段。虽然这种链表的内存开销变小了，但是无法进行逆向遍历。

51.4.3　std::vector 标准向量

我们将 std::vector 标准向量称为 PODT[1]C 数组的安全封装容器。其内部结构和标准字符串 std::string 十分相似（参见 51.4.1 节）。它有一个数据缓冲区的专用指针，一个指向数组尾部的专用指针，以及一个指向分配缓冲区尾部的专用指针。

这种数组采取的各个元素以彼此相邻的方式存储于内存之中，这点和常规数组没什么区别（参见第 18 章）。新推出的 C++11 标准，为其定义了内置函数.data()。std::vector 的.data()的作用就和 std::string 中的.c_str()一样，用于返回缓冲区的地址。

这种数据结构使用堆/heap 来存储数据缓冲区，而堆的空间消耗可能会比数组本身还大。

MSVC 和 GCC 的实现机理基本相同，只是结构体的变量名称稍微有些不同。因此，所以这里使用同一个例子进行说明。下述源程序用于遍历 std::vector 的数据结构。

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // GCC structure is the same, but names are: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast, in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};

int main()
{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // with bounds checking
    printf ("%d\n", c[8]); // operator[], without bounds checking
};
```

① PODT 是 Plain Old Data Type，纯文本的老的数据类型。

下面是采用 MSVC 编译后的程序输出。

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

从程序中我们可以看到，当主函数 main()开始运行后并没有立即分配数据缓冲区。在第一次调用 push_back()函数之后，它才分配了缓冲区。此后，每调用一次 push_back()函数，数组的空间和 size(capacity)都增大一次。不仅空间容量逐渐增长，而且缓冲区的地址也会同期改变。这是因为每次调用 push_back()函数时，都会在堆里重新分配空间。所以这种操作的时间开销很大。要想提高效率，就必须预测数组的大小并使用.reserve()方式为其预留存储空间。

最后一个数值是随机的噪音。它不属于链表的某个节点，只是一个随机数。从这一点我们可以看出，std::vector（标准向量）的下标运算符“[]”不会检测索引值是否超越下标界限。要进行这种数组边界检查，可以使用内置函数.at()。虽然.at()方法的运行速度较慢，但是它能在下标越 std::out_of_range（越界）的错误提示。

我们来看它的汇编指令。使用 MSVC 2012（启用 /GS- /Ob1 选项）编译上述源程序，可得到：

指令清单 51.33 MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
```

```
__Pos$ = 8  ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::vector<int,std::allocator<int> ↙
    ↳ >::at, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _this$[ebp]
    mov  edx, DWORD PTR [eax+4]
    sub  edx, DWORD PTR [ecx]
    sar  edx, 2
    cmp  edx, DWORD PTR __Pos$[ebp]
    ja   SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPPO@invalid?5vector?$DMT?$DO?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z
$LN1@at:
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax]
    mov  edx, DWORD PTR __Pos$[ebp]
    lea  eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov  esp, ebp
    pop  ebp
    ret  4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int,std::allocator<int> ↙
    ↳ >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8  ; size = 4
$T6 = -4  ; size = 4
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 36
    mov  DWORD PTR _c$[ebp], 0 ; Myfirst
    mov  DWORD PTR _c$[ebp+4], 0 ; Mylast
    mov  DWORD PTR _c$[ebp+8], 0 ; Myend
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T6[ebp], 1
    lea  ecx, DWORD PTR $T6[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
    ↳ allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T5[ebp], 2
    lea  eax, DWORD PTR $T5[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
    ↳ allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
```

```
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T4[ebp], 3
lea  edx, DWORD PTR $T4[ebp]
push edx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
↳ allocator<int> >::push_back
lea  eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T3[ebp], 4
lea  ecx, DWORD PTR $T3[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
↳ allocator<int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
push 6
lea  ecx, DWORD PTR _c$[ebp]
call ?reserve@?$vector@HV?$allocator@H@std@@@std@@QAEXI@Z ; std::vector<int,std::allocator< ↙
↳ int> >::reserve
lea  eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T2[ebp], 5
lea  ecx, DWORD PTR $T2[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
↳ allocator<int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T1[ebp], 6
lea  eax, DWORD PTR $T1[ebp]
push eax
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std:: ↙
↳ allocator<int> >::push_back
lea  ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
push 5
lea  ecx, DWORD PTR _c$[ebp]
call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ; std::vector<int,std::allocator<int ↙
↳ > >::at
mov  edx, DWORD PTR [eax]
push edx
push OFFSET $SG52650 ; '%d'
call DWORD PTR __imp__printf
add  esp, 8
mov  eax, 8
shl  eax, 2
mov  ecx, DWORD PTR _c$[ebp]
mov  edx, DWORD PTR [ecx+eax]
push edx
push OFFSET $SG52651 ; '%d'
```

```
    call DWORD PTR __imp__printf
    add  esp, 8
    lea  ecx, DWORD PTR _c$[ebp]
    call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ   ; std::vector<int,std::allocator<int ↙
    ↳ > >::_Tidy
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

从中我们可以看到.at()方法检查下标边界以及异常处理的详细细节。最后一个输出值是 printf()函数直接从内存中提取的数值。它在提取数值的时候没有做任何越界检查。

有读者可能会问了，为什么标准向量的数据结构为什么不像标准函数 std::string 中的那样、单独用一个字段记录 size（大小）和 capacity（体积）这类的信息呢？虽然笔者无法查证，但是笔者相信大概是.at()在进行边界检查时效率更好吧。

GCC 生成的汇编指令也十分雷同，不过它以内联函数的形式实现.at()。

指令清单 51.34　GCC 4.8.1 –fno –inline –small –functions –O1 编译

```
main proc near
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    mov  dword ptr [esp+14h], 0
    mov  dword ptr [esp+18h], 0
    mov  dword ptr [esp+1Ch], 0
    lea  eax, [esp+14h]
    mov  [esp], eax
    call  _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 1
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call  _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
    ↳ int const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 2
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
    ↳ int const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 3
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
    ↳ int const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
```

```
        call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
        mov  dword ptr [esp+10h], 4
        lea  eax, [esp+10h]
        mov  [esp+4], eax
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _ZNSt6vectorIiSaIiEE9push_backERKi   ; std::vector<int,std::allocator<int>>::push_back( ↙
       ↳ int const&)
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
        mov  ebx, [esp+14h]
        mov  eax, [esp+1Ch]
        sub  eax, ebx
        cmp  eax, 17h
        ja   short loc_80001CF
        mov  edi, [esp+18h]
        sub  edi, ebx
        sar  edi, 2
        mov  dword ptr [esp], 18h
        call _Znwj            ; operator new(uint)
        mov  esi, eax
        test edi, edi
        jz   short loc_80001AD
        lea  eax, ds:0[edi*4]
        mov  [esp+8], eax     ; n
        mov  [esp+4], ebx     ; src
        mov  [esp], esi       ; dest
        call memmove

loc_80001AD: ; CODE XREF: main+F8
        mov  eax, [esp+14h]
        test eax, eax
        jz   short loc_80001BD
        mov  [esp], eax       ; void *
        call _ZdlPv           ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
        mov  [esp+14h], esi
        lea  eax, [esi+edi*4]
        mov  [esp+18h], eax
        add  esi, 18h
        mov  [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
        mov  dword ptr [esp+10h], 5
        lea  eax, [esp+10h]
        mov  [esp+4], eax
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
       ↳ int const&)
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
        mov  dword ptr [esp+10h], 6
        lea  eax, [esp+10h]
        mov  [esp+4], eax
        lea  eax, [esp+14h]
        mov  [esp], eax
        call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
       ↳ int const&)
        lea  eax, [esp+14h]
```

```
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  eax, [esp+14h]
    mov  edx, [esp+18h]
    sub  edx, eax
    cmp  edx, 17h
    ja   short loc_8000246
    mov  dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
    call _ZSt20__throw_out_of_rangePKc ; std::__throw_out_of_range(char const*)

loc_8000246:                            ; CODE XREF: main+19C
    mov  eax, [eax+14h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    mov  eax, [eax+20h]
    mov  [esp+8], eax
    mov  dword ptr [esp+4], offset aD ; "%d\n"
    mov  dword ptr [esp], 1
    call __printf_chk
    mov  eax, [esp+14h]
    test eax, eax
    jz   short loc_80002AC
    mov  [esp], eax      ; void *
    call _ZdlPv          ; operator delete(void *)
    jmp  short loc_80002AC

    mov  ebx, eax
    mov  edx, [esp+14h]
    test edx, edx
    jz   short loc_80002A4
    mov  [esp], edx      ; void *
    call _ZdlPv          ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
    mov  [esp], ebx
    call _Unwind_Resume


loc_80002AC: ; CODE XREF: main+1EA
             ; main+1F4
    mov  eax, 0
    lea  esp, [ebp-0Ch]
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
                ; .eh_frame:080005BC
    retn
main endp
```

编译器也对.reserve()函数进行了内联处理。如果缓冲区不大，程序会调用 new()方法分配缓冲区的存储空间；使用 memmove()方法来复制缓冲区中的内容；最后通过 delete()方法来释放缓冲区空间。

如果采用 GCC 编译的话，编译程序的输出为：

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
```

```
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0
```

从程序我们可以看到，GCC 的缓冲区 buffer 的增长方式与 MSVC 不同。具体来说，通过一个简单的例子实践可以看到：由 MSVC 编译的程序每次会把缓冲区增加 50%；而由 GCC 编译的程序则会将缓冲区扩张 100%——也就是翻倍。

51.4.4 std::map()和 std::set()

“二叉树”是另外一种常见的基础数据结构。二叉树是每个节点最多有两个子树的有序树，每个节点都有自己的关键字（key）-值（value）。

二叉树通常会用于构造联合数组（associative arrays）。联合数组又称为词典（dictionary）或 hash 表，它由一系列的“关键字-值”构成。

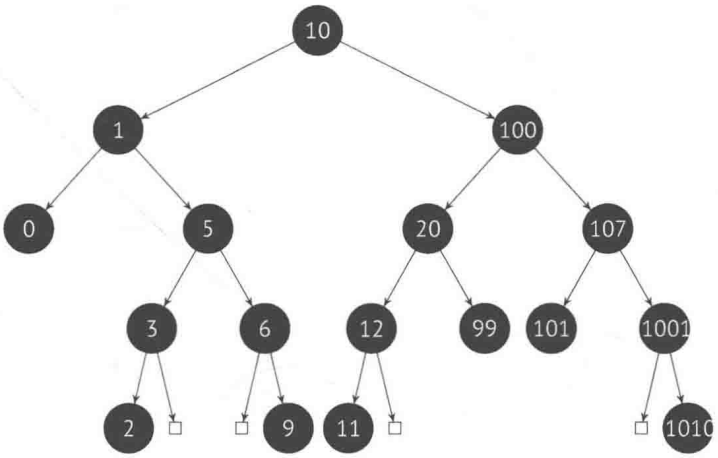
二叉制树最少具备三个重要的特性：

- 所有的关键字都以有序方式存储。
- 任何类型的关键字都能构成二叉树。二叉树的算法与关键字的数据类型无关。只要能够定义一个比较二叉树关键字的排序函数，就可以构造一个二叉树。
- 与链表和数组相比，在二叉树中搜索特定关键字的速度比较快。

现在开始举例说明：我们将存储数列 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010，并形成下述二叉树结构：

从下面的图中，我们很容易发现这样一个规律：所有左边节点的值都比其右边节点的值小，而所有右边节点的值都比左边的节点的值大。

在遵守以上规则的前提下，我们做起查询算法来就相对简单了。我们只需要关注当前节点的数值以及我们要查询的值，将这两者进行比较，当前者比后者小时，那么我们只需要向右搜索对比查找；反之，当前者比后者大时，我们就会向左搜索对比查找。一直到找到节点的值与要搜索的值相等为止。这就是为什么刚才说只需要一个对比函数即可进行节点数或者字符串对比。



所有的键都有一个唯一的值，不能重复。

我们可以计算需要花费的步数，公式大约为：$\log_2 n$（n 是二进制树中键的个数），这个公式计算的是我们要找到一个平衡树的搜索步数。这就意味着在一个 1000 个键的二进制树中，通过大约 10 步我们就能找到需要的值；而 10000 个键的二进制树中则需要 13 步。看起来还是很不错的。树总是需要像这样平衡一下，也就是说，键必须在所有水平上平衡分布。插入和移除一些节点可能都需要保持树的平衡状态。

有几个可用的流行平衡算法，包括 AVL 树和红黑树。后面的算法采用的办法是将每个节点标注为 red（红）或者 black（黑），以简化结点的平衡过程。

不管是 GCC 还是 MSVC 的模板函数 std::map()和 std::set()都使用了红黑树（red-black）的算法。

在 std::set()中只含有键，而 std::map()中除了键（std::set）还有相应的每个节点的数值。

MSVC

```
#include <map>
#include <set>
#include <string>
#include <iostream>

// structure is not packed!
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
```

```
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d Isnil=%d\n",
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnil);
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->second);
    }

    if (traverse)
    {
        if (n->Isnil==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnil==0)
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnil==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
};
const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnil==0)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
```

```
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
};
```

指令清单 51.35 MSVC 2012

```
dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 Isnil=0
```

```
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 Isnil=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 Isnil=0
first=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
```

因为结构设计得并不紧密，所以 2 个 char 型值各占用了 4 字节空间。

对于 std::map 结构来说，first 和 second 的数据数据组合可被视为 std::pair 的一个数据元素。而 std::set 的每个节点则只有 1 个值。

51.4.2 节介绍过，MSVC 编译器在构造 std::list 的时候会保链表的长度信息。我们可以在程序看到它也存储了这种数据类型的具体尺寸。

这两种数据结构和 std::list 的相同之处是，迭代器都是指向节点的指针。.begin()迭代函数指向最小的键。实际上，整个数据结构里都没有最小键的指针（和链表的情况一样）每当程序调用这个迭代器的时候，它就遍历所有键、查找出最小值。单目递增运算符 operator --和单目递减运算符 operator++可将指针指向树里的前一个或后一个节点。MIT 出版社出版的《Introduction to Algorithms, Third Edition》（Thomas H. Cormen 等人著）介绍了这两个运算符的具体算法。

迭代器.end()指向了一个隐蔽的虚节点。其 Isnil 为 1，即是说它没有对应的关键字(key)或值（value）、不是真正意义上的数据结点，仅是一个存储控制信息的容器。这个虚节点的父节点是真正的根节点的指针，也就是整个信息树的顶点。

GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse,   bool dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
            n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
```

```
        };
    };

    if (traverse==false)
        return;

    if (n->M_left)
        dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
    if (n->M_right)
        dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n",
        m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
```

```
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";

    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, true);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, true);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, false);
};
```

指令清单 51.36　GCC 4.8.1

```
dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
```

```
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 M_color=0
key=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=0
```

GCC 的实现方法与 MSVC 十分相似，具体内容可参见：http://gcc.gnu.org/onlinedocs/libstdc++ /libstdc++-html-USERS-4.1/stl__tree_8h-source.html。与 MSVC 相比，GCC 创建的数据结构并没有 Isnil 字段，所以其内存存储结构更为紧凑。另外，迭代器.end()同样指向了一个没有任何关键字或值的虚节点。

平衡树的动态调整技术（GCC）

在平衡树里添加一个结点，可能会导致树的失衡。所以在插入结点时，需要对树进行调整，以保持树的平衡。下面的程序将演示 GCC 的调整技术。

指令清单 51.37　GCC 程序

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
```

```
    s.insert(123);
    s.insert(456);
    printf ("123, 456 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    printf ("\n");
};
```

指令清单 51.38 GCC 4.8.1 程序

```
123, 456 are inserted
root----123
        R-------456

11, 12 are inserted
root----123
        L-------11
                R-------12
        R-------456

100, 1001 are inserted
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001

667, 1, 4, 7 are inserted
root----12
        L-------4
                L-------1
                R-------11
                        L-------7
        R-------123
                L-------100
                R-------667
                        L-------456
                        R-------1001
```

第 52 章　数组与负数索引

数组的负数索引值完全不阻碍寻址。例如，array[-1]实际上表示数组 array 起始地址之前的存储空间！

这种技术的用途相当有限。笔者认为除了本章范例的这种场景之外，应该没有什么领域用得上这项技术了。众所周知，在表示数组的第一个元素时，C/C++使用的数组下标是 0，而部分其他编程语言（FORTRAN 等）使用的数组下标可能是 1。在移植代码的时候，可能会忽视这种问题。此时借助负数索引值就可以用查找 C/C++数组中的第一个元素。

```
#include <stdio.h>

int main()
{
        int random_value=0x11223344;
        unsigned char array[10];
        int i;
        unsigned char *fakearray=&array[-1];

        for (i=0; i<10; i++)
                array[i]=i;

        printf ("first element %d\n", fakearray[1]);
        printf ("second element %d\n", fakearray[2]);
        printf ("last element %d\n", fakearray[10]);

        printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X\n",
                array[-1],
                array[-2],
                array[-3],
                array[-4]);
};
```

指令清单 52.1　非优化的 MSVC 2010 下的程序

```
 1 $SG2751 DB      'first element %d', 0aH, 00H
 2 $SG2752 DB      'second element %d', 0aH, 00H
 3 $SG2753 DB      'last element %d', 0aH, 00H
 4 $SG2754 DB      'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4'
 5         DB      ']=%02X', 0aH, 00H
 6
 7 _fakearray$ = -24         ; size = 4
 8 _random_value$ = -20      ; size = 4
 9 _array$ = -16             ; size = 10
10 _i$ = -4                  ; size = 4
11 _main   PROC
12         push    ebp
13         mov     ebp, esp
14         sub     esp, 24
15         mov     DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
16         ; set fakearray[] one byte earlier before array[]
17         lea     eax, DWORD PTR _array$[ebp]
18         add     eax, -1 ; eax=eax-1
19         mov     DWORD PTR _fakearray$[ebp], eax
20         mov     DWORD PTR _i$[ebp], 0
21         jmp     SHORT $LN3@main
22         ; fill array[] with 0..9
23 $LN2@main:
```

```
24          mov     ecx, DWORD PTR _i$[ebp]
25          add     ecx, 1
26          mov     DWORD PTR _i$[ebp], ecx
27  $LN3@main:
28          cmp     DWORD PTR _i$[ebp], 10
29          jge     SHORT $LN1@main
30          mov     edx, DWORD PTR _i$[ebp]
31          mov     al, BYTE PTR _i$[ebp]
32          mov     BYTE PTR _array$[ebp+edx], al
33          jmp     SHORT $LN2@main
34  $LN1@main:
35          mov     ecx, DWORD PTR _fakearray$[ebp]
36          ; ecx=address of fakearray[0], ecx+1 is fakearray[1] or array[0]
37          movzx   edx, BYTE PTR [ecx+1]
38          push    edx
39          push    OFFSET $SG2751 ; 'first element %d'
40          call    _printf
41          add     esp, 8
42          mov     eax, DWORD PTR _fakearray$[ebp]
43          ; eax=address of fakearray[0], eax+2 is fakearray[2] or array[1]
44          movzx   ecx, BYTE PTR [eax+2]
45          push    ecx
46          push    OFFSET $SG2752 ; 'second element %d'
47          call    _printf
48          add     esp, 8
49          mov     edx, DWORD PTR _fakearray$[ebp]
50          ; edx=address of fakearray[0], edx+10 is fakearray[10] or array[9]
51          movzx   eax, BYTE PTR [edx+10]
52          push    eax
53          push    OFFSET $SG2753 ; 'last element %d'
54          call    _printf
55          add     esp, 8
56          ; subtract 4, 3, 2 and 1 from pointer to array[0] in order to find values before array[]
57          lea     ecx, DWORD PTR _array$[ebp]
58          movzx         edx, BYTE PTR [ecx-4]
59          push    edx
60          lea     eax, DWORD PTR _array$[ebp]
61          movzx   ecx, BYTE PTR [eax-3]
62          push    ecx
63          lea     edx, DWORD PTR _array$[ebp]
64          movzx   eax, BYTE PTR [edx-2]
65          push    eax
66          lea     ecx, DWORD PTR _array$[ebp]
67          movzx   edx, BYTE PTR [ecx-1]
68          push    edx
69          push    OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X,  array[-3]=%02X, array[-4]=%02 X'
70          call    _printf
71          add     esp, 20
72          xor     eax, eax
73          mov     esp, ebp
74          pop     ebp
75          ret     0
76  _main   ENDP
```

数组 array[]有 10 个字节型数据元素，它们的值依次是 0 到 9。我们还构造数组 fakearray[]和相应的指针，使 fakearrary 的数组指针地址比 array[]数组的地址提前一个字节，确保 fakearray[1]的地址和 array[0]对齐。但是我们还是很好奇，在 array[0]之前的负值索引元素的地址到底是什么。为此，我们在这里在 array[]数组的地址之存储了一个双字常数 0x11223344。只要不进行优化编译，编译器就会按照变量声明的顺序来分配其存储空间。因此，我们可以在编译后的可执行文件里验证这个存储关系：双字常数 random_value 正好排列于 array[]数组之前。

运行这个刚编译出来的可执行文件，可以看到程序运行的结果是：

```
first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44
```

请注意 x86 平台的小端字节序现象。

在调试工具 OllyDbg 的栈窗口里，我们可以看到栈内数据的排布如下：

指令清单 52.2　非优化的 MSVC2010 编译结果

```
CPU Stack
Address    Value
001DFBCC   /001DFBD3 ; fakearray pointer
001DFBD0   |11223344 ; random_value
001DFBD4   |03020100 ; 4 bytes of array[]
001DFBD8   |07060504 ; 4 bytes of array[]
001DFBDC   |00CB0908 ; random garbage + 2 last bytes of array[]
001DFBE0   |0000000A ; last i value after loop was finished
001DFBE4   |001DFC2C ; saved EBP value
001DFBE8   \00CB129D ; Return Address
```

现在，栈内地址与变量值的对应关系是：

- 数组 fakearray[]的地址是 0x001dfbd3，它比数组 array[]的地址 0x001dfbd4 落后了一个字节 （栈是逆向增长的存储结构）。

虽然这确实是某种意义上的 hack，但是它确实不很靠谱（编译结果可能和预期相差甚远）。因此，本书不建议在生产环境下使用这种代码。即使如此，本例仍然不失为典型的演示程序。

第 53 章　16 位的 Windows 程序

虽然 16 位的 Windows 程序已经近乎绝迹，但是有关复古程序以及研究加密狗的研究，往往会涉及这部分知识。

微软于 1993 年 8 月发布了最后一个 16 位的 Windows 系统，即 Windows 3.11（同年发行的中文操作系统 Windows 3.2 也是 16 位操作系统）。在此之后问世的 16/32 位混合系统 Windows 96/98/ME 系统，以及 32 位的 Windows NT 系统都可以运行 16 位应用程序。后来推出的 64 位 Windows NT 系列操作系统不再支持 16 位应用程序。

16 位应用程序的代码结构和 MSDOS 的程序十分相似。这种类型的可执行文件采用了一种名为“New Executable (NE)”的可执行程序格式。

本章的所有程序均由 OpenWatcom 1.9 编译。编译时的选项开关如下：

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

53.1　例子#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBeep(MB_ICONEXCLAMATION);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 30h ; '0'       ; MB_ICONEXCLAMATION constant
                push    ax
                call    MESSAGEBEEP
                xor     ax, ax              ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

这个程序不难分析。

53.2　例子#2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
```

```
        return 0;
};

WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; 0x18. "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; 0x10. "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp

dseg02:0010 aCaption db 'caption',0
dseg02:0018 aHelloWorld db 'hello, world',0
```

基于 Pascal 语言的调用约定要求：参数从左至右入栈（与 cdecl 相反）。调用方函数依次传递 NULL、"hello, world"、"caption"和 MB_YESNOCANCEL。这个规范还要求被调用函数恢复栈指针，所以 RETN 指令有一个 0Ah 参数，即被调用函数在退出的时候要释放 10 字节的栈空间。这种调用规范和 stdcall（参阅 64.2 节）十分相似，只是参数传递的顺序是从左到右的“自然语言”顺序，

16 位应用程序的指针是一对数据：函数首先传递的是数据段的地址，然后再传递段内的指针地址。本例子只用到了一个数据段，所以 DS 寄存器的值一直是可执行文件数据段的地址。

53.3　例子#3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};

WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; "hello, world"
                push    ax
```

```
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                cmp     ax, 2           ; IDCANCEL
                jnz     short loc_2F
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedCanc ; "you pressed cancel"
                jmp     short loc_49
loc_2F:
                cmp     ax, 6           ; IDYES
                jnz     short loc_3D
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedYes ; "you pressed yes"
                jmp     short loc_49
loc_3D:
                cmp     ax, 7           ; IDNO
                jnz     short loc_57
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedNo ; "you pressed no"
loc_49:
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax
                push    ax
                call    MESSAGEBOX
loc_57:
                xor     ax, ax
                pop     bp
                retn    0Ah
WinMain         endp
```

这段代码是基于前面那个例子进行了一些改动。

53.4 例子#4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
        return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
        return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
        return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
```

```
                HINSTANCE hPrevInstance,
                LPSTR lpCmdLine,
                int nCmdShow )
{
        func1 (123, 456, 789);
        func2 (600000, 700000, 800000);
        func3 (600000, 700000, 800000, 123);
        return 0;
};

func1           proc near

c               = word ptr  4
b               = word ptr  6
a               = word ptr  8

                push    bp
                mov     bp, sp
                mov     ax, [bp+a]
                imul    [bp+b]
                add     ax, [bp+c]
                pop     bp
                retn    6
func1           endp

func2           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_8]
                mov     dx, [bp+arg_A]
                mov     bx, [bp+arg_4]
                mov     cx, [bp+arg_6]
                call    sub_B2 ; long 32-bit multiplication
                add     ax, [bp+arg_0]
                adc     dx, [bp+arg_2]
                pop     bp
                retn    12
func2           endp

func3           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh
arg_C           = word ptr  10h

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_A]
                mov     dx, [bp+arg_C]
                mov     bx, [bp+arg_6]
                mov     cx, [bp+arg_8]
                call    sub_B2 ; long 32-bit multiplication
                mov     cx, [bp+arg_2]
```

```
                add     cx, ax
                mov     bx, [bp+arg_4]
                adc     bx, dx          ; BX=high part, CX=low part
                mov     ax, [bp+arg_0]
                cwd                     ; AX=low part d, DX=high part d
                sub     cx, ax
                mov     ax, cx
                sbb     bx, dx
                mov     dx, bx
                pop     bp
                retn    14
func3           endp
WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 123
                push    ax
                mov     ax, 456
                push    ax
                mov     ax, 789
                push    ax
                call    func1
                mov     ax, 9       ; high part of 600000
                push    ax
                mov     ax, 27C0h   ; low part of 600000
                push    ax
                mov     ax, 0Ah     ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch     ; high part of 800000
                push    ax
                mov     ax, 3500h   ; low part of 800000
                push    ax
                call    func2
                mov     ax, 9       ; high part of 600000
                push    ax
                mov     ax, 27C0h   ; low part of 600000
                push    ax
                mov     ax, 0Ah     ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch     ; high part of 800000
                push    ax
                mov     ax, 3500h  ; low part of 800000
                push    ax
                mov     ax, 7Bh    ; 123
                push    ax
                call    func3
                xor     ax, ax      ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

当 16 位系统（MSDOS 和 Win16）传递 long 型 32 位“长”数据时（这种平台上的 int 型数据是 16 位数据），它会将 32 位数据拆成 2 个 16 位数据、成对传递。这种方法和第 24 章介绍的“32 位系统处理 64 位数据”的方法十分相似。

此处的 sub_B2 是编译器开发人员编写的（仿真）库函数，用于长数据的乘法运算；即它可实现 2 个 32 位数据的乘法运算。程序中的其他库函数，请参见本书的附录 D 和附录 E。

ADD/ADC 指令分别对高低 16 位数据进行加法运算：ADD 指令可设置/清除 CF 标识位，而 ADC 指令会代入这个标识位的值。同理，SUB/SBB 指令对可实现 32 位数据的减法运算：SUB 可设置/清除 CF 标识位，SBB 会在计算过程中代入借位标识位的值。

在返回函数值的时候，32 位的返回值通过 DX：AX 寄存器对回传。

另外，当主函数 WinMain()函数向其他函数传递 32 位常量时，它也把常量拆分为 1 对 16 位数据了。

如果要把 int 型常数 123 当作 long 型 32 位参数，那么编译器就会使用 CWD 指令把 AX 里的 16 位数据符号扩展为 32 位的数据、再连同 DX 寄存器里的高 16 位数据一同传递。

53.5　例子#5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

void PASCAL remove_digits (char *s)
{
        while (*s)
        {
                if (*s>='0' && *s<='9')
                        *s='-';
                s++;
        };
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        string_compare ("asd", "def");
        string_compare_far ("asd", "def");
        remove_digits (str);
        MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
        return 0;
};

string_compare proc near
```

```
arg_0 = word ptr  4
arg_2 = word ptr  6

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
      mov     al, [bx]
      cmp     al, [si]
      jz      short loc_1C
      xor     ax, ax
      jmp     short loc_2B


loc_1C: ; CODE XREF: string_compare+Ej
      test    al, al
      jz      short loc_22
      jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
      mov     ax, 1
      jmp     short loc_2B


loc_27: ; CODE XREF: string_compare+18j
      inc     bx
      inc     si
      jmp     short loc_12


loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
      pop     si
      pop     bp
      retn    4
string_compare endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr  4
arg_2 = word ptr  6
arg_4 = word ptr  8
arg_6 = word ptr  0Ah

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
      mov     es, [bp+arg_6]
      mov     al, es:[bx]
      mov     es, [bp+arg_2]
      cmp     al, es:[si]
      jz      short loc_4C
      xor     ax, ax
      jmp     short loc_67
```

```
loc_4C: ; CODE XREF: string_compare_far+16j
      mov     es, [bp+arg_6]
      cmp     byte ptr es:[bx], 0
      jz      short loc_5E
      mov     es, [bp+arg_2]
      cmp     byte ptr es:[si], 0
      jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
      mov     ax, 1
      jmp     short loc_67


loc_63: ; CODE XREF: string_compare_far+2Cj
      inc     bx
      inc     si
      jmp     short loc_3A


loc_67: ; CODE XREF: string_compare_far+1Aj
        ; string_compare_far+31j
      pop     si
      pop     bp
      retn    8
string_compare_far endp

remove_digits proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr 4

      push    bp
      mov     bp, sp
      mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
      mov     al, [bx]
      test    al, al
      jz      short loc_86
      cmp     al, 30h ; '0'
      jb      short loc_83
      cmp     al, 39h ; '9'
      ja      short loc_83
      mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
      inc     bx
      jmp     short loc_72


loc_86: ; CODE XREF: remove_digits+Aj
      pop     bp
      retn    2
remove_digits   endp

WinMain proc near ; CODE XREF: start+EDp
      push    bp
      mov     bp, sp
      mov     ax, offset aAsd ; "asd"
      push    ax
      mov     ax, offset aDef ; "def"
      push    ax
      call    string_compare
      push    ds
      mov     ax, offset aAsd ; "asd"
```

```
        push    ax
        push    ds
        mov     ax, offset aDef ; "def"
        push    ax
        call    string_compare_far
        mov     ax, offset aHello1234World ; "hello 1234 world"
        push    ax
        call    remove_digits
        xor     ax, ax
        push    ax
        push    ds
        mov     ax, offset aHello1234World ; "hello 1234 world"
        push    ax
        push    ds
        mov     ax, offset aCaption ; "caption"
        push    ax
        mov     ax, 3 ; MB_YESNOCANCEL
        push    ax
        call    MESSAGEBOX
        xor     ax, ax
        pop     bp
        retn    0Ah
WinMain endp
```

这个程序使用了“near”和“far”两种不同类型的指针。每种指针都对应着 16 位 8086CPU 的一种特定的指针寻址模式。这方面详细内容，请参见本书第 94 章的详细介绍。

“near”指针的寻址空间是当前数据段（DS）内的所有地址。字符串比较函数 string_compare() 读取 2 个指针，把 DS 寄存器的值当作寻址所需的基（段）地址、对这两个指针进行寻址。所以此处的“mov al,[bx]”指令等效于“mov al, ds:[bx]”指令，只不过原指令没有明确标出它使用的 DS 寄存器而已。

“far”指针的寻址空间不限于当前数据段，它可以是其他 DS 段的内存地址。由于需要指定基（段）地址，所以 2 个 16 位数据才能表示 1 个 far 型指针。本例的 string_compare_far()函数从 2 对 16 位数据里提取 2 个内存地址。函数把指针的基地址存入段寄存器 ES，然后在使用 Far 指针寻址时通过基地址寻址（mov al, es:[bx]）。本章的例 2 表明，16 位程序的 MessageBox()函数（属于系统函数）使用的也是 far 指针。确实，当 Windows 内核访问文本字符串指针时，它不了解字符串指针的基地址是什么，所以在调用内核函数的时候需要指明指针的段地址。

near 指针的寻址范围是 64k，这恰好是 1 个数据段的长度。对于小型程序来说，这种指针可能就够用了、不必在每次寻址的时候都要传递指针的段地址。大型的程序通常会占用多个 64k 的数据段，所以就要在每次寻址的时候指明指针的数据段（段地址）。

代码段也有寻址意义上的差别。一个 64k 内存段就可以盛下所有指令的小型程序，可以只用 CALL NEAR 指令调用其他函数；其被调用方函数可以只用 RETN 指令返回调用方函数。但是，大型程序会占用数个代码段，它就需要使用 CALL FAR 指令、用 1 对 16 位数据作跳转的目的地址；而去其被调用方函数就必须通过 RETF 指令返回调用方函数。

这就是编译器“内存模型（memory model）”选项的实际意义。

面向 MS-DOS 和 Win16 的编译器，为各种内存模型准备了相应的不同库。这些库文件在代码指针和数据指针的寻址模式存在相应的区别。

53.6 例子#6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
```

```
int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        struct tm *t;
        time_t unix_time;

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900,   t->tm_mon, t-> tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};

WinMain         proc near

var_4           = word ptr -4
var_2           = word ptr -2

                push    bp
                mov     bp, sp
                push    ax
                push    ax
                xor     ax, ax
                call    time_
                mov     [bp+var_4], ax      ; low part of UNIX time
                mov     [bp+var_2], dx      ; high part of UNIX time
                lea     ax, [bp+var_4]      ; take a pointer of high part
                call    localtime_
                mov     bx, ax ; t
                push    word ptr [bx]       ; second
                push    word ptr [bx+2]     ; minute
                push    word ptr [bx+4]     ; hour
                push    word ptr [bx+6]     ; day
                push    word ptr [bx+8]     ; month
                mov     ax, [bx+0Ah]        ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax              ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax              ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax
                mov     sp, bp
                pop     bp
                retn    0Ah
WinMain         endp
```

“unix_time”是个 32 位数据。它首先被 Time()函数存储在寄存器对 DX:AX 里，而后被主函数复制到了 2 个本地的 16 位变量。接着这个指针（地址对）又被传递给 localtime()函数。Localtime()函数把这个指针指向的数据解析为标准库定义的 tm 结构体，返回值是这种结构体的指针。另外，这也意味着如果不使用完其返回的数值，就不应重复调用这个函数。

在调用 time()函数和 localtime()函数的时候，编译器使用的是 Watcom 调用约定：前 4 个参数分别通过 AX、DX、BX 和 CX 寄存器传递，其余参数通过数据栈传递。遵循这种调用约定的库函数，其函数名称（汇编层面）的尾部也有下划线标识。

不过，sprintf()函数遵循的调用约定既不是 PASCAL 也不是 Watcom，所以编译器使用常规的 cdecl 规范传递参数（请参考 64.1 节）。

53.6.1 全局变量

我们对刚才的例子略作改动，使用全局变量再次实现它的功能：

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900,   t->tm_mon, t-> tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};

unix_time_low    dw 0
unix_time_high   dw 0
t                dw 0

WinMain          proc near
                 push    bp
                 mov     bp, sp
                 xor     ax, ax
                 call    time_
                 mov     unix_time_low, ax
                 mov     unix_time_high, dx
                 mov     ax, offset unix_time_low
                 call    localtime_
                 mov     bx, ax
                 mov     t, ax             ; will not be used in future...
                 push    word ptr [bx]     ; seconds
                 push    word ptr [bx+2]   ; minutes
                 push    word ptr [bx+4]   ; hour
                 push    word ptr [bx+6]   ; day
                 push    word ptr [bx+8]   ; month
                 mov     ax, [bx+0Ah]      ; year
                 add     ax, 1900
```

```
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax            ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax            ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax            ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

虽然编译器保留了汇编宏 t 的赋值指令，但是这个值实际上没有被后续代码调用。因为编译器无法判断其他模块（文件）是否会访问这个值，所有保留了有关的赋值语句。

第四部分

Java



第 54 章　Java

54.1　简介

Java 程序的反编译工具已经十分成熟了。一般来讲，它们都是 JVM（基于栈机制的 Java 虚拟机）的字节码（bytecode，指令流里的指令只有一个字节，故而得名。不过 java 指令中的操作数属于变长信息）分析工具。著名的 JAD（http://varaneckas.com/jad/）就是一款颇具代表性的 JAVA 反编译工具。

相对于 x86 平台更底层指令的反编译技术来说，面向 JVM 的 bytecode 更容易反编译。这主要是因为：

① 字节码含有更为丰富的数据类型信息。

② JVM 内存模型更严格，因此字节码分析起来更为有章可循。

③ Java 编译器不做任何优化工作（而 JVM JIT 在运行时会做优化工作），因此在反编译字节码之后，我们基本可以直接理解 Java 类文件里的原始指令。

什么时候 JVM bytecode 反编译有用呢？

① 无需重新编译反汇编的结果，而能给类文件做应急补丁。

② 分析混淆代码。

③ 需要编写自己的代码混淆器。

④ 创建面向 JVM 的、类似编译程序的代码生成工具（类似 Scala,Clohure 等等）。

让我们从简单的代码开始演示。除非特别指明，否则我们这里用到的都是 JDK 1.7 的自带工具。

反编译类文件的命令是：javap –c –verbose。

笔者采用的例子摘自于参考书目 Jav13。

54.2　返回一个值

或许 Java 的最简函数是直接返回数值、不做其他操作的函数。当然，功能再少一点的、什么操作都没有的“闲置”函数，肯定不存在。函数必须具有某种行为，因此统称为“方法”。在 Java 的概念中，“类/class”是一切对象的模版，所有方法必定不能脱离“类”而单独存在。但是为了简化起见，本文还是把“方法”称为“函数”。

```
public class ret
{
        public static int main(String[] args)
        {
                return 0;
        }
}
```

我们采用命令 javac 来编译它，命令行是：

```
javac ret.java
```

编译完后，我们可以用 JDK 自带的反汇编器－Javap 来分析字节码，此时采用的命令应当是：

```
javap -c -verbose ret.class
```

反编译完成后，我们得到的代码如下所示。

指令清单 54.1 JDK 1.7（摘要）

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: iconst_0
       1: ireturn
```

Java 平台的开发人员认为，0 是用得最多的常量。因此他们为 PUSH 0 的指令单独设计了单字节的指令码，即 iconst_0。此外还有 iconst_1（将 1 入栈），iconst_2（将 2 入栈）……，一直到 iconst_5 这样的单字节字节码。而且确实有 iconst_m1（将−1 入栈）这类的将负数推送入栈的单字节指令。

JVM 常常采用栈的方式来传递参数并从函数中返回值。因此语句 iconst_0 将数字 0 压入栈，而指令 ireturn 则是从栈顶返回整型数（ireturn 中的字母 i 的意思就是“返回值为 integer/整数”）。这里注意我们用 TOS 来代表栈顶，它是英文“Top Of Stack”的首字母缩写。

我们来重新编写一下这个例子，将返回值修改为整数 1234：

```
public class ret
{
        public static int main(String[] args)
        {
                return 1234;
        }
}
```

这样的话，我们得到的结果是如下所示的代码。

指令清单 54.2 JDK1.7（摘要）

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: sipush        1234
       3: ireturn
```

指令 sipush 的功能是将操作数（这里是整数 1234）入栈（si 是 short integer 短型整数的缩写）。Short（短型）的意思就是针对 16 位的数值进行操作，而这里的整数 1234 正好就是一个 16 位的数值。

如果操作数比整型数据更大，那么字节码会是什么情况呢？让我们来看看实例：

```
public class ret
{
        public static int main(String[] args)
        {
                return 12345678;
        }
}
```

指令清单 54.3 常量池

```
...
   #2 = Integer            12345678
...

  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // int 12345678
         2: ireturn
```

JVM 的 opcode 无法直接封装 32 位数据，这是开发环境的局限决定的。像本例这样的 32 位常数将会

存储到“常量池”里。常量池是一个由数组组成的表，类型为 cp_info constant_pool[]，用来存储程序中使用的各种常量，包括 Class/String/Integer 等各种基本 Java 数据类型，详情参见 The Java Virtual Machine Specification 4.4 节。

并非只有 JVM 如此处理常量。像 MIPS、ARM 以及其他的 RISC 型的 CPU 都不能在 32 位的 opcode 中封装 32 位常量，因此包括 MIPS 和 ARM 在内的 RISC 类型的 CPU 都得分步骤构建这些数值，或者将其保存在数据段中。有关范例可以参考本书的 28.3 节或者 29.1 节。

在 MIPS 的概念中也有传统意义上的常量池，不过它的名字则叫做“数据缓冲池（文字池）/literal pool”。这种文字池与可执行程序中的“.lit4/.lit8”数据段相对应。数据段.lit4 用于保存 32 位的单精度浮点常数，而.lit8 则用于保存 64 位的双精度浮点常数。

我们来试试其他类型的数据。

布尔型 Boolean:

```
public class ret
{
        public static boolean main(String[] args)
        {
                return true;
        }
}
```

```
  public static boolean main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_1
         1: ireturn
```

当返回值为 Ture 时，JVM bytecode 层面的返回值就是整数 1。像 C/C++一样，Java 程序同样会把布尔型数值保存在 32 位的栈中。虽然说“逻辑真”和“整数 1”的数值完全相同，但是我们不可能把布尔值当作整数值使用、也不可能把整数值当作布尔值使用。既定的类文件事先声明了数值的数据类型，而且这些数据类型会在程序运行时被实时检查。

16 位的短整数型也是一样：

```
public class ret
{
        public static short main(String[] args)
        {
                return 1234;
        }
}
```

```
  public static short main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: sipush        1234
         3: ireturn
```

还有字符型：

```
public class ret
{
        public static char main(String[] args)
        {
                return 'A';
        }
}
```

```
  public static char main(java.lang.String[]);
```

```
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        65
         2: ireturn
```

指令 bipush 的意思是 push byte（保存字节）。Java 环境中的 car 型数据是 16 位的 UTF-16 字符，同短整数型数据一样同属于 16 位 short 型短数据。但是大写字母 A 的 ASCII 码是十进制数 65，而且我们可以用指令将一个字节的数压入栈中。

下面我们来看看 byte（字节）：

```
public class retc
{
        public static byte main(String[] args)
        {
                return 123;
        }
}

  public static byte main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        123
         2: ireturn
```

也许读者有些疑问：既然这些数据在运行的时候都是当作 32 位整型数据处理的，那么为什么还要不厌其烦地把它们声明为 16 位的数据类型呢？另外，字符型 char 数据与 short 短整数型的数值也是相同的，为什么还要刻意地把它声明为字符型 char 数据呢？

答案也很简单，是为了增加数据类型的控制以及增加源代码的可读性。Char 字符型的限定符虽然在数值上与 short 短型整数相同，但是只要一看到 char 字符型的限定，我们立刻会联想到它是一个 UTF16 的字符集，而不会把它当作其他类型的方式去理解。在遇到被限定符 short 修饰的数据类型时，我们自然而然地就会把它理解为 16 位数据。同理，应当使用 boolean 声明的数据就不要把它声明为 C 语言风格的 int 型数据。

我们还可以通过限定符 long 声明 JAVA 的 64 位的整数型数据：

```
public class ret3
{
        public static long main(String[] args)
        {
                return 1234567890123456789L;
        }
}
```

指令清单 54.4　常量池

```
...
   #2 = Long 1234567890123456789l
...

  public static long main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc2_w        #2                  // long 1234567890123456789l
         3: lreturn
```

上述 64 位常量同样位于程序的常量池部分。它被 ldc2_w 指令提取之后，再由 lreturn（long return）指令回传给调用方函数。ldc2_w 指令也能够从常量池里提取双精度浮点数（同样是 64 位常量）。

```
public class ret
{
        public static double main(String[] args)
        {
                return 123.456d;
        }
}
```

指令清单 54.5　常量池

```
...
   #2 = Double              123.456d
...

  public static double main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
        0: ldc2_w        #2                  // double 123.456d
        3: dreturn
```

这里的指令 dreturn 代表 return double，意思是返回双精度常数。

最后，我们举一个单精度浮点数的例子。单精度浮点常数的后面有一个限定符 f，而双精度数的限定符则是字母 d。字母 f 是 float 的缩写，而 d 则是 double 的缩写。

```
public class ret
{
        public static float main(String[] args)
        {
                return 123.456f;
        }
}
```

指令清单 54.6　常量池

```
...
   #2 = Float               123.456f
...

  public static float main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // float 123.456f
         2: freturn
```

同为从常量池提取 32 位数据的指令，“提取整数”和“提取浮点数”的指令都是 ldc。而指令 freturn 代表的是 return float，声明了返回值为单精度浮点数。

最后，我们来看看如果什么数也不返回时情况会是怎么样的，也就是 return 指令后不带任何参数。

```
public class ret
{
        public static void main(String[] args)
        {
                return;
        }
}

  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
```

没有返回值的函数，最后只有一条 return 指令。它不返回任何值，只是把程序控制流递交给调用方函

数。根据函数最后一条返回值处理指令，我们就能比较容易地推导出函数返回值的数据类型。

54.3 简单的计算函数

我们继续来看看简单的计算函数：

```
public class calc
{
        public static int half(int a)
        {
                return a/2;
        }
}
```

这里我们看到的是一个除以 2 的简单计算函数，用到的指令是 iconst_2。我们来分析一下这几条指令：

```
  public static int half(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: iload_0
         1: iconst_2
         2: idiv
         3: ireturn
```

首先，iload_0 指令是提取外来的第 0 个函数参数，再把它压入栈中，而 iconst_2 指令则是将数值 2 压入栈中。这两个指令执行完后，函数栈的存储内容将如下所示：

```
      +---+
TOS ->| 2 |
      +---+
      | a |
      +---+
```

TOS 是“Top Of Stack”的缩写，即栈顶。

idiv 指令则是从栈顶取出这两个值并进行除非运算，然后把返回的结果保存在栈顶。

```
      +--------+
TOS ->| result |
      +--------+
```

ireturn 指令则会提取栈顶的数据、把它作为返回值回传给调用方函数。

下面我们来看看双精度的除法的运算指令：

```
public class calc
{
        public static double half_double(double a)
        {
                return a/2.0;
        }
}
```

指令清单 54.7 常量池

```
...
   #2 = Double            2.0d
...

  public static double half_double(double);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=2, args_size=1
```

```
        0: dload_0
        1: ldc2_w       #2                  // double 2.0d
        4: ddiv
        5: dreturn
```

双精度浮点数的运算指令和单精度浮点数的指令十分相似。在提取常量时，它使用的指令时 ldc2_w。此外，所有的三条运算指令（dload_0、ddiv 以及 dreturn）都带有前缀 d，这个限定符表明操作数属于双精度浮点数 double。

下面我们来看看含有两个参数的函数情况：

```
public class calc
{
        public static int sum(int a, int b)
        {
                return a+b;
        }
}
```

```
  public static int sum(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: iadd
         3: ireturn
```

指令 iload_0 用于提取第一个函数参数 *a*，而 iload_1 则用于导入第二个函数参数 *b*。在执行完这两条指令之后，栈内数据如下图所示：

```
      +---+
TOS ->| b |
      +---+
      | a |
      +---+
```

而指令 iadd 的含义则是将两个参数中的数值相加，并将结果保存在栈顶 TOS 中。

```
      +--------+
TOS ->| result |
      +--------+
```

如果我们将以上函数的两个参数的数据类型更换成 long 类型的话：

```
        public static long lsum(long a, long b)
        {
                return a+b;
        }
```

我们看到的字节码则会变为：

```
public static long lsum(long, long);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=4, args_size=2
       0: lload_0
       1: lload_2
       2: ladd
       3: lreturn
```

第二条 lload 指令会提取外来的第二个参数。指令后缀直接从 0 递增到 2，是因为 long 型数据是 64 位的数据，它正好占有两个 32 位数据的存储位置（即后文介绍的“参数槽”）。

下面我们来看看一个更加复杂的例子：

```
public class calc
{
        public static int mult_add(int a, int b, int c)
        {
                return a*b+c;
        }
}
```

```
  public static int mult_add(int, int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=3
         0: iload_0
         1: iload_1
         2: imul
         3: iload_2
         4: iadd
         5: ireturn
```

第一步的运算是乘法，乘积的结果存放在栈顶 TOS 中。

```
      +----------+
TOS ->| product |
      +----------+
```

指令 iload_2 将第三个参数压入栈中参加运算：

```
      +----------+
TOS ->|     c    |
      +----------+
      | product |
      +----------+
```

现在就能采用指令 iadd 进行加法求和运算了。

54.4 JVM 的内存模型

前面提到过，在 x86 和其他底层运行平台上，栈通常用于传递参数的参数、存储局部变量。而我们这里要提到的 JVM 略有不同。

JVM 的内存模型可分为：

- 局部变量数组（Local Variable Array，LVA）。它用来存储外来的函数参数和局部变量。iload_0 一类指令的作用是从 LVA 中提取数值，而 istore 则可以将数值保存在 LVA 里。函数会从第 0 个参数槽（不涉及 this 指针的函数）或第 1 个参数槽（涉及传递 this 指针的函数）依次读取各项外来参数，然后分配局部变量的存储空间。

  每个参数槽（args slot）都是 32 位存储单元，因此，数据类型为 long（长）或者 double（双）的参数数据都会占用两个参数槽。
- 操作数栈即俗称的（java）“栈”，用于存储计算操作数，或者向被调用方函数传递参数。Java 程序不能直接运行于 x86 那样的底层硬件环境，因此它必须通过明确的入栈、出栈指令才能访问自己的栈，不能像汇编指令那样直接对栈寻址。
- 堆。堆主要用于存储对象和数组。

以上 3 种内存模型相互独立、互相隔离。

54.5 简单的函数调用

Math.random()函数可以产生从 0.0～1.0 之间的任意（伪）随机数。因此，如欲生成 0.0～0.5 之间的随

机数，就要对上述结果进行除法运算：

```
public class HalfRandom
{
        public static double f()
        {
                return Math.random()/2;
        }
}
```

指令清单 54.8 常量池

```
...
   #2 = Methodref          #18.#19        // java/lang/Math.random:()D
   #3 = Double             2.0d
...
  #12 = Utf8               ()D
...
  #18 = Class              #22            // java/lang/Math
  #19 = NameAndType        #23:#12        // random:()D
  #22 = Utf8               java/lang/Math
  #23 = Utf8               random

  public static double f();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=0, args_size=0
         0: invokestatic  #2             // Method java/lang/Math.random:()D
         3: ldc2_w        #3             // double 2.0d
         6: ddiv
         7: dreturn
```

指令 invokestatic 调用函数 Math.random()，并将结果保存在栈顶 TOS。这个值随后被除以 2，最终成为函数返回值。但是这些函数的名称是如何编码的？编译器使用了 Methodref 的表达方法把外部函数的信息编排在常量池中。常量池里的相应数据声明了与被调用函数有关的类（Class）以及方法（NameAndType）名称。Methodref 表达式的第一个字段（Fieldref 里的第一个值）是 Class 的索引号-#18，这个索引号（实际上是指针）对应着一个 Class 名称（依次查询#18、#22 号常量，可得到 java/lang/Math）。Methodref 表达式的第二个字段（Fieldref 里的第二个值）是方法名称的索引号#19，这个索引号对应着方法名称 NameAndType（依次查询#19、#23 号常量，可得到方法名称 random）。方法名称由 2 个索引号组成，第一个索引号对应着外部函数名称，而第二个索引号对应着函数返回值的数据类型“()D”——双精度浮点数。

综合上述信息可知：

① JVM 能检查数据类型的正确性。

② JAVA 的反编译器能从已经编译好的类文件中恢复出其原来的数据类型。

最后，我们来看一个经典的字符串显示例子：Hello, world!

```
public class HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println("Hello, World");
        }
}
```

指令清单 54.9 常量池

```
...
   #2 = Fieldref           #16.#17        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #18            // Hello, World
   #4 = Methodref          #19.#20        // java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
...
  #16 = Class              #23            // java/lang/System
  #17 = NameAndType        #24:#25        // out:Ljava/io/PrintStream;
  #18 = Utf8               Hello, World
  #19 = Class              #26            // java/io/PrintStream
  #20 = NameAndType        #27:#28        // println:(Ljava/lang/String;)V
...
  #23 = Utf8               java/lang/System
  #24 = Utf8               out
  #25 = Utf8               Ljava/io/PrintStream;
  #26 = Utf8               java/io/PrintStream
  #27 = Utf8               println
  #28 = Utf8               (Ljava/lang/String;)V
...

  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String Hello, World
         5: invokevirtual  #4                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
```

偏移量为 3 的 ldc 指令从常量池中提取字符串 Hello,World 的指针，然后将其压入栈中。在 Java 中，这种二级指针称为 reference（引用），但是它的本质仍然还是指针或者地址[①]。

熟悉的指令 invokevirtual 从常量池中提取 println 函数的信息，然后调用该函数。我们已经知道，标准库定义了许多版本的 println()函数，每个版本都处理的数据类型都各不相同。本例调用的 println()函数，肯定是专门处理 string 型数据的那个版本。

第一条指令 getstatic 的功能是什么呢？这个指令从对象 System.out 中提取引用指针的有关字段，再把它压入栈。这个引用指针的作用与 println 方法的 this 指针相似。因此，从内部来讲，println 函数的输入参数实际上是两个指针：①this 指针，也就是指向对象的指针；②字符串“Hello,World”的地址。

因此这并不矛盾：只有在 System.out 初始化为实例的时候，才能调用 println()方法。

为了方便分析人员阅读， javap 把有关信息全部追加到了字节码的注释里了。

54.6 调用函数 beep()（蜂鸣器）

这是一个最简单的调用（调用了无参数的两个函数），其功能就是发出蜂鸣声 beep：

```
    public static void main(String[] args)
    {
            java.awt.Toolkit.getDefaultToolkit().beep();
    };

  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: invokestatic  #2                  // Method java/awt/Toolkit.getDefaultToolkit:()↙
    ↳ Ljava/awt/Toolkit;
         3: invokevirtual #3                  // Method java/awt/Toolkit.beep:()V
         6: return
```

第一条指令是偏移量为 0 的 invokestatic 指令，它调用了 java.awt.Toolkit.getDefaultToolkit()函数。后者的返回值是 Toolkit Class 类实例的引用指针。偏移量为 3 的 invokevirtual 指令调用这个类的 beep()函数。

① 关于常规指针和引用指针的详细区别可以查阅本书的 51.3 节。

54.7 线性同余随机数产生器（PRNG）

我们再来看看 PRNG（Pseudo Random Numbers Generator）随机数产生器，其实我们已经在本书的第 20 章介绍过它的 C 语言代码了。

```
public class LCG
{
        public static int rand_state;

        public void my_srand (int init)
        {
                rand_state=init;
        }

        public static int RNG_a=1664525;
        public static int RNG_c=1013904223;

        public int my_rand ()
        {
                rand_state=rand_state*RNG_a;
                rand_state=rand_state+RNG_c;
                return rand_state & 0x7fff;
        }
}
```

程序在启动之初就初始化了数个成员变量（Class Fields）。这是如何进行的呢？这就得借助 javap 查看该类的构造函数：

```
  static {};
    flags: ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: ldc           #5                  // int 1664525
         2: putstatic     #3                  // Field RNG_a:I
         5: ldc           #6                  // int 1013904223
         7: putstatic     #4                  // Field RNG_c:I
        10: return
```

上述指令展示了变量的初始化过程。变量 RNG_a 和 RNG_c 分别占据参数槽的第三和第四存储单元。而 putstatic 函数将有关常数存放在相应地址。

函数 my_rand()将输入值保存在变量 rand_state 中：

```
  public void my_srand(int);
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=2, args_size=2
         0: iload_1
         1: putstatic     #2                  // Field rand_state:I
         4: return
```

iload_1 指令提取输入变量，然后将其压入栈中。但是为什么此处是 iload_1 指令而不是读取第 0 个参数的 iload_0 指令？这是由于该函数调用了类的成员变量，因此要用第 0 个参数传递 this 指针。依此类推，函数的第二个参数槽用于传递成员变量、同时是隐性参数 rand_state，此后的 putstatic 指令把栈顶的数值肤之道第二个参数槽、完成指定任务。

现在我们来看看 my_rand()函数：

```
  public int my_rand();
    flags: ACC_PUBLIC
    Code:
```

```
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field rand_state:I
       3: getstatic     #3                  // Field RNG_a:I
       6: imul
       7: putstatic     #2                  // Field rand_state:I
      10: getstatic     #2                  // Field rand_state:I
      13: getstatic     #4                  // Field RNG_c:I
      16: iadd
      17: putstatic     #2                  // Field rand_state:I
      20: getstatic     #2                  // Field rand_state:I
      23: sipush        32767
      26: iand
      27: ireturn
```

这段代码分别提取类实例的各成员变量、进行各种运算，再使用 putstatic 指令更新 rand_state 的值。在偏移量为 20 处，rand_state 的值会重新调入（此前的 putstatic 指令把它从栈里抛了出去）。虽然表面看来这个程序的效率很低，但是 JVM 肯定能够进行充分的优化、足以弥补字节码的效率缺陷。

54.8 条件转移

我们来看一个简单的例子：

```
public class abs
{
        public static int abs(int a)
        {
                if (a<0)
                        return -a;
                return a;
        }
}
```

```
  public static int abs(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: ifge          7
         4: iload_0
         5: ineg
         6: ireturn
         7: iload_0
         8: ireturn
```

如果栈顶/TOS 的值大于或等于零，那么 ifge 将会跳转到偏移量为 7 的指令。特别需要注意的是，ifxx 指令还会从栈顶抛弃一个值，否则它就不能进行比较运算。

后面的 ineg 指令是对整数求负的运算指令。

我们再看一个例子：

```
        public static int min (int a, int b)
        {
                if (a>b)
                        return b;
                return a;
        }
```

这个函数的字节码如下所示：

```
  public static int min(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
```

```
    stack=2, locals=2, args_size=2
       0: iload_0
       1: iload_1
       2: if_icmple      7
       5: iload_1
       6: ireturn
       7: iload_0
       8: ireturn
```

if_icmple 指令从栈中提取（pop）两个数值并将之进行比较。如果第二个操作数小于或等于第一个操作数，那么它将跳转到偏移量为 7 的指令，否则继续执行下一条指令。

若对上述程序进行调整，通过较大值函数 max()进行比较：

```
public static int max (int a, int b)
{
        if (a>b)
                return a;
        return b;
}
```

那么字节码则会变为：

```
public static int max(int, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=2
       0: iload_0
       1: iload_1
       2: if_icmple      7
       5: iload_0
       6: ireturn
       7: iload_1
       8: ireturn
```

其实与刚才取较小值的程序基本相同，但是最后两个 iload 指令（在偏移量为 5 和 7 的位置上）位置对换了一下。

下面再看一个更复杂一些的例子：

```
public class cond
{
        public static void f(int i)
        {
                if (i<100)
                        System.out.print("<100");
                if (i==100)
                        System.out.print("==100");
                if (i>100)
                        System.out.print(">100");
                if (i==0)
                        System.out.print("==0");
        }
}
```

```
public static void f(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: iload_0
       1: bipush        100
       3: if_icmpge     14
       6: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       9: ldc           #3                  // String <100
      11: invokevirtual #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      14: iload_0
```

```
15: bipush        100
17: if_icmpne     28
20: getstatic     #2            // Field java/lang/System.out:Ljava/io/PrintStream;
23: ldc           #5            // String ==100
25: invokevirtual #4            // Method java/io/PrintStream.print:(Ljava/lang/String;)V
28: iload_0
29: bipush        100
31: if_icmple     42
34: getstatic     #2            // Field java/lang/System.out:Ljava/io/PrintStream;
37: ldc           #6            // String >100
39: invokevirtual #4            // Method java/io/PrintStream.print:(Ljava/lang/String;)V
42: iload_0
43: ifne          54
46: getstatic     #2            // Field java/lang/System.out:Ljava/io/PrintStream;
49: ldc #7 // String ==0
51: invokevirtual #4            // Method java/io/PrintStream.print:(Ljava/lang/String;)V
54: return
```

这段代码用以实现两个功能。首先，它可以判断输入的数与 100 的大小关系，如果是小于 100 的话，显示“<100”的字样；如果是等于 100，则显示“=100”的字样；如果是大于 100 的话，则显示“>100”的字样。另外一个功能是一个特例，就是如果输入的数为 0 的话，则显示“==0”字样。

我们这里还是用到了前面提到的指令 ifXX。还记得其功能吧？

指令 if_icmpge 从栈中提取（pop）出两个值，然后对它们进行比较。如果第二个数大于第一个数的话，那么就跳转到偏移量为 14 的位置；其实指令 if_icmpne 和指令 if_icmple 的运行机理基本相同，只是其转移的条件不相同而已。

在偏移量为 43 的位置，我们还可以看到一个指令 ifne。我们认为这是一个措辞不当的助记符，如果把它的助记符换位 ifnz 似乎更加贴切一些（意思是当栈顶的值不是 0 时跳转），而实际的执行过程也是这样的——当输入的值不是 0 时，程序会跳转到偏移量为 54 的地方。而如果输入值为 0，程序的执行流则不会发生跳转、继续执行偏移量为 46 的指令、显示“==0”这个字符串。

必须注意的是：JVM 没有无符号数的数据类型。因此我们只会遇到比较有符号数的比较指令。

54.9 传递参数

我们来将前面讲到的两个取较大值和较小值的函数混合在一起使用，也就是函数 min()和函数 max()。

```
public class minmax
{
        public static int min (int a, int b)
        {
                if (a>b)
                        return b;
                return a;
        }

        public static int max (int a, int b)
        {
                if (a>b)
                        return a;
                return b;
        }

        public static void main(String[] args)
        {
                int a=123, b=456;
                int max_value=max(a, b);
                int min_value=min(a, b);
                System.out.println(min_value);
                System.out.println(max_value);
```

```
        }
}
```

下面是主函数 main()的代码：

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=5, args_size=1
       0: bipush        123
       2: istore_1
       3: sipush        456
       6: istore_2
       7: iload_1
       8: iload_2
       9: invokestatic  #2                  // Method max:(II)I
      12: istore_3
      13: iload_1
      14: iload_2
      15: invokestatic  #3                  // Method min:(II)I
      18: istore 4
      20: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      23: iload 4
      25: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      28: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      31: iload_3
      32: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      35: return
```

调用方函数通过栈向被调用方函数传递参数，而被调用方函数通过 TOS()/栈顶向调用方函数传递返回值。

54.10 位操作

JVM 的位操作指令和其他指令集的工作原理基本相同。

```
        public static int set (int a, int b)
        {
                return a | 1<<b;
        }

        public static int clear (int a, int b)
        {
                return a & (~(1<<b));
        }
```

```
public static int set(int, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=2
       0: iload_0
       1: iconst_1
       2: iload_1
       3: ishl
       4: ior
       5: ireturn

public static int clear(int, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=2
       0: iload_0
       1: iconst_1
       2: iload_1
       3: ishl
```

```
        4: iconst_m1
        5: ixor
        6: iand
        7: ireturn
```

指令 iconst_m1 将-1 这个数调入栈中，其实这个值就是 0Xffffffff。将任意数与-1 进行异或 XOR 运算，其实就是对原操作数的所有位逐位取反（这一点可以参见本书的附录 A.6.2）。

而当我们将所有的数据类型扩展为 64 位的 long 类型时：

```
public static long lset (long a, int b)
{
        return a | 1<<b;
}

public static long lclear (long a, int b)
{
        return a & (~(1<<b));
}
```

```
public static long lset(long, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=3, args_size=2
       0: lload_0
       1: iconst_1
       2: iload_2
       3: ishl
       4: i2l
       5: lor
       6: lreturn

public static long lclear(long, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=3, args_size=2
       0: lload_0
       1: iconst_1
       2: iload_2
       3: ishl
       4: iconst_m1
       5: ixor
       6: i2l
       7: land
       8: lreturn
```

除了操作指令都具有一个表示操作数是 64 位值的“L”前缀之外，这个程序的字节码和上一个程序几乎相同。此外，第二个函数的参数还有一个整型数据。假如需要把 int 型的 32 位数据扩展为 64 位 long 型数据，那么编译器就会分配 i2l 完成这项任务。

54.11 循环

```
public class Loop
{
        public static void main(String[] args)
        {
                for (int i = 1; i <= 10; i++)
                {
                        System.out.println(i);
                }
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: iconst_1
         1: istore_1
         2: iload_1
         3: bipush        10
         5: if_icmpgt     21
         8: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        11: iload_1
        12: invokevirtual #3                  // Method java/io/PrintStream.println:(I)V
        15: iinc          1, 1
        18: goto          2
        21: return
```

这里举一个例子，显示从 1～10 一共 10 个整型数。采用循环指令的方式。

指令 iconst_1 将数值 1 调入栈顶 TOS，而 istore_1 指令则将局部变量阵列 LVA 中的这项数值存储在第一个参数槽里。为什么把它存储在 1 号参数槽而不是第 0 个参数槽呢？这是因为主函数 main()有一个参数是 String 数组，这个字符串的引用指针会占用第 0 号参数槽。

因此，局部变量 *i* 必须存放于第 1 个参数槽。

在偏移量为 3 和 5 的地方的指令，分别将变量 i 与循环控制变量的上限（这里是 10）比较。如果此时的变量 *i* 比 10 大，那么指令流就会转向偏移量为 21 的地方，直接退出函数；否则，程序就会调用 println()函数显示当前的数值。显示完后，在偏移量为 11 的地方，局部变量 *i* 会重新装入新的值，继续为显示方法做准备。另外，在调用 println 方法的时候，我们提供的参数时 integer 型参数。注释中的“(I)V”分别表示数据类型为 integer，函数类型为 void。

当显示函数 println 结束时，*i* 的数值在偏移量为 15 的地方递增，也就是加 1。这条指令有两个操作数。第一个操作数，第一个操作数表示实际运算数存储于第一号参数槽，第二个操作数表示递增的增量是 1。

goto 指令的功能就是跳转/GOTO。它跳转到循环体中偏移量为 2 的地方。

让我们来看一个稍微复杂一些的例子：斐波那契数列，简称为 Fibonacci，其实前面已经提到过了。但是这里我们来看看如何用程序来实现它。

```
public class Fibonacci
{
        public static void main(String[] args)
        {
                int limit = 20, f = 0, g = 1;

                for (int i = 1; i <= limit; i++)
                {
                        f = f + g;
                        g = f - g;
                        System.out.println(f);
                }
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=5, args_size=1
         0: bipush        20
         2: istore_1
         3: iconst_0
         4: istore_2
         5: iconst_1
         6: istore_3
```

```
 7: iconst_1
 8: istore        4
10: iload         4
12: iload_1
13: if_icmpgt     37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic     #2              // Field java/lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3              // Method java/io/PrintStream.println:(I)V
31: iinc          4, 1
34: goto          10
37: return
```

我们来看看本地存储数组（Local Varible Array，LVA）与各参数槽的存储关系：

① 0 号参数槽存储的是主函数 main()的唯一参数；

② 1 号参数槽存储的是循环控制变量 limit，其值固定为 20；

③ 2 号参数槽存储的是变量 *f*。

④ 3 号参数槽存储的是变量 *g*。

⑤ 4 号参数槽存储的是变量 *i*。

可见，Java 编译器会按照源代码声明变量的顺序，在 LVA 中依次分配各变量的存储空间。

当直接向第 0、1、2、3 号参数槽存储数据时，可使用专用的 istore_n 指令。然而当直接向 4 及更高编号的参数槽存储数据时，就没有这样便利的专用操作指令了，需要使用带有参数的 istore 指令。如偏移量为 8 的指令所示，后一种 istore 指令将操作数当作参数槽的编号进行存储操作。其实 iload 指令也是如此。本文就不再解释偏移量为 10 的 iload 指令了。

但是，像循环迭代上限 limit 这样的常量也占用了参数槽，难道它还经常更新数值吗？JVM JIT 编译器能够充分优化这类事务，我们不必专们进行人工干预。

54.12 switch()语句

下列范例证明，switch()语句是由 tableswitch 指令实现的。

```
public static void f(int a)
{
        switch (a)
        {
        case 0: System.out.println("zero"); break;
        case 1: System.out.println("one\n"); break;
        case 2: System.out.println("two\n"); break;
        case 3: System.out.println("three\n"); break;
        case 4: System.out.println("four\n"); break;
        default: System.out.println("something unknown\n"); break;
        };
}
```

上述程序的字节码与源程序几乎是逐一对应：

```
public static void f(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
```

```
       0: iload_0
       1: tableswitch   { // 0 to 4
                    0: 36
                    1: 47
                    2: 58
                    3: 69
                    4: 80
              default: 91
          }
      36: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      39: ldc           #3                  // String zero
      41: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      44: goto          99
      47: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      50: ldc           #5                  // String one\n
      52: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang /String;)V
      55: goto          99
      58: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      61: ldc           #6                  // String two\n
      63: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      66: goto          99
      69: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      72: ldc           #7                  // String three\n
      74: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      77: goto          99
      80: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      83: ldc           #8                  // String four\n
      85: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      88: goto          99
      91: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      94: ldc           #9                  // String something unknown\n
      96: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      99: return
```

如果输入值是 0，则显示为 zero；如果输入值是 1，则显示 one；如果输入值是 2，则显示 two；如果输入值是 3，则显示 three；如果输入值是 4，则显示 four；如果不是以上的 5 种情况，则显示字符串 something unknown。

54.13　数组

54.13.1　简单的例子

我们首先创建一个含有 10 个元素的整数型数组，然后逐次填入 0～9：

程序如下：

```
public static void main(String[] args)
{
        int a[]=new int[10];
        for (int i=0; i<10; i++)
                a[i]=i;
        dump (a);
}
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
       0: bipush        10
       2: newarray      int
       4: astore_1
       5: iconst_0
       6: istore_2
```

```
 7: iload_2
 8: bipush        10
10: if_icmpge     23
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc          2, 1
20: goto          7
23: aload_1
24: invokestatic  #4                  // Method dump:([I]V
27: return
```

指令 newarray 创建一个可容纳 10 个整型（int）元素的数组。这个数组的大小是由 bipush 设定的，它会被保存在栈顶 TOS；而数组的类型则是由 newarray 指令的操作数定义。我们看到 newarrary 的操作数是 int，因此它会创建整型数组。执行完指令 newarray 后，系统会给新建的数组分配一个引用指针（reference），并且把这个引用指针存储到栈顶 TOS。其后的 astore_1 指令吧引用指针存储到 LVA 的第一个参数槽。主函数 main()的第二部分是一个循环语句。这个循环执行的指令将变量 *i* 依次保存到相应的数值单元中。指令 aload_1 获取数组的引用指针，并且把它保存在栈中。而指令 iastore 的功能则把栈里的整型数据保存在数组中，与此同时它会通过栈顶 TOS 获取数组的引用指针。而主函数 main()的第三部分的功能是执行函数 dump()。在偏移量为 23 的地方，我们可以看到 aload_1 指令。它负责制备 dump()的唯一参数。

下面我们来继续看看函数 dump()的功能：

```
public static void dump(int a[])
{
        for (int i=0; i<a.length; i++)
                System.out.println(a[i]);
}
```

该函数执行的功能是循环显示目标数组中的值。

程序如下所示。

```
public static void dump(int[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
       0: iconst_0
       1: istore_1
       2: iload_1
       3: aload_0
       4: arraylength
       5: if_icmpge      23
       8: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      11: aload_0
      12: iload_1
      13: iaload
      14: invokevirtual  #3                  // Method java/io/PrintStream.println:(I)V
      17: iinc           1, 1
      20: goto           2
      23: return
```

函数会从第 0 号参数槽获取数组的引用指针/reference。而源代码中的 a.length 表达式被编译器转换成了 arraylength（数组长度）指令：它通过引用指针获取数组的信息，并把数组长度保存在栈顶 TOS 中。在偏移量为 13 的指令 iaload 则负责加载既定的数组元素。在数组类型确定的情况下，对某个数组元素寻址需要知道数组的首地址和即定元素的索引编号。前者由偏移量为 11 的指令 aload_0 完成；后者则通过偏移量为 12 的指令 iload_1 实现。

很多人会想当然的认为，在那些带有字母前缀 a 的指令里，a 大概是数组 array 的缩写。其实这种猜测并不确切。此类指令是操作数据对象引用指针的指令。数组和字符串只是对象型数据的一种特例罢了。

54.13.2　数组元素求和

我们来看看另外一个例子，其功能是将一个输入的数组的各项数值相加求和。

```
public class ArraySum
{
        public static int f (int[] a)
        {
                int sum=0;
                for (int i=0; i<a.length; i++)
                        sum=sum+a[i];
                return sum;
        }
}
```

```
  public static int f(int[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=3, args_size=1
         0: iconst_0
         1: istore_1
         2: iconst_0
         3: istore_2
         4: iload_2
         5: aload_0
         6: arraylength
         7: if_icmpge      22
        10: iload_1
        11: aload_0
        12: iload_2
        13: iaload
        14: iadd
        15: istore_1
        16: iinc          2, 1
        19: goto          4
        22: iload_1
        23: ireturn
```

在这个成员函数的存储空间里，外来数组的引用指针存储于 LVA 的 0 号存储槽里，而局部变量 sum 则存储在 LVA 的 1 号存储槽里。

54.13.3　输入变量为数组的主函数 main()

下面展示的是一个单参数的 main()函数。这个外来参数是一个字符串。

```
public class UseArgument
{
        public static void main(String[] args)
        {
                System.out.print("Hi, ");
                System.out.print(args[1]);
                System.out.println(". How are you?");
        }
}
```

第 0 个参数是程序名（就像在 C/C++等中的一样），因此程序员指定的第一个参数存放于第一个参数槽。

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=1, args_size=1
         0: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc            #3                  // String Hi,
         5: invokevirtual  #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
```

```
 8: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iconst_1
13: aaload
14: invokevirtual  #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
17: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
20: ldc            #5                  // String . How are you?
22: invokevirtual  #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
25: return
```

偏移量为 11 的指令 aload_0 加载了 LVA（局部变量数组）的第 0 个存储单元。而函数 main()的唯一一个指定参数则通过偏移量为 12 和 13 的 iconst_1 和 aaload 指令的作用是获取数组的第一个元素的引用指针（也就是索引号为 0 的元素首地址）。偏移量为 14 的指令通过 TOS 向被调用方函数传递字符串的引用指针，这个引用指针就是此后 println 方法的输入变量。

54.13.4 预设初始值的数组

```
class Month
{
        public static String[] months =
        {
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
                "July",
                "August",
                "September",
                "October",
                "November",
                "December"
        };

        public String get_month (int i)
        {
                return months[i];
        };
}
```

我们在这里列举一个有预设值的数组，它的元素是字符串型的，其值是从一月到十二月的英文单词，分别是 January、February、March、April、May、June、July、August、September、October、November 和 December。

函数 get_month 的功能比较简单，它是输入一个整型的数，从而能在数组的对应位置输出相应月份的字符串。

```
public java.lang.String get_month(int);
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
       0: getstatic      #2                  // Field months:[Ljava/lang/String;
       3: iload_1
       4: aaload
       5: areturn
```

aaload 指令从栈里 POP 出数组的引用指针和元素的索引编号，并将指定元素推送入栈。在 Java 的概念里，字符串是对象型数据。在操作对象型数据（确切的说是引用指针）时应当使用带有 a 前缀的指令。同理，后面的 areturn 指令从侧面印证了返回指是字符串对象的引用指针。

另外一个问题是：数组 months[]是如何被初始化的呢？也就是说，这个数组的初始值 1 到 12 月份的英文字符串是如何被相应地赋值到数组的相关位置的？

```
static {};
  flags: ACC_STATIC
  Code:
    stack=4, locals=0, args_size=0
       0: bipush        12
       2: anewarray     #3                  // class java/lang/String
       5: dup
       6: iconst_0
       7: ldc           #4                  // String January
       9: aastore
      10: dup
      11: iconst_1
      12: ldc           #5                  // String February
      14: aastore
      15: dup
      16: iconst_2
      17: ldc           #6                  // String March
      19: aastore
      20: dup
      21: iconst_3
      22: ldc           #7                  // String April
      24: aastore
      25: dup
      26: iconst_4
      27: ldc           #8                  // String May
      29: aastore
      30: dup
      31: iconst_5
      32: ldc           #9                  // String June
      34: aastore
      35: dup
      36: bipush        6
      38: ldc           #10                 // String July
      40: aastore
      41: dup
      42: bipush        7
      44: ldc           #11                 // String August
      46: aastore
      47: dup
      48: bipush        8
      50: ldc           #12                 // String September
      52: aastore
      53: dup
      54: bipush        9
      56: ldc           #13                 // String October
      58: aastore
      59: dup
      60: bipush        10
      62: ldc           #14                 // String November
      64: aastore
      65: dup
      66: bipush        11
      68: ldc           #15                 // String December
      70: aastore
      71: putstatic     #2                  // Field months:[Ljava/lang/String;
      74: return
```

指令 anewarray 负责创建一个指定大小的数组，并将对象的引用指针推送入栈（字母 a 表示返回值为引用指针）。anewarry 指令的操作数声明了目标数组的数据类型，在上面的指令里这个操作数是 java/lang/String。在此之前的 bipush 12 则设置了数组的大小（这个值会被 anewarrary 指令 pop 出栈），而这个大小正好是一年的月份总数。后面出现的 dup 指令是在栈计算机领域非常著名的栈顶复制指令（在 Forth 等基于堆栈的编程语言里都有这条指令）。它将复制数组的引用指针。这是因为 aastore 指令会从栈顶 pop 出引用指针，而后面的 aastore 指令还需要再次读取该引用指针。显而易见的是，Java 编译器认为在存储数组元素时分配 dup 指令比分配 getstatic 指令更为稳妥，否则它也不会一口气派发了 12 个 dup 指令。

aastore 指令从 TOS 里依次提取（即 POP）元素值、数组下标和数组的引用指针，并将指定值存储到指定的数组元素里。

最后的 putstatic 指令将栈顶的数据出栈并把它存储到常量解析池的#2 号位置。因此它把新建数组的引用指针保存到了整个实例的第二个字段，也就是给 months 字段赋值。

54.13.5 可变参数函数

可变参数函数利用了数组的数据结构。

```
        public static void f(int... values)
        {
                for (int i=0; i<values.length; i++)
                        System.out.println(values[i]);
        }

        public static void main(String[] args)
        {
                f (1,2,3,4,5);
        }
public static void f(int...);
  flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
  Code:
    stack=3, locals=2, args_size=1
       0: iconst_0
       1: istore_1
       2: iload_1
       3: aload_0
       4: arraylength
       5: if_icmpge      23
       8: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      11: aload_0
      12: iload_1
      13: iaload
      14: invokevirtual  #3                  // Method java/io/PrintStream.println:(I)V
      17: iinc           1, 1
      20: goto           2
      23: return
```

在 f()函数中，偏移量为 3 的 aload_0 指令提取了整数数组的指针。此后的指令依次提取数组大小等信息。

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=1, args_size=1
       0: iconst_5
       1: newarray       int
       3: dup
       4: iconst_0
       5: iconst_1
       6: iastore
       7: dup
       8: iconst_1
       9: iconst_2
      10: iastore
      11: dup
      12: iconst_2
      13: iconst_3
      14: iastore
      15: dup
      16: iconst_3
      17: iconst_4
      18: iastore
      19: dup
```

```
        20: iconst_4
        21: iconst_5
        22: iastore
        23: invokestatic  #4                  // Method f:([I]V
        26: return
```

main()函数通过 newarray 指令构造了一个数组，接着填充这个数组，随后调用了 f()函数。

虽然 newarray 属于某种构造函数，但是在 main()结束之后整个数组没有被析构函数释放。实际上 Java 没有析构函数。JVM 具有自动的垃圾回收机制。

另外，当函数 main()退出后，数组对象的值其实是未消失的。其实在 JAVA 环境中就没有清除这个指令，原因是 JAVA 的内存机制会自动清理不用内存的功能，当然是在其认为必要时进行。

系统自带的 format()方法又是如何处理可变参数的呢？它把输入参数分为了字符串对象和数组型对象两大部分：

```
public PrintStream format(String format, Object... args)
```

参考链接：http://docs.oracle.com/javase/tutorial/java/data/numberformat.html。

我们再来看看下面的例子：

```
public static void main(String[] args)
{
        int i=123;
        double d=123.456;
        System.out.format("int: %d double: %f.%n", i, d);
}
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=7, locals=4, args_size=1
       0: bipush        123
       2: istore_1
       3: ldc2_w        #2                   // double 123.456d
       6: dstore_2
       7: getstatic     #4                   // Field java/lang/System.out:Ljava/io/PrintStream;
      10: ldc           #5                   // String int: %d double: %f.%n
      12: iconst_2
      13: anewarray     #6                   // class java/lang/Object
      16: dup
      17: iconst_0
      18: iload_1
      19: invokestatic  #7                   // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      22: aastore
      23: dup
      24: iconst_1
      25: dload_2
      26: invokestatic  #8                   // Method java/lang/Double.valueOf:(D)Ljava/lang/Double;
      29: aastore
      30: invokevirtual #9                   // Method java/io/PrintStream.format:(Ljava/lang/↙
↳ String;[Ljava/lang/Object;]Ljava/io/PrintStream;
      33: pop
      34: return
```

可见，int 型数据和 double 型数据首先经由各自的 valueOf 方法处理、返回相应的数据值。format()方法的输入值应当为 Object 型实例。而 Integer 和 Double 类是超类 Object 的子类，所以这种实例可以作为 format()函数参数里的数组元素。另外一方面，所有数组都是均质的，也就是说它不能保存不同类型的数据元素，因此 int 和 double 类的数据类型不可能是超类数组以外任何类型数组的数据元素。

偏移量为 13 的指令构造了一个 Object 型的数组实例，而偏移量为 22、29 的指令，分别把整型 Integer 对象，和双精度 Double 型对象添加到超类对象的数组里。

整个程序的倒数第二行指令 pop、即清除了栈顶 TOS 中的元素数值，因此在执行最后一个指令 return 时，

该方法的数据栈已经被彻底释放（又称作平衡/balanced）。

54.13.6 二维数组

在 Java 中，二维数组其实就是一个存储着另一维度数组引用指针的一维数组。

```
public static void main(String[] args)
{
        int[][] a = new int[5][10];
        a[1][2]=3;
}
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
       0: iconst_5
       1: bipush        10
       3: multianewarray  #2, 2              // class "[[I"
       7: astore_1
       8: aload_1
       9: iconst_1
      10: aaload
      11: iconst_2
      12: iconst_3
      13: iastore
      14: return
```

为了展示效果，我们在这里创建一个大小为 10×5 的整型二维数组，采用的指令是 new int[5][10]。

Java 采用 multinewarray 指令构造多维数组。本例先通过 iconst_5 和 bipush 指令将各纬度的长度值推送入栈，再使用 multinewarray 指令声明数据类型（常量解析池#2）和数组维度(2)。

偏移量为 9、10 的 iconst_1 和 aaload 指令用于加载第 1 行的引用指针。偏移量为 11 的 iconst_2 指令则声明了指定列。偏移量为 12 的指令明确该元素的取值。偏移量为 13 的 iastore 指令最终完成元素赋值。

二维数组的读取操作又是如何实现的呢？

```
public static int get12 (int[][] in)
{
        return in[1][2];
}
```

```
public static int get12(int[][]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: aload_0
       1: iconst_1
       2: aaload
       3: iconst_2
       4: iaload
       5: ireturn
```

从这个程序我们可以看到：偏移量为 2 的 aaload 指令读入了指定行的引用指针，而偏移量为 3 的 iconst_2 指令声明了列编号。最终 iaload 指令读取了指定元素的数值。

54.13.7 三维数组

三维数组可视为存储了二维数组引用指针的一维数组。

```
public static void main(String[] args)
{
        int[][][] a = new int[5][10][15];
```

```
        a[1][2][3]=4;

        get_elem(a);
    }
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
       0: iconst_5
       1: bipush        10
       3: bipush        15
       5: multianewarray  #2, 3              // class "[[[I"
       9: astore_1
      10: aload_1
      11: iconst_1
      12: aaload
      13: iconst_2
      14: aaload
      15: iconst_3
      16: iconst_4
      17: iastore
      18: aload_1
      19: invokestatic   #3                 // Method get_elem:([[[I]I
      22: pop
      23: return
```

我们这里举的例子中，三个维度的数值分别是 5、10 以及 15。

它需要使用两次 aaload 指令才能找到最后一维数组的引用指针。

```
    public static int get_elem (int[][][] a)
    {
        return a[1][2][3];
    }
public static int get_elem(int[][][]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: aload_0
       1: iconst_1
       2: aaload
       3: iconst_2
       4: aaload
       5: iconst_3
       6: iaload
       7: ireturn
```

54.13.8　小结

在 Java 中，是否可能发生缓冲区溢出的情况？不可能。Java 数组的数据实例存储了数组长度的明确信息，数组操作会作边界检查。一旦发生上标/下标溢出问题，运行环境就会进行异常处理。

Java 和 C/C++的多维数组在底层结构上存在显著的区别，因此 JAVA 不太适合用进行大规模科学计算。

54.14　字符串

54.14.1　第一个例子

Java 的字符串和数组都是同等对象，因此它们的构造过程没有什么区别。

```
        public static void main(String[] args)
        {
                System.out.println("What is your name?");
                String input = System.console().readLine();
                System.out.println("Hello, "+input);
        }
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String What is your name?
         5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: invokestatic  #5                  // Method java/lang/System.console:()Ljava/io/Console;
        11: invokevirtual #6                  // Method java/io/Console.readLine:()Ljava/lang/String;
        14: astore_1
        15: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        18: new           #7                  // class java/lang/StringBuilder
        21: dup
        22: invokespecial #8                  // Method java/lang/StringBuilder."<init>":()V
        25: ldc           #9                  // String Hello,
        27: invokevirtual #10                 // Method java/lang/StringBuilder.append:(Ljava/
    ↳ lang/String;)Ljava/lang/StringBuilder;
        30: aload_1
        31: invokevirtual #10                 // Method java/lang/StringBuilder.append:(Ljava/
    ↳ lang/String;)Ljava/lang/StringBuilder;
        34: invokevirtual #11                 // Method java/lang/StringBuilder.toString:()
    ↳ Ljava/lang/String;
        37: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang
    ↳ /String;)V
        40: return
```

我们这里举的例子是交互式的，具体功能是能根据用户输入的用户名，显示一句问候，作为回显结果。

偏移量为 11 的指令调用了 readLine 函数。其返回值，即用户输入的字符串的引用指针，最后通过栈顶 TOS 返回。偏移量为 14 的指令将字符串的引用指针存储在 LVA 的第一个存储单元中。偏移量为 30 的指令再次加载了用户输入字符串的引用指针，在 StringBuilder 类的实例中与字符串（Hello, ）连接为新的字符串。最后偏移量为 37 的 invokevirtual 指令调用了 println 方法，显示最终的字符串。

54.14.2 第二个例子

另外一个例子是：

```
public class strings
{
        public static char test (String a)
        {
                return a.charAt(3);
        };

        public static String concat (String a, String b)
        {
                return a+b;
        }
}
  public static char test(java.lang.String);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: iconst_3
         2: invokevirtual #2                  // Method java/lang/String.charAt:(I)C
         5: ireturn
```

编译器会利用 StringBuilder 类来连接字符串：

```
public static java.lang.String concat(java.lang.String, java.lang.String);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=2
       0: new           #3                  // class java/lang/StringBuilder
       3: dup
       4: invokespecial  #4                 // Method java/lang/StringBuilder."<init>":()V
       7: aload_0
       8: invokevirtual  #5                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
↳ lang/String;)Ljava/lang/StringBuilder;
      11: aload_1
      12: invokevirtual  #5                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
↳ lang/String;)Ljava/lang/StringBuilder;
      15: invokevirtual  #6                 // Method java/lang/StringBuilder.toString:() ↙
↳ Ljava/lang/String;
      18: areturn
```

我们再看一个将字符串和整型数连接在一起的例子：

```
public static void main(String[] args)
{
        String s="Hello!";
        int n=123;
        System.out.println("s=" + s + " n=" + n);
}
```

这里同样调用了 StringBuilder 类的 append 方法连接字符串，再通过 println 函数显示最终的字符串。

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
       0: ldc           #2                  // String Hello!
       2: astore_1
       3: bipush        123
       5: istore_2
       6: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
       9: new           #4                  // class java/lang/StringBuilder
      12: dup
      13: invokespecial  #5                 // Method java/lang/StringBuilder."<init>":()V
      16: ldc           #6                  // String s=
      18: invokevirtual  #7                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
↳ lang/String;)Ljava/lang/StringBuilder;
      21: aload_1
      22: invokevirtual  #7                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
↳ lang/String;)Ljava/lang/StringBuilder;
      25: ldc           #8                  // String n=
      27: invokevirtual  #7                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
↳ lang/String;)Ljava/lang/StringBuilder;
      30: iload_2
      31: invokevirtual  #9                 // Method java/lang/StringBuilder.append:(I)Ljava ↙
↳ /lang/StringBuilder;
      34: invokevirtual  #10                // Method java/lang/StringBuilder.toString:() ↙
↳ Ljava/lang/String;
      37: invokevirtual  #11                // Method java/io/PrintStream.println:(Ljava/lang ↙
↳ /String;)V
      40: return
```

54.15 异常处理

让我们再来回顾一下 54.13.4 节中已经讲到的例子，那是一个关于月份显示的程序实例。显然如果输入数组的数值小于 0 或者大于 11 的话，都会触发异常处理函数。

指令清单 54.10 IncorrectMonthException.java（不正确的月份显示例外）

```
public class IncorrectMonthException extends Exception
{
        private int index;

        public IncorrectMonthException(int index)
        {
                this.index = index;
        }
        public int getIndex()
        {
                return index;
        }
}
```

指令清单 54.11 Month2.java（月份 2）

```
class Month2
{
        public static String[] months =
        {
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
                "July",
                "August",
                "September",
                "October",
                "November",
                "December"
        };

        public static String get_month (int i) throws IncorrectMonthException
        {
                if (i<0 || i>11)
                        throw new IncorrectMonthException(i);
                return months[i];
        };

        public static void main (String[] args)
        {
                try
                {
                        System.out.println(get_month(100));
                }
                catch(IncorrectMonthException e)
                {
                        System.out.println("incorrect month index: "+ e.getIndex());
                        e.printStackTrace();
                }
        };
}
```

本质上讲，IncorrectMonthException.class 只具备一个对象构造函数和一个访问器。

这个类由 Exception 继承而来，因此它首先调用 Exception 类的构造函数，接着声明了自己唯一的输入值字段。

```
  public IncorrectMonthException(int);
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
```

```
    1: invokespecial #1                  // Method java/lang/Exception."<init>":()V
    4: aload_0
    5: iload_1
    6: putfield      #2                  // Field index:I
    9: return
```

而 getIndex()就是一个访问器/accessor。它通过 aload_0 指令从 LVA 的第 0 个存储槽获取 IncorrectMonthException 的 this 指针，再通过 getfield 指令从对象实例里提取整数值。

```
public int getIndex();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #2                  // Field index:I
       4: ireturn
```

现在让我们来看看 Month2.class 中的 get_month()。

指令清单 54.12 Month2.class

```
public static java.lang.String get_month(int) throws IncorrectMonthException;
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
       0: iload_0
       1: iflt          10
       4: iload_0
       5: bipush        11
       7: if_icmple     19
      10: new           #2                  // class IncorrectMonthException
      13: dup
      14: iload_0
      15: invokespecial #3                  // Method IncorrectMonthException."<init>":(I)V
      18: athrow
      19: getstatic     #4                  // Field months:[Ljava/lang/String;
      22: iload_0
      23: aaload
      24: areturn
```

我们来分析分析这个程序：

在偏移为 1 的 iflt 指令在栈顶值小于 1 的情况下触发跳转。“iflt”是英文 if less than 的缩写。

当 index 参数是无效值时，程序会跳转到偏移量为 10 的 new 指令，创建一个新的对象。而该对象的类型就是指令的操作数（常量解析池#2）IncorrectMonthException。接着偏移量为 15 的指令调用构造函数，并通过栈顶 TOS 传递局部变量 index。当执行到偏移量为 18 的指令处时，异常处理实例已经构造完毕， athrow 指令将从栈顶提取由上一条指令传递的异常处理方法的引用指针，并通知 JVM 系统该方法为当前类实例的异常处理函数。

此处的 athrow 指令并不返回控制流。此后的偏移量为 19 的指令开始是另外一个基本的模块，它与异常处理过程没有关系，可视为从偏移量为 7 的指令开始的领悟一个逻辑分支。

例外的句柄是如何工作的？我们来看看类 Month2.class 中的函数 main()。

指令清单 54.13 Month2.class

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
       0: getstatic     #5                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: bipush 100
       5: invokestatic  #6                  // Method get_month:(I)Ljava/lang/String;
       8: invokevirtual #7                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      11: goto 47
```

```
        14: astore_1
        15: getstatic     #5                  // Field java/lang/System.out:Ljava/io/PrintStream;
        18: new           #8                  // class java/lang/StringBuilder
        21: dup
        22: invokespecial #9                  // Method java/lang/StringBuilder."<init>":()V
        25: ldc           #10                 // String incorrect month index:
        27: invokevirtual #11                 // Method java/lang/StringBuilder.append:(Ljava/ ↙
    ↳ lang/String;)Ljava/lang/StringBuilder;
        30: aload_1
        31: invokevirtual #12                 // Method IncorrectMonthException.getIndex:()I
        34: invokevirtual #13                 // Method java/lang/StringBuilder.append:(I)Ljava ↙
    ↳ /lang/StringBuilder;
        37: invokevirtual #14                 // Method java/lang/StringBuilder.toString:() ↙
    ↳ Ljava/lang/String;
        40: invokevirtual #7                  // Method java/io/PrintStream.println:(Ljava/lang ↙
    ↳ /String;)V
        43: aload_1
        44: invokevirtual #15                 // Method IncorrectMonthException.printStackTrace ↙
:()V
        47: return
      Exception table:
        from    to target type
            0    11    14   Class IncorrectMonthException
```

自偏移量为 14 的指令开始的内容就是异常表 Exception table。在程序从偏移量 0 运行到偏移量 11（含）期间，发生的全部异常状况都会交给 IncorrectMonthException 处理。当输入值为无效值时，程序流向导至偏移量为 14 的指令。实际上，主程序在偏移量为 11 的地方就已经结束。正常情况下，程序不会执行到偏移量为 14 的指令，而且也没有任何条件转移指令或者无条件转移指令会跳转到该处。只有当程序遇到例外情况时，程序才运行到偏移量大于 11 的指令。异常处理的第一条位于偏移量 14。此处的 astore_1 会把外部传入的、异常处理实例的引用指针存储到 LVA 的第一个存储槽。在此之后，偏移量为 31 的指令将会通过这个引用指针调用异常处理实例的 getIndex()方法。此时偏移量为 30 的指令把这个引用指针已经提取出来了。异常表的其他指令都是字符串处理指令：getIndex()方法返回局部变量 index 的整数值，这个值由 toString()方法转换为字符串，再与字符串“incorrect month index:”连接，最终通过 println()和 printStackTrace()方法显示出来。在调用了 printStackTrace()之后，整个异常处理过程宣告完毕，程序恢复正常状态。虽然本例偏移量位 47 的指令是结束 main()函数的 return 指令，但是此处可以是其他的、在正常状态下需要执行的任何指令。

接下来，我们来看看 IDA 显示异常处理方法的具体方式。

指令清单 54.14　笔者计算机里某个 class 文件的异常处理方法

```
      .catch java/io/FileNotFoundException from met001_335 to met001_360\
  using met001_360
      .catch java/io/FileNotFoundException from met001_185 to met001_214\
  using met001_214
      .catch java/io/FileNotFoundException from met001_181 to met001_192\
  using met001_195
      .catch java/io/FileNotFoundException from met001_155 to met001_176\
  using met001_176
      .catch java/io/FileNotFoundException from met001_83 to met001_129 using \
met001_129
      .catch java/io/FileNotFoundException from met001_42 to met001_66 using \
met001_69
      .catch java/io/FileNotFoundException from met001_begin to met001_37\
  using met001_37
```

54.16　类

一个简单的类如下所示。

指令清单 54.15 test.java

```
public class test
{
        public static int a;
        private static int b;

        public test()
        {
            a=0;
            b=0;
        }
        public static void set_a (int input)
        {
                a=input;
        }
        public static int get_a ()
        {
                return a;
        }
        public static void set_b (int input)
        {
                b=input;
        }
        public static int get_b ()
        {
                return b;
        }
}
```

构造函数把两个变量设置为 0：

```
  public test();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial  #1                  // Method java/lang/Object."<init>":()V
         4: iconst_0
         5: putstatic      #2                  // Field a:I
         8: iconst_0
         9: putstatic      #3                  // Field b:I
        12: return
```

设置 *a*：

```
  public static void set_a(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic      #2                  // Field a:I
         4: return
```

获取 *a*：

```
  public static int get_a();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic      #2                  // Field a:I
         3: ireturn
```

设置 *b*：

```
  public static void set_b(int);
```

```
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic     #3                  // Field b:I
         4: return
```

获取 *b*：

```
  public static int get_b();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic     #3                  // Field b:I
         3: ireturn
```

在底层指令层面上，类中那些具有 public 和 private 属性的成员对象没有实质区别。但是.class 文件级别，外部指令无法直接访问其他类里的 private 属性成员。

接下来，我们演示创建对象和调用方法。

指令清单 54.16 ex1.java 程序

```
public class ex1
{
        public static void main(String[] args)
        {
                test obj=new test();
                obj.set_a (1234);
                System.out.println(obj.a);
        }
}
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: new           #2                  // class test
         3: dup
         4: invokespecial  #3                 // Method test."<init>":()V
         7: astore_1
         8: aload_1
         9: pop
        10: sipush 1234
        13: invokestatic   #4                 // Method test.set_a:(I)V
        16: getstatic     #5                  // Field java/lang/System.out:Ljava/io/PrintStream;
        19: aload_1
        20: pop
        21: getstatic     #6                  // Field test.a:I
        24: invokevirtual  #7                 // Method java/io/PrintStream.println:(I)V
        27: return
```

new 指令可以创建新的对象，但是它并没有调用构造函数（偏移量为 4 的指令调用了构造函数）。偏移量为 13 的指令调用了 set_a()方法。偏移量为 21 的 getstatic 指令访问了类的一个字段。

54.17 简单的补丁

54.17.1 第一个例子

本节通过一个简单的程序演示补丁的实现方法：

```
public class nag
{
        public static void nag_screen()
```

```
        {
                System.out.println("This program is not registered");
        };
        public static void main(String[] args)
        {
                System.out.println("Greetings from the mega-software");
                nag_screen();
        }
}
```

我们可否去掉字符串“This program is not registered”？

我们使用调试工具 IDA 加载类文件.class，如图 54.1 所示。

```
                ; Segment type: Pure code
                  .method public static nag_screen()V
                  .limit stack 2
                  .line 4
178 000 002 |       getstatic java/lang/System.out Ljava/io/PrintStream;  ; CODE XREF: main+8↓P
018 003             ldc "This program is not registered"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 5
177                 return
??? ??? ???+     .end method
??? ??? ???+
???             ; ---------------------------------------------------------------------------


                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 1
                  .line 8
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 005             ldc "Greetings from the mega-software"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
184 000 006         invokestatic nag.nag_screen()V
                  .line 10
177                 return
```

图 54.1　IDA

如图 54.2 所示，我们试图把该函数的第一个字节改为 177，即 return 的字节码。

```
                ; Segment type: Pure code
                  .method public static nag_screen()V
                  .limit stack 2
                |   .line 4

                nag_screen:                                     ; CODE XREF: main+8↓P
177                 return
000                 0 ; 0x00
002                 2 ; 0x02
018 003             ldc "This program is not registered"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 5
177                 return
??? ??? ???+     .end method
??? ??? ???+
???             ; ---------------------------------------------------------------------------
```

图 54.2　IDA

但是如此一来程序就崩溃了（运行环境为 JRE 1.7）：

```
Exception in thread "main" java.lang.VerifyError: Expecting a stack map frame
Exception Details:
  Location:
    nag.nag_screen()V @1: nop
  Reason:
    Error exists in the bytecode
  Bytecode:
    0000000: b100 0212 03b6 0004 b1

        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
        at java.lang.Class.getMethod0(Class.java:2856)
        at java.lang.Class.getMethod(Class.java:1668)
        at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)
```

也许，JVM 还存在某种与栈有关的检查机制。

好吧，我们采用一个其他的方法来“打补丁”：直接覆盖 nag()的调用指令，如图 54.3 所示。

```
                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 1
                  .line 8
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 005             ldc "Greetings from the mega software"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
000                 nop
000                 nop
000                 nop
                  .line 10
177                 return
                ; ---------------------------------------------------------------------------
```

图 54.3 IDA

0 就是 NOP 的字节码。

经过运行检验，这次的“打补丁”是成功的。

54.17.2 第二个例子

下面我们再看看另外一个简单的例子：

```
public class password
{
        public static void main(String[] args)
        {
                System.out.println("Please enter the password");
                String input = System.console().readLine();
                if (input.equals("secret"))
                        System.out.println("password is correct");
                else
                        System.out.println("password is not correct");
        }
}
```

其实现的基本思路是按照提示输入密码字符串，当输入的密码字符串为“secret”时，显示字符串密码正确（password is correct）；否则显示字符串密码不正确（password is not correct）。

将该程序调入到调试工具 IDA 中，如图 54.4 所示。

```
                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 2
                  .line 3
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 003             ldc "Please enter the password"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 4
184 000 005         invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006         invokevirtual java/io/Console.readLine()Ljava/lang/String;
076                 astore_1 ; met002_slot001
                  .line 5
043                 aload_1 ; met002_slot001
018 007             ldc "secret"
182 000 008         invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 014         ifeq met002_35
                  .line 6
178 000 002         getstatic java/lang/System.out Ljava/io/PrintStream;
018 009             ldc "password is correct"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011         goto met002_43
                  .line 8

                met002_35:                                  ; CODE XREF: main+21↑j
178 000 002         .stack use locals
                      locals Object java/lang/String
                    .end stack
                    getstatic java/lang/System.out Ljava/io/PrintStream;
018 010             ldc "password is not correct"
182 000 004         invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
```

图 54.4 IDA

关键之处是比较字符串的 ifeq 指令。这个指令其实是英文 if equal（如果相等）的缩写。实际上这个助记符不太贴切，它要是 ifz（也就是如果 TOS 是零）就更加确切了。也就是说，如果栈顶 TOS 的值是零，它就进行跳转。在这个例子中，只有当输入的密码有误才会触发跳转（布尔“假”/False 的对应值是 0）。我们的第一个想法就是调整这个指令。在 ifeq 的字节码里，有两个字节专门封装转移目标地址的偏移量。要想把它强行改为“无条件不转移”，必须把第三个字节的改为 3（ifeq 占用 3 个字节，PC 的偏移量加 3 就是执行下一条指令）：

我们把相应指令改为如图 54.5 所示的样子 。

```
                ; Segment type: Pure code
                  .method public static main([Ljava/lang/String;)V
                  .limit stack 2
                  .limit locals 2
                  .line 3
178 000 002       getstatic java/lang/System.out Ljava/io/PrintStream;
018 003           ldc "Please enter the password"
182 000 004       invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 4
184 000 005       invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006       invokevirtual java/io/Console.readLine()Ljava/lang/String;
076               astore_1 ; met002_slot001
                  .line 5
043               aload_1 ; met002_slot001
018 007           ldc "secret"
182 000 008       invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 003       ifeq met002_24
                  .line 6

                met002_24:                                          ; CODE XREF: main+21↑j
178 000 002       getstatic java/lang/System.out Ljava/io/PrintStream;
018 009           ldc "password is correct"
182 000 004       invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011       goto met002_43
                  .line 8
178 000 002       .stack use locals
                    locals Object java/lang/String
                  .end stack
                  getstatic java/lang/System.out Ljava/io/PrintStream;
018 010           ldc "password is not correct"
182 000 004       invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
                  .line 9
```

图 54.5　IDA

结果，修改后的程序无法在 JRE 1.7 环境下正确执行。

```
Exception in thread "main" java.lang.VerifyError: Expecting a stackmap frame at branch target 24
Exception Details:
  Location:
    password.main([Ljava/lang/String;]V @21: ifeq
  Reason:
    Expected stackmap frame at this location.
  Bytecode:
    0000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
    0000010: 1207 b600 0899 0003 b200 0212 09b6 0004
    0000020: a700 0bb2 0002 120a b600 04b1
  Stackmap Table:
    append_frame(@35,Object[#20])
    same_frame(@43)

        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
        at java.lang.Class.getMethod0(Class.java:2856)
        at java.lang.Class.getMethod(Class.java:1668)
        at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)
```

但是在 JRE 1.6 版本下，如此修改的程序的确可以正常运行。

笔者也尝试了“将这个 ifeq 指令的字节码直接更换成 3 个空指令 NOP”的做法。即使是这样，修改后的程序仍然不能正常运行。大概是 JRE 1.7 的栈映射核查更为全面吧！

接下来，我们更换一种方法：把 ifeq 之前的、调用 input.equals 方法的全部指令全都替换为 NOP，把它改为如图 54.6 所示的样子。

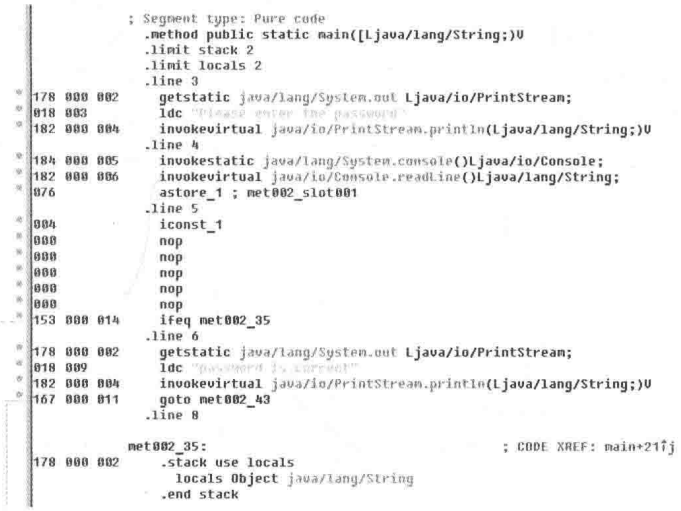


图 54.6 IDA

如此修改之后，在执行到 ifeq 指令的时候栈顶的值永远是 1，因此不会满足 ifeq 的跳转条件。试验说明，这种方法果然有效。

54.18 总结

和 C/C++语言相比，Java 语言少了些什么数据类型？

① 结构：采用类。

② 联合：采用类继承。

③ 无符号数据类型：这也直接导致了在 JAVA 下，实现密码算法比较困难。

④ 函数指针。

第五部分

在代码中发现重要而有趣的内容

目前的软件往往都很庞大，可以说，极简主义不是现代软件的突出特性。但是这并不是说当下的编程者书写的程序代码行多了，而是因为很多的库都普遍与可执行文件静态地链接在一起了。如果所有的外部库都转移至外部动态链接库 DLL 的话，那么可能情况就会有所不同（对于 C++而言，另外的一个理由是静态模板库 STL 以及其他的模板库文件）。

因此，确定函数的来源很重要，其或者是来自标准库或者通用库（类似 Boost 和 libpng），或者是与我们要找的代码相关。

为了找到我们要的代码而重新编写代码是有些荒唐。

对于一个反编译的工程师来说，一个主要的工作是快速地发现他要找的代码。

反编译工具 IDA 能让我们在字符串、字节串以及常数中搜索字符串。该工具甚至能将代码输出到 lst 或则 asm 文件中，进而可以采用 grep 和 awk 等命令处理字符串。

当想要知道某些代码是做什么用的时候，它可能采用一些开源的库（类似上面提到的 libpng），因此当你看到一些变量、常数或者字符串很熟悉时，可以用 Google 搜索一下。如果发现一个开源工程在使用时，就可以拿来做比较。这样的话，也能解决一部分问题。

例如，如果一个程序采用了 XML 文件，第一步就需要确定采用了 XML 的哪个库来做处理过程，因为通常都是采用的标准库或者通用库，而不是开发者自己来自行开发。

再举一个例子，在 SAP 6.0 的软件中，笔者曾经试图了解网络数据包是如何压缩并解压缩的。因为它是一个很大的软件，一个包括调试信息在内的 PDB 文件使用起来比较方便。通过它能发现一个函数 CsDecomprLZC 被调用，而这个函数就是对网络数据包进行解压缩的。因此通过查询 Google 就能发现这个

函数是用在了 MaxDB 中，而这就是一个 SAP 的开源项目。

查询的命令是：http://www.google.com/search?q=CsDecomprLZC。

令人吃惊的是，MaxDB 和 SAP 6.0 软件在网络数据的压缩和解压缩方面采用了相同的代码。

第 55 章　编译器产生的文件特征

55.1　Microsoft Visual C++

MSVC 的版本和其 DLL 的对应关系如下所示。

市场发行的版本	内部版本	命令行版本	能导入的 DLL 的版本	发行日期
6	6.0	12.00	msvcrt.dll, msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll, msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll, msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll, msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll, msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll, msvcp120.dll	October 17, 2013

msvcp*.dll 含有 C++相关函数，因此导入这些 DLL 文件的可执行程序很可能是 C++程序。

55.1.1　命名规则

名字通常都是以“?”开始的。

有关 MSVC 命名规则的详细介绍，请参考本书 51.1.1 节。

55.2　GCC 编译器

GCC 不仅可以编译*NIX 平台的应用程序，在 Cygwin 和 MinGW 环境下它同样可以编译面向 Win32 平台的应用程序。

55.2.1　命名规则

命名通常以符号“_Z”开始。

有关 GCC 命名规则的详细介绍，请参考本书 51.1.1 节。

55.2.2　Cygwin

GCC 在 Cygwin 环境下编译的应用程序，通常会导入 cygwin1.dll 文件。

55.2.3　MinGW

GCC 在 Cygwin 环境下编译的应用程序，可能会导入 msvcrt.dll 文件。

55.3　Intel FORTRAN

由 Intel Fortran 编译的应用程序，可能会导入以下 3 个文件：

① Libifcoremd.dll。

② Libifportmd.dll。

③ Libiomp5.dll（支持 OpenMP）。

库文件 libifcoremd.dll 定义了很多以字符串“for_”开头的函数。它就是 FORTRAN 的代表性前缀。

55.4 Watcom 以及 OpenWatcom

55.4.1 命名规则

由 Watcom 编译出来的程序，其符号名称通常以“W”字母开头。

以 W?method$_class$n__v 为例：大写字母 W 开头代表它是 Watcom 编译出来的程序，方法的名称为 method，类的名称为 class，没有参数且无返回值的 void 方法会命名为：

```
W?method$_class$n__v
```

55.5 Borland 编译器

下面所示的是 Borland Delphi 编译器以及 C++ Builder 的命名规范：

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvtl
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiitliiiiii
@TrueColorTo16BitN$qpviiiiiitliiiiii
@DIB24BitTo8BitBitmap$qpviiiiiitliiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

由 Borland 编译出来的程序，其符号名称必定以字符“@”开头，后面几个字母分别代表：类名称、方法名称以及方法的参数类型。

这些符号名称可能出现在 exe 文件的输入表、dll 文件的输出表以及调试数据等地方。

VCL 的全称是 Borland Visual Component Libraries，意思是 Borland 的可视化组件库。它们保存在 bpl 文件中，而不是保存在 dll 文件中。比如说文件 vcl50.dll 和 rtl60.dll。

由 Borland 编译出来的程序还可能会导入 BORLNDMM.DLL 文件。

55.5.1 Delphi 编程语言

通过观察我们不难看出，几乎所有的 Delphi 可执行文件在代码段的起始部分都有一个“Boolean”字符串，后面还跟着其他数据类型的类型名称。

这里列出一个非常典型的 Delphi 程序的代码段片段，它就位于 Win32 程序的 PE 头之后。

```
00000400  04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00  |..@...Boolean...|
00000410  00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65  |........@..False|
00000420  04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69  |.True.@.,.@...Wi|
00000430  64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90  |deChar..........|
00000440  44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff  |D.@...Char......|
00000450  00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69  |....X.@...Smalli|
00000460  6e 74 02 00 80 ff ff ff 7f 00 00 90 70 10 40 00  |nt..........p.@.|
00000470  01 07 49 6e 74 65 67 65 72 04 00 00 00 80 ff ff  |..Integer.......|
00000480  ff 7f 8b c0 88 10 40 00 01 04 42 79 74 65 01 00  |......@...Byte..|
00000490  00 00 00 ff 00 00 00 90 9c 10 40 00 01 04 57 6f  |..........@...Wo|
```

```
000004a0  72 64 03 00 00 00 00 ff ff 00 00 90 b0 10 40 00  |rd............@.|
000004b0  01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 00 ff  |..Cardinal......|
000004c0  ff ff ff 90 c8 10 40 00 10 05 49 6e 74 36 34 00  |......@...Int64.|
000004d0  00 00 00 00 00 00 80 ff ff ff ff ff ff ff 7f 90  |................|
000004e0  e4 10 40 00 04 08 45 78 74 65 6e 64 65 64 02 90  |..@...Extended..|
000004f0  f4 10 40 00 04 06 44 6f 75 62 6c 65 01 8d 40 00  |..@...Double..@.|
00000500  04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90  |..@...Currency..|
00000510  14 11 40 00 0a 06 73 74 72 69 6e 67 20 11 40 00  |..@...string .@.|
00000520  0b 0a 57 69 64 65 53 74 72 69 6e 67 30 11 40 00  |..WideString0.@.|
00000530  0c 07 56 61 72 69 61 6e 74 8d 40 00 40 11 40 00  |..Variant.@.@.@.|
00000540  0c 0a 4f 6c 65 56 61 72 69 61 6e 74 98 11 40 00  |..OleVariant..@.|
00000550  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000560  00 00 00 00 00 00 00 00 00 00 00 00 98 11 40 00  |..............@.|
00000570  04 00 00 00 00 00 00 00 18 4d 40 00 24 4d 40 00  |.........M@.$M@.|
00000580  28 4d 40 00 2c 4d 40 00 20 4d 40 00 68 4a 40 00  |(M@.,M@. M@.hJ@.|
00000590  84 4a 40 00 c0 4a 40 00 07 54 4f 62 6a 65 63 74  |.J@..J@..TObject|
000005a0  a4 11 40 00 07 07 54 4f 62 6a 65 63 74 98 11 40  |..@...TObject..@|
000005b0  00 00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00  |........System..|
000005c0  c4 11 40 00 0f 0a 49 49 6e 74 65 72 66 61 63 65  |..@...IInterface|
000005d0  00 00 00 00 01 00 00 00 00 00 00 00 00 c0 00 00  |................|
000005e0  00 00 00 00 46 06 53 79 73 74 65 6d 03 00 ff ff  |....F.System....|
000005f0  f4 11 40 00 0f 09 49 44 69 73 70 61 74 63 68 c0  |..@...IDispatch.|
00000600  11 40 00 01 00 04 02 00 00 00 00 00 c0 00 00 00  |.@..............|
00000610  00 00 00 46 06 53 79 73 74 65 6d 04 00 ff ff 90  |...F.System.....|
00000620  cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8  |..D$...Ql...D$..|
00000630  e9 6f 6c 00 00 83 44 24 04 f8 e9 79 6c 00 00 cc  |.ol...D$...yl...|
00000640  cc 21 12 40 00 2b 12 40 00 35 12 40 00 01 00 00  |.!.@.+.@.5.@....|
00000650  00 00 00 00 00 00 00 00 c0 00 00 00 00 00 00 00  |................|
00000660  46 41 12 40 00 08 00 00 00 00 00 00 00 8d 40 00  |FA.@..........@.|
00000670  bc 12 40 00 4d 12 40 00 00 00 00 00 00 00 00 00  |..@.M.@.........|
00000680  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000690  bc 12 40 00 0c 00 00 00 4c 11 40 00 18 4d 40 00  |..@.....L.@..M@.|
000006a0  50 7e 40 00 5c 7e 40 00 2c 4d 40 00 20 4d 40 00  |P~@.\~@.,M@. M@.|
000006b0  6c 7e 40 00 84 4a 40 00 c0 4a 40 00 11 54 49 6e  |l~@..J@..J@..TIn|
000006c0  74 65 72 66 61 63 65 64 4f 62 6a 65 63 74 8b c0  |terfacedObject..|
000006d0  d4 12 40 00 07 11 54 49 6e 74 65 72 66 61 63 65  |..@...TInterface|
000006e0  64 4f 62 6a 65 63 74 bc 12 40 00 a0 11 40 00 00  |dObject..@...@..|
000006f0  00 06 53 79 73 74 65 6d 00 00 8b c0 00 13 40 00  |..System......@.|
00000700  11 0b 54 42 6f 75 6e 64 41 72 72 61 79 04 00 00  |..TBoundArray...|
00000710  00 00 00 00 00 03 00 00 00 6c 10 40 00 06 53 79  |.........l.@..Sy|
00000720  73 74 65 6d 28 13 40 00 04 09 54 44 61 74 65 54  |stem(.@...TDateT|
00000730  69 6d 65 01 ff 25 48 e0 c4 00 8b c0 ff 25 44 e0  |ime..%H......%D.|
```

数据段（DATA）的头四个字节通常是以下三个组合中的一个任意一个：00 00 00 00、32 13 8B C0 或者 FF FF FF FF。在处理被压缩或者被加密的 Dephi 可执行文件时，这组常数就会具有指标性意义。

55.6　其他的已知 DLL 文件

- Vcomp*.dll。微软用来实现 OpenMP 的文件。

第 56 章　Win32 环境下与外部通信

在了解函数的输入和输出的情况下，我们基本可以判断出函数的具体功能。这种分析方法能够显著地节省分析时间。

如需关注文件和注册表层面的行为，使用 SysInternals 的 Process Monitor 即可，它可以给我们提供关于以上这两者的基本信息。

如需查看网络层面的通信数据，完全可以使用 Wireshark 这类软件。

然而要进一步分析行为级数据，就得深入程序内部挖掘指令层面的信息。

首先就要调查该程序调用的操作系统 API 和标准库函数。

如果目标程序由可执行文件和多个 DLL 文件构成，那么由这些 DLL 文件所提供的、可调用的函数名称就很有指标性意义。

如果我们只关心那些调用 MessageBox()、显示特定文字的指令，我们可以在程序的数据段检索文本字符，找到引用这个字符串的指令，再顺藤摸瓜地找到那些调用既定 MessageBox()函数的代码。

在分析电脑游戏时，如果可以确定特定关卡里出现的敌人总数是随机数，那么我们可以在代码中查找 rand()函数或者类似的随机数生成函数（例如梅森旋转算法），继而找到这些函数的调用指令，最终调整程序里使用随机数的那些指令。本书 75 章演示了这种分析实例。

那些电脑游戏以外的、仍然调用 rand()函数的程序就更值得关注了。令人感到吃惊的是，某些著名软件采用的数据压缩算法（加密机制）都调用了 rand()函数。有兴趣的读者可参阅 https://yurichev.com/blog/44/。

56.1　在 Windows API 中最经常使用的函数

这里列出了一些最常使用的 API 函数。需要特别说明的是，这些函数可能不是由程序源代码直接调用的。在程序调用库函数或者调用 CRT 的时候，下述函数可能会被后者间接调用。

- 注册表的操作可以通过库文件 advspi32.dll 的如下功能实现：RegEnumKeyEx、RegEnumValue、RegGetValue、RegOpenKeyEx 和 RegQueryValueEx。
- 对类似 ini 的文本文件可以通过库文件 user32.dll 的如下函数实现：GetPrivateProfileString。
- 对话窗的操作通过库文件 user32.dll 的如下函数实现：MessageBoxEx、SetDlgItemText 及 GetDlgItemText。
- 对资源的操作（可以参考本书的 68.2.8 节）通过库文件 user32.dll 的函数 LoadMenu 实现。
- 对 TCP/IP 网络的操作是通过库文件 ws2_32.dll 的如下函数实现：WSARecv 和 WSASend。
- 对文件的操作是通过库文件 kernel32.dll 的如下函数实现相应的操作：CreateFile、ReadFile、ReadFileEx、WriteFile 及 WriteFileEx 等。
- 访问 Internet 是通过库文件 wininet.dll 的 WinHttpOpen 等函数来实现相关功能的。
- 检查一个可执行文件是否含有数字签名则是通过库文件 wintrust.dll 的函数 WinVerifyTrust 等来实现的。
- 如果是动态链接的话，标准的 MSVC 库文件 msvcr*.dll 是通过以下函数实现相关操作的：assert、itoa、ltoa、open、printf、read、strcmp、atol、atoi、fopen、fread、fwrite、memcmp、rand、strlen、strstr 以及 strchr。

56.2　tracer:解析指定模块的所有函数

在调试程序时，tracer 会给目标程序设置很多 INT3 断点。虽然这种类型的断点只能运行一次，但是它

同样可以用截获特定 DLL 文件的所有函数。

一个典型的使用例子为：

```
--one-time-INT3-bp:somedll.dll!.*
```

如果要在调用所有以 xml 开头的函数之前设置 INT 3 断点，那么可以采用命令：

```
--one-time-INT3-bp:somedll.dll!xml.*
```

稍显遗憾的是，这些断点只能触发一次。

如果程序执行到函数断点且成功发生中断，tracer 就会显示该函数的调用信息，只是它只能显示一次。另外一个美中不足的地方是，tracer 不能查看被调用方函数获取的外来参数。

无论如何，tracer 的这项功能还是非常有用的。一个 DLL 文件通常会定义大量的函数。当我们知道既定程序调用了某个的 DLL 文件、想要确切知道它调用了 DLL 里的哪些函数的时候，我们就特别需要这样的一款工具。

举例来讲，我们可以使用 tracer 给 cygwin 的程序 uptime.exe 设置断点，看它调用了哪些系统函数：

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

这样一来，我们就可以看到它调用了 cygwin1.dll 的哪些库函数（虽然只会显示一次）以及调用指令的地偏移量信息：

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3))
```

第57章 字 符 串

57.1 字符串

57.1.1 C/C++中的字符串

在C/C++中，常规字符串都是以0字节结尾的ASCII字符串，因此又称ASCIIZ字符串。这是历史上硬件局限性决定的。在参考书目【Rit79】中，我们可以看到以下说明：

> I/O操作的最小单位是word而不是byte。毕竟PDP-7是一种以字为单位寻址的设备。字和字节的差异性在这方面的唯一影响就是：处理字符串的程序必须要忽略字符串中的Null字符，因为在构造字符串的时候必须使用null字节将字符串凑成偶数个字节、形成word字。

在Hiew或者FAR Manager中，下述程序的字符串在可执行文件中会如图57.1所示：

```
int main()
{
        printf ("Hello, world!\n");
};
```



图57.1 Hiew

57.1.2 Borland Delphi

如指令清单57.1所示，在Pascal及Borland Delphi编译的可执行程序中，字符串之前都会有一个声明字符串长度的8位/32位数据。

指令清单57.1 Delphi

```
CODE:00518AC8                   dd 19h
CODE:00518ACC aLoading__Plea db 'Loading... , please wait.',0

...

CODE:00518AFC                   dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

57.1.3 Unicode编码

多数人认为，所谓Unicode编码就是用两个字节/16位数据来编码一个字符的字符封装格式。实际上这是一种常见的术语理解错误。Unicode实际上是一个标准，它规定的只是将一个数字写成字符的方法，但是没有定义具体的编码方式。

而目前比较流行的编码方式有 UTF-8 和 UTF-16LE。前者广泛被利用在互联网和*NIX 系统中，而后者主要使用在 Windows 环境下。

UTF-8

UTF-8 是目前使用最广泛也最成功的字符编码方法之一。所有的拉丁字符都像 ASCII 码一样进行编码，ASCII 码表以外的字符则采用多字节来编码。因为 0 的作用不变，所以所有的标准 C 字符串函数都能正确处理包括 UTF-8 编码在内的所有字符串。

下面我们通过一个对照表来看看不同语言下的 UTF-8 的对比显示情况，采用的工具是 FAR，代码页是 437。如图 57.2 所示。

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

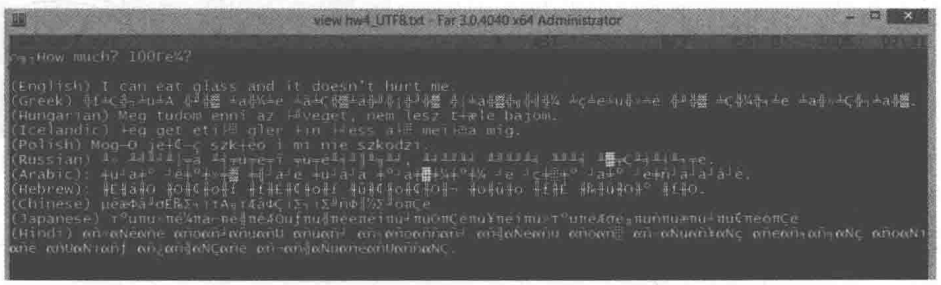


图 57.2　FAR UTF-8

从以上的对照，我们可以清楚地看到，只有英文的字符串看起来和 ASCII 表中的完全一样。匈牙利语言使用一些拉丁字符以及音节分隔标记来表示。这些符号使用多个字节来编码，我们这里采用了红色的下画线表示。从这个表，我们还可以看到爱尔兰语和波兰语也采用了同样的办法。而这个字符串对比的开始处，我们采用了一个欧元符号，它是用三个字节表示的。其余的系统与拉丁文没有关系。至少在俄语、阿拉伯语、希伯来语以及北印度语中，我们会发现其中一个字节是反复出现的，这也不奇怪：一个语言系统中的字符往往是在 Unicode 表中的相同位置处，因此它们的代码总是以系统的数字打头。

在最开始，也就是在第一个可见字符串“How much?”之前，我们会看到还有三个字节，实际上它们是字节顺序标记（Byte order mark，BOM）。BOM 声明了字符串的编码系统。

UTF-16LE

很多 Windows 系统下的 win32 函数有-A 和-W 后缀。前面这种函数用于处理常规字符串，而后面这种带有-w 的函数则是 UTF-16LE 字符串的专用函数（w 代表 wide）。

在 UTF-16 字符串的拉丁符号中，我们用工具 Hiew 或者 FAR 可以看到，这些字符都被字节 0 间隔开了，如图 57.3 所示。

程序如下所示。

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

而在 Windows NT 系统中，我们可以经常看到的显示如图 57.4 所示。



图 57.3 Hiew



图 57.4 Hiew

在 IDA 的提示信息中，严格采用双字节对单字符编码的编码方式称为 Unicode。

例如：

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

而图 57.5 所示的则是俄语的字符串，它采用的是 UTF-16LE 编码方式。



图 57.5 Hiew 工具，编码方式 UTF-16LE

我们比较容易分辨的是这些字符被星型的字符分割，而这个星型字符的 ASCII 值是 4。实际上，西里尔字母位于 Unicode 表的第 4 映射区，因此所有的西里尔字母在 UTF-16LE 中的编码范围是 0x400～0x4ff。详情请参考 https://en.wikipedia.org/wiki/Cyrillic_(Unicode_block)

再回过头来看看我们上面列出的一个显示多语言字符串的例子。图 57.6 所示的是其在 UTF-16LE 编码方式下的样子。

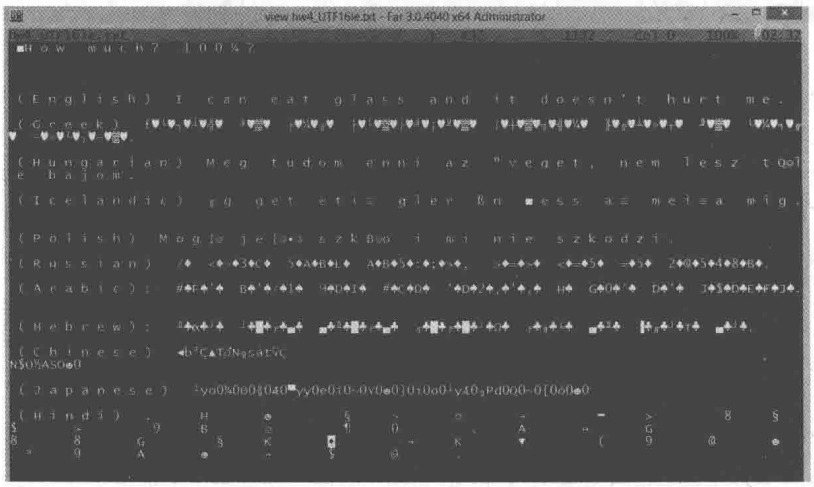


图 57.6 采用工具软件 FAR，编码格式为 UTF-16LE

从以上图中我们可以看到，字节分割符 BOM 位于文件的开头，而所有的拉丁字母都用字节零来分割。一些带读音分割标志的字符（主要是匈牙利语和爱尔兰语）也采用红色的下划线标出来了。

57.1.4 Base64

Base64 编码十分流行，是把二进制数据转换为文本字符串的常用标准。本质上说，这种算法用 4 个可显示字符封装 3 个二进制字节。它的字符集包括 26 个拉丁字母（含大小写）、0～9 共 10 个数字、加号“+”

及反斜杠“/”，总共 64 个字符。

Base64 编码的一个显著特征是它通常（但不一定）以 1 到 2 个等号“=”为结尾。

比如说以下两个 Base64 编码：

```
AVjbbVSVfcUMulxvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=
WVjbbVSVfcUMulxvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuQ==
```

可以肯定的是，等号“=”绝不会出现在 Base64 编码字符串的中间。

57.2 错误/调试信息

对于逆向分析来说，程序中的调试信息都很重要。在某种程度上，调试信息能报告程序正在运行的状态。调试信息通常会由 printf()一类的函数显示出来，或者被输出到日志文件中。但是在 release/发行版、而非 debug/测试版的软件中，即使有关指令调用了相关调试函数、也不会有任何实质性的输出内容。如果调试信息的转储数据中含有局部变量或者全局变量的信息，那么逆向工程人员就算赚到了——我们至少知道了变量的名称。比如说，我们可以通过转储信息确定 Oracle RDBMS 有一个函数叫做 ksdwrt()。

内容可自然解释的字符串通常是逆向分析的重点。IDA 反编译器可以显示出字符串的调用方函数和调用指令。数量掌握这种分析之后，您可能会找到一些有趣的东西（可以参考 https://yurichev.com/blog/32/）。

错误信息有时也很重要。Oracle RDBMS 构造了一系列函数专门处理错误信息。有兴趣的读者可以访问 https://yurichev.com/blog/43/了解详细信息。

多数情况下，我们能够迅速判断出汇报错误的函数以及引发它们报错的具体条件。有意思的是，正因如此，一些注重版权保护的程序会刻意在程序出错的时候临时调整错误信息或错误代码。毕竟，开发人员不会希望别人马上就能摸清他的防盗版措施。

本书的 78.2 节就演示了一个对错误信息加密的程序。

57.3 可疑的魔数字符串

一些经常被用在后门程序中的魔数字符串看起来就很可疑。比如说，我们注意到一个关于 TP-Link WR740 家用路由器存在后门的报道（参见 http://sekurak.pl/tp-link-httptftp-backdoor/）。只有当他人访问下述 URL 时，才会触发这个后门：

```
http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html。
```

事实上，字符串 userRpmNatDebugRpm26525557 必定存在于固件中的某个文件。然而在这个后门东窗事发之前，Google 搜索不到任何信息，当然这个后门被曝光后的情况完全相反。像这种后门类的字符串，查遍 RFC 资料你也找不到它。再怎么调整字节序，它也不会和科学算法沾边。它也绝不是错误信息或者调试信息。因此，尽快地定位类似这个的可疑字符串是一个好主意。

字符串通常采用 Base64 编码。因此把文件中的字符串全都进行解码处理，再扫一眼就知道哪个文件含有这个字符串了。

更准确来讲，这种隐藏后门的办法被称为“不公开即安全（security through obscurity）”，也就是见光死。

第 58 章　调用宏 assert()（中文称为断言）

一般来讲，assert()宏在可执行文件中保留了源代码的文件名、行数以及执行条件。

最有价值的信息是 assert()宏的执行条件。我们可以通过它们来推断出变量名或者结构体的字段名称。另外一个有用的信息是文件名，通过它我们可以推断出源代码是采用什么语言编写的。同时我们还可能通过文件名来识别出其是否采用了知名的开放源代码库。

指令清单 58.1　调用 assert()宏的例子

```
. text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D  cmp  edx, 1
.text:107D4B30  jz   short loc_107D4B4A
.text:107D4B32  push 1ECh
.text:107D4B37  push offset aWrite_c ; "write.c"
.text:107D4B3C  push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41  call ds:_assert

...

.text:107D52CA  mov  edx, [ebp-4]
.text:107D52CD  and  edx, 3
.text:107D52D0  test edx, edx
.text:107D52D2  jz   short loc_107D52E9
.text:107D52D4  push 58h
.text:107D52D6  push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB  push offset aN30      ; "(n & 3) == 0"
.text:107D52E0  call ds:_assert

...

.text:107D6759  mov  cx, [eax+6]
.text:107D675D  cmp  ecx, 0Ch
.text:107D6760  jle  short loc_107D677A
.text:107D6762  push 2D8h
.text:107D6767  push offset aLzw_c    ; "lzw.c"
.text:107D676C  push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771  call ds:_assert
```

最简单的分析方法就是用 google 来搜索条件和文件名。搜索结果显示这与开源的库文件有关。比如说，搜索字符串“sp->lzw_nbits <= BITS_MAX”，我们就它与压缩算法 LZW 的开源代码相关。

第 59 章　常　　数

人类在现实生活中喜欢使用整数。编程人员也是人，他们同样喜欢用 10、100、1000 这样的整数。

熟悉的反编译工程师都会明白，与之对应的十六进制数的关系是：10=0xA;100=0x64;1000=0x3E8 而 10000=0x2710。然而，在二进制层面这些数字都不算太“整”。

从二进制层面来看，更为常用的整数则是 0xAAAAAAAA（1010101010101010）和 0x55555555（0101010101010101）这类特征明显的常量。以常数 0x55AA 为例：引导扇区、主引导扇区 MBR 以及 IBM 兼容扩展卡的 ROM（只读存储单元）等关键数据都会使用这个常量。

一些算法，特别是某些加密算法，常常使用某些特殊的常数，如果我们使用调试工具 IDA 就能很容易发现这一点。

以 MD5 算法为例，其内部变量初始值分别是：

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

也就是说，如果某段代码连续出现了上述四个常量，那么这段代码很可能就与 MD5 的算法相关。

而另外的例子则是 CRC16/32 的算法，其预置的常数表经常如下所示。

指令清单 59.1　Linux/Lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        …
```

如需详细了解 CRC32 算法，请参考本书的第 37 章。

59.1　魔数

很多文件在文件头使用特定的魔数来表示其文件格式，这些魔数可以是一个字节或者多个字节的组合（https://en.wikipedia.org/wiki/Magic_number_(programming)）。

比如，我们熟知的，所有的 Win32 以及 MS-DOS 格式的可执行文件的开始处总是“MZ”这两个字符。

而在标准的 MIDI 文件则必须以“MThd”这 4 个字符开头。因此，那些需要使用 MIDI 文件的程序，基本上都会检测目标文件的头 4 个字符是不是“MThd”。如果用程序来表示，则可能是下面这个样子的：（注意：buf 是内存缓冲区的起始地址）

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

当然，数据比较函数同样可以用来验证文件头中的魔数。常用的函数有：比较内存块的 memcmp()函数，或者 CMPSB 一类的比较指令（可以参考本书附录 A.6.3）。

一旦发现某个程序开始检测其他文件的魔数标识，我们就可以确信它已经加载了某种类型的目标文件。不止如此，我们还可以解读出文件操作的缓冲区以及它使用缓冲区的方式方法等信息。

59.1.1 动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）

网络协议同样使用了魔术。比如，DHCP 协议的网络数据就会用到魔数 0x63538263—这个魔数叫做 magic cookie。所有符合 DHCP 协议的数据包都必须使用这个魔数。如果我们找到了这个魔数，我们就可能确定此处代码可能用于实现 DHCP 协议。不仅如此，能接受 DHCP 包的程序都必须验证这个魔数，也就是与它做比对。

比如说，我们以 Windows 7 操作系统（64 位操作系统）中的文件 dhcpcore.dll 文件为例，我们可以在这个文件中搜索这个魔数。结果发现了 2 次，出现在两个函数中，其名称分别是 DhcpExtractOptionsForValidation() 和 and DhcpExtractFullOptions()。

指令清单 59.2 dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h                  ; DATA XREF:↙
    ↳ DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h                  ; DATA XREF:↙
    ↳ DhcpExtractFullOptions+97
```

以下列出在该程序文件中是如何使用这个魔数的：

指令清单 59.3 dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

指令清单 59.4 dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

59.2 寻找常数

在单个文件里搜索常数时，可以使用 IDA 的搜索功能。其快捷键是 ALT-B 或 ALT-I。

在海量文件检索常量时，可以使用笔者开发的小工具—binary grep (https://github.com/yurichev/bgrep)。它同样可以检索非可执行文件中的特定信息。

第 60 章　检索关键指令

如果一个程序使用了为数不多的 FPU（Float Point Unit，浮点运算单元）指令，那么我们可以采用人工排查的方式把它们逐一筛选出来。

以 Microsoft 的表格工具软件 Excel 为例。我们可能要研究它对录入公式的处理方法，例如除法操作。

首先要用 IDA 加载 Office 2010 里的 excel.exe（本章以版本号为 14.0.4756.1000 的 excel 为例），然后生成完整的指令清单并将其保存为后缀名为.lst 的文本文件。接下来，我们就可以用终端指令检索指令清单中的全部 FDIV 指令（Floationg Point Divide，浮点数除法）。严格说来，使用 grep 不能检索出全部的浮点数除法运算指令。如果除数是常量，那么编译器就不太可能分配 FDIV 指令。当然这种特例不在我们的研究范围之内。

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

总共有 144 个匹配结果。

然后，我们在 Excel 里输入计算公式“=(1/3)”，并检查每个指令的运行结果。

在调试器（或 tracer）里逐一排查除法运算指令以后，我们幸运地发现在第 14 个 FDIV 指令就是我们要找的指令：

```
.text:3011E919 DC 33                                    fdiv qword ptr [ebx]
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

此时，第一个参数（被除数）被保存在 ST(0)中，而除数则保存在[EBX]中。

FDIV 后面的 FSTP 指令，将结果写入内存：

```
.text:3011E91B DD 1E                                    fstp qword ptr [esi]
```

在 FSTP 指令处设置断点，我们能看到下述运算结果：

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

为了验证我们的结果，我们做一个简单而有趣的试验：在内存中对具体的内存单元直接进行修改，以便得到“直接运算可能不能生成的效果”。

比如：

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
```

```
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

这样的话，我们在 Excel 的相关单元就会发现数字 666。从这一点可以看出，我们找到了正确的指令位置。

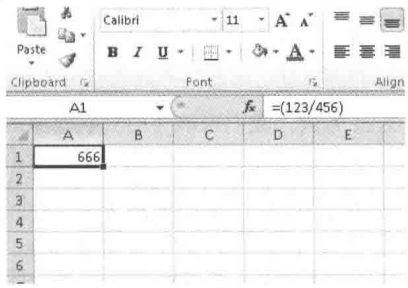


图 60.1 通过修改内存发现效果

如果我们调试的是 64 位的同版本 Excel 程序，我们只会找到 12 条 FDIV 指令。而我们关注的运算指令是第 3 个 FDIV 指令：

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

大概是在编译 64 位的 Excel 程序时，编译器使用 SSE 指令替代了 float 和 double 型数据的除法运算指令。其中，SSE 指令集的 DIVSD 指令就出现了 268 次。

第 61 章　可疑的代码模型

61.1　XOR 异或指令

像 XOR op,op 或者 XOR EAX,EAX 这样的指令通常用来将某个寄存器清零。只有当 XOR 指令的两个操作数不同的时候，它才进行真正的“异或”运算。这种实际意义上的异或运算，在常规应用程序很少见到，反而在加密算法中比较常见，即使是业余人员编写的程序也是如此。而如果 XOR 的第二个操作数是一个很大的数，那么这个程序就显得特别可疑。这种情况往往意味着它会进行加密或者解密、校验和等类型的复杂计算。

需要说明的是，18.3 节介绍的编译器采用的“百灵鸟”技术同样会生成大量的 XOR 指令。不过这些 XOR 指令和加/解密等科学运算无关。

我们可以利用下述 AWK 脚本处理 IDA 生成的指令清单文件（.lst），检索其中的 xor 指令：

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp")↙
    ↳ { print $1, $2, tmp, ",", $4 } }' filename.lst
```

值得注意的是，这种类型的脚本也适用于适配不正确的反汇编代码（可以参考本书第 49 章）。

61.2　手写汇编代码

当前的编译器都不会分配循环指令 LOOP 和位移循环指令 RCL。从另外一方面来讲，喜欢直接手写汇编语言的编程人员非常熟悉这些指令。因此，如果你看到了这些指令的话，那么这部分代码十有八九由编程人员手工编写而来。本书附录 A.6 都将这些指令添加了“(M)”标记。

手写的汇编程序很少会具备完整的函数开头和函数结尾。

通常来说，人工手写的程序没有固定的参数传递方法。

举一个例子：Windows 2003 操作系统的内核文件 ntoskrnl.exe。

```
MultiplyTest proc near                  ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                             ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                          ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near                  ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
             mov     eax, 417A000h
             mul     ecx
             cmp     edx, 2
             stc
             jnz short locret_62057F
             cmp eax, 0FE7A000h
             stc
             jnz short locret_62057F
```

```
            clc
locret_62057F:                          ; CODE XREF: Multiply+10
                                        ; Multiply+18
            retn
Multiply    endp
```

实际上，只要查看 WRK[①] v1.2 的源代码就会发现：这部分指令确实来自于手写的汇编语言源文件 WRK-v1.2\base\ntos\ke\i386\cpu.asm。

① WRK 是 Windows Research Kernel（Windows 研究内核）的缩写。

第 62 章　魔数与程序调试

通常来说，逆向工程的主要目标理解程序处理数据的具体方法。这些数据可能来自于某个文件或者来自于网络通信，不过数据的出处无关紧要。手工跟踪一个值往往是一项非常劳神费力的事情。为了完成这个任务，一个最简单的方法是使用自己的特有魔数，虽然这个办法不是百分之百可靠。

在某种意义上讲，魔数的作用和 X 光的造影剂十分相似：在病人的血液里注入造影剂之后，医生就可以增强 X 射线的观察效果、清晰地观察病人身体的内部情况。借助造影剂的作用，医生可以在 X 光机下清晰地观察肾脏里血液的循环过程，从而更为准确地判断脏器是否存在结石或者肿瘤等问题。

魔数要尽量“打眼”。在观测 32 位数据时，我们可以将魔数设置为 0x0BADF00D （BADFOOD）、或者 0x11101979（某人的生日，例如“1979 年 11 月 10 日”）。我们可以将这样的 4 字节数值写入到我们要调查的程序之中。

接着，我们可以利用 tracer 的代码覆盖率模式（code coverage/cc 模式）跟踪程序，再利用 grep 或者直接搜索文本文件（跟踪结果的文本文件），我们就能够很容易地发现这些值的调用点及处理方法。

在代码覆盖率模式（cc 模式）下，使用 tracer 跟踪程序并生成与 grep 兼容的输出（grepable）文件。其结果大致如下所示：

```
0x150bf66 (_kziaia+0x14), e=          1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=          1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=          1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=          1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=          1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

我们同样可以在网络数据包中构造魔数。使用魔数的关键在于：要使用不会重复的且不会在程序里出现的标志性数据。

除了跟踪器 tracer 之外，MS-DOS 模拟器（DosBox）也可以用来观测魔数。在 heavydebug（重度调试）模式下，DosBox 可以把执行每条指令时的寄存器状态输出到纯文本文件[①]。因此，我们同样可以利用魔数来调试 DOS 程序。

① 关于 MS-DOS 模拟器（DosBox）的特性可以查询以下博客：https://yurichev.com/blog/55/。

第 63 章　其他的事情

63.1　总则

逆向工程人员应当尽可能地以编程人员的角度来分析问题。要理解他们的观点、并且时常扪心自问：如果自己是编程人员的话，自己会如何设计程序。

63.2　C++

在分析 C++的 class 时，本书 51.1.5 节中讲到的 RTTI 数据可能就是分析的重点。

63.3　部分二进制文件的特征

在十六进制的编辑器里，16 位/32 位/64 位数据数组的特征十分明显。本节以一个很简单的 MIPS 程序为例。前文介绍过：每个 MIPS 指令都是整齐的 32 位（即 4 字节）指令。当然 ARM 模式或者 ARM64 模式的程序也有这个特点。因此，在形式上这些程序的指令都构成了某种 32 位数组。

图 63.1 所示的屏幕截图，就表现出这种数组特征。笔者添加了 3 条红色的垂直线，以突出这种现象：

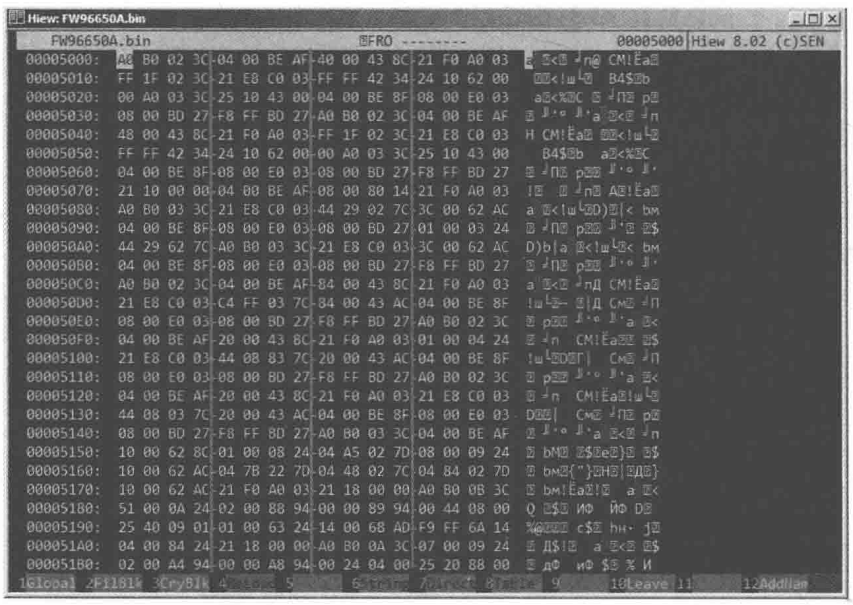


图 63.1　Hiew：一个非常典型的 MIPS 代码

此外，本书第 86 章介绍了另外一个典型的例子。

63.4　内存“快照”对比

这种内存“快照”的对比技术可以凸显出内存中变化的数据，过去就常见于 8 位游戏的作弊领域。

如果您在老式的 8 位游戏机（这些机器通常内存不大，而游戏本身占用的内存更少）上启动一个游戏，您就可能看到部分游戏数据—例如，现在装有 100 颗子弹。这时你就可以将所有的内存空间做一个镜像。把镜像存为文件后，您可以先开一枪，这时子弹的剩余数量会显示为 99 发。这个时候，你再对整个内存空间做另外一个影像。我们对这两个内存影像进行比对，肯定某个字节的数值从 100 降为 99。

考虑到这些 8 位的游戏程序一般都是汇编语言程序，而且各变量都是全局变量。因此基于以上的对比结果，我们就能分析出是哪个内存地址存放着子弹的数量。有了这个关键的突破点，我们接着就可以在游戏程序的汇编代码（需要进行反汇编处理）中搜索那些指令引用了这个地址，就不难发现对子弹总数进行递减运算的那条指令。然后，我们把相关指令换成 NOP，就能保证子弹总数保持 100 不变。

在 8 位游戏机上运行的游戏程序，总是会被加载在固定的内存地址。此外，同款游戏的发行版本也比较固定（一旦热销也就不会再进行什么升级了）。因此铁杆玩家们都知道改写哪些地址就能“黑掉”这些游戏了。他们通常会用 BASIC 语言的单字节赋值指令 POKE 直接向已知地址写入特定字节。甚至在一些杂志中，常常会出现一些关于 8 位机的 POKE 指令列表。

与此类似，修改游戏分数排行榜也比较容易，而且它不仅仅适用于 8 位游戏。首先，记录一下自己的游戏得分并将存盘文件备份出来。当排行榜的总分出现变化时，再备份一次存盘文件。接下来，我们可以用 DOS 系统自带的二进制文件比较工具 FC 直接比对两个文件（存盘文件是二进制文件）。两个文件肯定有几个字节发生了变化，直接修改这些数值就可以调整游戏得分。然而，现在的游戏开发团队通常都非常清楚这些伎俩，因此可能会开发一些程序来应对这些措施。

其他类似的例子可以在本书的第 85 章中找到。

63.4.1　Windows 注册表

在安装一个程序以后，我们就能比对安装前后 Windows 的注册表。这是一个非常流行的方法，可以用来确认特定程序使用了哪些注册表键值。这也许就是“Windows 注册表清理”程序颇为流行的原因。

63.4.2　瞬变比较器 Blink-comparator

在介绍了文件比对和内存快照比对的方法之后，笔者不禁想起了瞬变比较器/Blink-comparator（https://en.wikipedia.org/wiki/Blink_comparator）：过去，它曾是天文学家观测天体移动的一种相片比对设备。它能够快速地切换不同时期拍摄的照片，以便天文学家通过肉眼快速地发现两图之间的区别。

实际上，正是借助于瞬变比较器，人们才在 1930 年发现了冥王星。

第六部分

操作系统相关



第 64 章　参数的传递方法（调用规范）

64.1　cdecl [C Declaration 的缩写]

这是C/C++语言最常使用的参数传递方法。

调用方函数逆序向被调用方函数传递参数：以“最后（右侧）的参数、倒数第二……至第一个参数”的顺序传递参数。被调用方函数退出后，应由调用方函数调整栈指针 ESP、将栈恢复成调用其他函数之前的原始状态。

指令清单 64.1　cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

64.2　stdcall [Standard Call 的缩写]

这种调用方式与上面提到的 cdecl 规范类似，只是有一点不同：被调用方函数在返回之前会执行“RET x”指令还原参数栈，而不会使用单纯的“RET”指令直接返回。这里的 *x* 的数值的计算方式是：*x*=参数个数*指针的大小（注意：指针的大小在 x86 结构中的值是 4，而在 x64 中是 8）。这样调用方函数本身就不会调整栈指针，因此调用方函数不会因此使用类似于“add esp,x”这样的指令。

指令清单 64.2　stdcall

```
push arg3
push arg2
push arg1
call function

function:
... do something ...
ret 12
```

此类约定在 Win32 的标准库文件中十分常见。然而因为 Win64 系统遵循调用约定的是 Win64 规范，所以 Win64 的库文件里不会出现 stdcall 的标志性操作指令。

以本书 8.1 节中的函数为例。我们给其中的函数增加__stdcall 限定符即可强制它使用 stdcall 调用约定：

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

它的编译结果与本书 8.2 节类似，但是最后的指令从 RET 变成了 RET 12。在遵循 stdcall 约定之后，栈指针 SP 不再由调用方函数更新了。

这样一来，我们就可以通过函数尾部的 RETN N 指令直观地推算出外来参数的个数。计算方法就是：N 除以 4。

指令清单 64.3 MSVC 2010

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_c$ = 16                                                ; size = 4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12                                      ; 0000000cH
_f2@12  ENDP

; ...
        push    3
        push    2
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

64.2.1 带有可变参数的函数

在 C/C++语言的标准函数中，printf()一类的函数可能是仅存的几个带有可变参数的函数。借助这类函数的帮助，我们能比较容易地观察出 cdecl 和 stdcall 调用规范之间的区别。我们首先假定编译器知道 printf()函数的参数总数。然而，在 Windows 环境下，printf()属于预先编译好的库函数，直接由文件 MSVCRT.DLL 提供。所以，我们无法通过从它的函数代码入手获悉可变参数的处理方式；但是另一方面，我们知道它肯定会处理格式化字符串。如果 printf()函数当真采取了 stdcall 规范、根据格式化字符串统计变参的数量并且在函数尾声恢复栈指针，那么这种局面就十分危险了：万一程序员打错了几个字母，程序就会崩溃。由此可知，对于那些带有可变参数的函数而言，cdecl 规范要比 stdcall 规范更好一些。

64.3 fastcall

这个调用约定优先使用寄存器传递参数，无法通过寄存器传递的参数则通过栈传递给被调用方函数。因为 fastcall 约定在内存栈方面的访问压力比较小，所以在早期的 CPU 平台上遵循 fastcall 规范的程序会比遵循 stdcall 和 cdecl 规范的程序性能更高。但是在现在的、更为复杂的 CPU 平台上，fastcall 规范的性能优势就不那么明显了。

这种调用约定没有统一的技术规范，不同的编译器有着各自不同的实现方法。因此在使用这种约定时，我们需要特别小心：如果用两个不同的编译器编译出来了 2 个相互调用的、遵循 fastcall 规范的 DLL 库，那么这种互相调用的访问操作基本都会出现故障。

不管是 MSVC 还是 GCC 都使用 ECX 和 EDX 传递第一个和第二个参数，用栈传递其余的参数。此外，应由被调用方函数调整栈指针、把参数栈恢复到调用之前的初始状态（这一点与 stdcall 类似）。

指令清单 64.4 fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
```

```
.. do something ..
ret 4
```

我们还是以本书 8.1 节中的例子来说明，把它稍微变化一下，增加一个修饰符号：

```
int __fastcall f3 (int a, int b, int c)
{
        return a*b+c;
};
```

编译完成后，我们看到的结果如下所示。

指令清单 64.5　MSVC 2010/OB0

```
_c$ = 8                                                 ; size = 4
@f3@12 PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

从以上程序我们可以看到，函数调用采用了指令 RET N 带一个操作数的方式返回 SP 堆栈指针。由此可以判断，调用方函数通过栈传递了多少个外部参数。

64.3.1　GCC regparm

从某种意义上来讲，GCC regparm 由 fastcall 进化而来。这种规范允许编程人员通过编译选项“-mregparm”设置通过寄存器传递的参数总数（最大值为 3）。换句话说，这种规范最多可通过 3 个寄存器，即 EAX、EDX 和 ECX 传递函数参数。

当然，如果“-mregparm”的值小于 3，那么就不会全面使用这三个寄存器。

这种约定要求调用方函数在调用过程结束以后调整栈指针，将参数栈恢复到其初始状态。

有关案例请参阅本书的 19.1.1 节。

64.3.2　Watcom/OpenWatcom

这被称为“寄存器调用规范”。头四个参数由寄存器 EAX、EDX、EBX 和 ECX 传递，其余的所有参数都则通过堆栈传递。在使用这种调用约定时，函数必须其函数名前添加标识符“__watcom”，与其他采用不同调用规范的函数区分开来。

64.4　thiscall

这是一种方便 C++类成员调用 this 指针而特别设定的调用规范。

MSVC 使用 ECX 寄存器传递 this 指针。

而 GCC 则把 this 指针作为被调用方函数的第一个参数传递。在汇编层面，这个指针显而易见：所有的

类函数都比源代码多出来一个参数。

有关详情，请参阅本书的 51.1.1 节。

64.5 64 位下的 x86

64.5.1 Windows x64

64 位环境下的参数传递方法在某种程度上与 fastcall 函数比较类似：头四个参数由寄存器 RCX、RDX、R8 和 R9 传递，而其余的参数都通过栈来传递。调用方函数必须预留 32 个字节或者 4 个 64 位的存储空间，以便被调用方函数保存头四个参数。小型函数可以仅凭寄存器就获取所有参数，而大型函数就可能需要保存这些传递参数的寄存器，把它们腾挪出来供后续指令调用。

调用方函数负责调整栈指针到其初始状态。

此外，Windows x86-64 系统的 DLL 文件也采用了这种调用规范。也就是说，虽然 Win32 系统 API 遵循的是 stdcall 规范，但是 Win64 系统遵循的是 Win64 规范、不再使用 stdcall 规范。

以下述程序为例：

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

指令清单 64.6 MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72                                 ; 00000048H

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72                                 ; 00000048H
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
```

```
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72                                 ; 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT:$SG2937
        call    printf

        add     rsp, 72                                 ; 00000048H
        ret     0
f1      ENDP
```

从以上的程序，我们可以很清楚地看到 7 个参数的传递过程。程序通过寄存器传递前 4 个参数、再通过栈传递了其余 3 个参数。f1()函数通过序言部分的指令，把传递参数的四个寄存器的外来值存储到“暂存空间/scratch space”里——这正是暂存空间的正确用法。编译器无法实现确定缺少了这 4 个寄存器之后，后续代码是否还有足够的寄存器可用，所以会把这 4 个寄存器的数据保管起来，以方便掉配这 4 个寄存器。Win64 调用规范约定：应由调用方函数分配暂存空间给被调用方函数使用。

指令清单 64.7　优化的 MSVC 2012/0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72                                 ; 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72                                 ; 00000048H
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72                                 ; 00000048H

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
```

```
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72                         ; 00000048H
        ret     0
main    ENDP
```

即使我们启用编译器的优化选项编译上述代码，编译器仍然会生成基本相同的指令；只是它不再分配上面提到的“零散空间”，因为已经不需要它了。

另外，我们也看到：在启用优化编译选项之后，MSVC 2012 将是一 LEA 指令（请参阅附录 A.6.2）进行数值传递。笔者并不确定它分配的这种指令是否能够提升运行效率，或许真有这种作用吧。

另外，本书的 74.1 节介绍了另外一个 Win64 调用约定的程序。有兴趣的读者可去看一下。

64 位下的 Windows：在 C/C++下传递 this 指针

C/C++编译器会使用 RCX 寄存器传递类对象的 this 指针、用 RDX 寄存器传递函数所需的第一个参数。关于这个方面的例子，可以查看本书的 51.1.1 节。

64.5.2 64 位下的 Linux

64 位 Linux 程序传递参数的方法和 64 位 Windows 程序的传递方法几乎相同。区别在于，64 位 Linux 程序使用 6 个寄存器（RDI、RSI、RDX、RCX、R8、R9）传递前几项参数，而 64 位 Windows 则只利用 4 个寄存器传递参数。另外，64 位 Linux 程序没有上面提到的“零散空间”这种概念。如果被调用方函数的寄存器数量紧张，它就可以用栈存储外来参数，把相关寄存器腾出来使用。

指令清单 64.8 优化的 GCC 4.7.3

```
.LC0:
        .string "%d %d %d %d %d %d %d\n"
f1:
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main:
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

在上述指令操作 EAX 寄存器的时候，它只把数据写到了 RAX 寄存器的低 32 位（即 EAX）而没有直接操作整个 64 位 RAX 寄存器。这是因为：在操作寄存器的低 32 位的时候，该寄存器的高 32 位会被自动清零。或许，这只是把 x86 代码移植到 x86-64 平台时的偷懒做法。

64.6　单/双精度数型返回值

除了 Win64 规范以外的所有的调用规范都规定：当返回值为单/双精度浮点型数据时，被调用方函数应当通过 FPU 寄存器 ST(0)传递返回值。而 Win64 规范规定：被调用方函数应当通过 XMM0 寄存器的低 32 位（float）或低 64 位寄存器（double）返回单/双精度浮点型数据。

64.7　修改参数

C/C++和其他语言的编程人员可能都曾问过这样一个问题：如果被调用方函数修改了外来参数的值，将会发生什么情况？答案十分简单：外来参数都是通过栈传递的，因此被调用方函数修改的是栈里的数据。在被调用方函数退出以后，调用方函数不会再访问自己传递给别人的参数。

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

指令清单 64.9　MSVC 2012

```
_a$ = 8                                          ; size = 4
_b$ = 12                                         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

由此可见，只要这些参数不是 C++的引用指针/references（本书的 51.3 节）也不是数据指针，那么被调用方函数可以随便操作外部传来的参数。

理论上讲，在被调用方函数结束以后，调用方函数能够获取被调用方函数修改过的参数，对它们加以进一步利用。然而实际上我们只能在手写的汇编指令中遇到这种情况，C/C++语言并不支持这种访问方法。

64.8　指针型函数参数

我们可以给函数参数分配一个指针，把它调配给其他函数：

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
        modify_a (&a);
        printf ("%d\n", a);
};
```

要不是看了下面的汇编代码，我们一时还很难理解这段程序是如何运行的。

指令清单 64.10 MSVC 2010 的优化

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
        lea     eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
        push    eax                       ; and pass it to modify_a()
        call    _modify_a
        mov     ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
        push    ecx                       ; and pass it to printf()
        push    OFFSET $SG2796 ; '%d'
        call    _printf
        add     esp, 12
        ret     0
_f      ENDP
```

变量 a 的地址通过栈传递给了一个函数，然后这个地址又被传递给了另外一个函数。第一个函数修改了变量 a 的值，而后 printf()函数获取到了这个修改后的变量值。

细心的读者可能会问：使用一种直接通过寄存器传递参数的调用约定，又会是什么情况呢？

即便真地使用了这种调用约定，还会有阴影空间（Shadow Space）的问题。传递的数值将会从寄存器保存到了本地栈的阴影空间里，然后以地址的形式传递给其他函数。

指令清单 64.11 优化的 MSVC 2012（64 位）

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
f       PROC
        mov     DWORD PTR [rsp+8], ecx ; save input value in Shadow Space
        sub     rsp, 40
        lea     rcx, QWORD PTR a$[rsp] ; get address of value and pass it to modify_a()
        call    modify_a
        mov     edx, DWORD PTR a$[rsp] ; reload value from Shadow Space and pass it to printf()
        lea     rcx, OFFSET FLAT:$SG2994 ; '%d'
        call    printf
        add     rsp, 40
        ret     0
f       ENDP
```

GCC 也将输入的数值保存到本地栈。

指令清单 64.12 优化的 GCC 4.9.1（64 位）

```
.LC0:
        .string "%d\n"
f:
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi   ; store input value to the local stack
        lea     rdi, [rsp+12]             ; take an address of the value and pass it to modify_a()
        call    modify_a
```

```
        mov     edx, DWORD PTR [rsp+12]   ; reload value from the local stack and pass it to printf()
        mov     esi, OFFSET FLAT:.LC0     ; '%d'
        mov     edi, 1
        xor     eax, eax
        call    __printf_chk
        add     rsp, 24
        ret
```

ARM64 下的 GCC 也已同样的方式传递参数。只是在这个平台上，这个空间被称为“寄存器（内容）保存区（Register Save Area）”。

指令清单 64.13　优化的 GCC 4.9.1 ARM64

```
f:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0          ; setup FP
        add     x1, x29, 32         ; calculate address of variable in Register Save Area
        str     w0, [x1,-4]!        ; store input value there
        mov     x0, x1              ; pass address of variable to the modify_a()
        bl      modify_a
        ldr     w1, [x29,28]        ; load value from the variable and pass it to printf()
        adrp    x0, .LC0 ; '%d'
        add     x0, x0, :lo12:.LC0
        bl      printf              ; call printf()
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "%d\n"
```

另外，与阴影空间（Shadow Space）的有关话题还可以参阅本书的 46.1.2 节。

第 65 章　线程本地存储 TLS

线程本地存储（Thread Local Storage，TLS）是一种在线程内部共享数据的数据交换区域。每个线程都可以在这个区域保存它们要在内部共享的数据。一个比较知名的例子是 C 语言的全局变量 errno。对于 errno 这类的全局变量来说，如果多线程进程的某一个线程对其进行了修改，那么这个变量就会影响到其他所有的线程。这显然和实际需求相悖，因此全局变量 errno 必须保存在 TLS 中。

为解决这个矛盾，C++ 11 标准新增了一个限定符 thread_local。它能将指定变量和特定的线程联系起来。由它限定的变量能被初始化，并且会被保存在 TLS 中。

指令清单 65.1　C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

接下来，我们使用 MinGW GCC 4.8.1 编译它，不要用 MSVC 2012 进行编译。

在分析可执行文件的 PE 头之后就会发现，变量 tmp 被分配到 TLS 专用的数据保存区域了。

65.1　线性同余发生器（改）

本书第 20 章展示的随机数生成函数其实有一个瑕疵：在多线程并发运行时，它是不安全的。原因在于：它有一个内部的变量，它可能会同时被不同的线程读取或者修改。

65.1.1　Win32 系统

未初始化的 TLS 数据

我们可以给这种全局变量增加限定符__declspec(thread)，这样它就能被分配到 TLS 中。请注意下述代码的第 9 行。

```
 1 #include <stdint.h>
 2 #include <windows.h>
 3 #include <winnt.h>
 4
 5 // from the Numerical Recipes book:
 6 #define RNG_a 1664525
 7 #define RNG_c 1013904223
 8
 9 __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
```

```
15
16 int my_rand ()
17 {
18         rand_state=rand_state*RNG_a;
19         rand_state=rand_state+RNG_c;
20         return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25         my_srand(0x12345678);
26         printf ("%d\n", my_rand());
27 };
```

用 MSVC 2013 编译上述程序，再用 Hiew 打开最后生成的可执行文件。我们可以看到这种文件的 PE 部分出现了全新的 TLS 段：

指令清单 65.2　优化的 MSVC 2013 x86

```
_TLS SEGMENT
_rand_state DD 01H DUP (?)
_TLS ENDS

_DATA SEGMENT
$SG84851 DB      '%d', 0aH, 00H
_DATA ENDS
_TEXT SEGMENT

_init$ = 8                                              ; size = 4
_my_srand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state[ecx], eax
        ret     0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        imul    eax, DWORD PTR _rand_state[ecx], 1664525
        add     eax, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR _rand_state[ecx], eax
        and     eax, 32767                              ; 00007fffH
        ret     0
_my_rand ENDP

_TEXT ENDS
```

参数 rand_state 现在是位于 TLS 段中。此后每个线程都会拥有各自的 rand_state 。这里表示的是如何寻址：从 FS:2Ch 调用线程信息块（Thread Information Block，TIB）的地址。如果需要，再增加一个额外的索引，最后再计算 TLS 段的地址。

这种程序的线程可以通过 ECX 寄存器访问各自的 rand_state 变量，因为该变量在各个线程的地址不再相同。

FS 段选择器并不陌生。它其实就是 TIB 的专用指针，用于提高线程数据的加载速度。

GS 段选择器是 Win64 程序使用的额外的索引寄存器。

在下面这个程序里，TLS 的地址是 0x58。

指令清单 65.3 优化的 MSVC 2013（64 位）

```
_TLS    SEGMENT
rand_state DD   01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG85451 DB     '%d', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT

init$ = 8
my_srand PROC
        mov     edx, DWORD PTR _tls_index
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     r8d, OFFSET FLAT:rand_state
        mov     rax, QWORD PTR [rax+rdx*8]
        mov     DWORD PTR [r8+rax], ecx
        ret     0
my_srand ENDP

my_rand PROC
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     ecx, DWORD PTR _tls_index
        mov     edx, OFFSET FLAT:rand_state
        mov     rcx, QWORD PTR [rax+rcx*8]
        imul    eax, DWORD PTR [rcx+rdx], 1664525        ; 0019660dH
        add     eax, 1013904223                          ; 3c6ef35fH
        mov     DWORD PTR [rcx+rdx], eax
        and     eax, 32767                               ; 00007fffH
        ret     0
my_rand ENDP

_TEXT ENDS
```

初始化的 TLS 数据

编程人员通常会想给变量 rand_state 设置一个固定的初始值，以防后期忘记对它进行初始化。如下述代码第 9 行所示，我们对它进行初始化赋值。

```
 1 #include <stdint.h>
 2 #include <windows.h>
 3 #include <winnt.h>
 4
 5 // from the Numerical Recipes book:
 6 #define RNG_a 1664525
 7 #define RNG_c 1013904223
 8
 9 __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15
16 int my_rand ()
17 {
```

```
18          rand_state=rand_state*RNG_a;
19          rand_state=rand_state+RNG_c;
20          return rand_state & 0x7fff;
21  }
22
23  int main()
24  {
25          printf ("%d\n", my_rand());
26  };
```

以上的代码看起来并没有什么不同，但是在 IDA 下我们可以发现：

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls              segment para public 'DATA' use32
.tls:00404000                   assume cs:_tls
.tls:00404000                   ;org 404000h
.tls:00404000 TlsStart          db    0                 ; DATA XREF: .rdata:TlsDirectory
.tls:00404001                   db    0
.tls:00404002                   db    0
.tls:00404003                   db    0
.tls:00404004                   dd 1234
.tls:00404008 TlsEnd            db    0                 ; DATA XREF: .rdata:TlsEnd_ptr
...
```

我们要关注的是这里显示的数 1234。每当启动新的线程时，它都会分配一个新的 TLS 段。此后，包括 1234 在内的所有数据都会被复制到新建都 TLS 段。

一个典型的应用场景是：

- 启动线程 A，系统同期创建该线程专用的 TLS，并将变量 rand_state 赋值为 1234。
- 此后，线程 A 多次调用 my_rand()函数，rand_state 变量的值不再会是初始值 1234。
- 另行启动线程 B，系统同期创建该线程专用的 TLS，变量 rand_state 也会被赋值为 1234。也就是说，在线程 A 和线程 B 里，同名变量会有不同的值。

TLS 回调

如果 TLS 中的变量必须填充为某些数据，并且是以某些不寻常的方式进行的话，该怎么办呢？比如说，我们有以下的这个任务：编程人员忘记调用 my_srand()函数来初始化随机数发生器（PRNG），而随机数发生器只有在正确初始化之后才会生成真实意义上的随机数，而不是 1234 这样的固定值。在这种情况下，可以采用 TLS 回调。

在采用这种 hack 之后，本例代码的可移植性（通用性）就变差了。毕竟，本例只是一个演示性质的敲门砖而已。它只是构造一个在进程/线程启动前就被系统调用的回调函数（tls_callback()）。这个回调函数用 GetTickCount()函数的返回值来初始化随机数发生器（PRNG）。

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
```

```
{
        my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        // rand_state is already initialized at the moment (using GetTickCount())
        printf ("%d\n", my_rand());
};
```

我们在 IDA 中查看一下，代码如下所示。

指令清单 65.4 优化的 MSVC 2013

```
.text:00401020 TlsCallback_0    proc near               ; DATA XREF: .rdata:TlsCallbacks
.text:00401020                  call    ds:GetTickCount
.text:00401026                  push    eax
.text:00401027                  call    my_srand
.text:0040102C                  pop     ecx
.text:0040102D                  retn    0Ch
.text:0040102D TlsCallback_0    endp

...

.rdata:004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr

...

.rdata:00402118 TlsDirectory    dd offset TlsStart
.rdata:0040211C TlsEnd_ptr      dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr    dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

在解压过程中使用 TLS 回调函数，可起到混淆视听的作用。经验不足的分析人员通常会感到晕头转向，无法在分析原始入口/OEP 之前就已经运行的回调函数。

65.1.2 Linux 系统

我们来看看在 GCC 下是如何定义线程本地的全局变量的：

```
__thread uint32_t rand_state=1234;
```

当然，这不是标准的 C/C++修饰符，而是 GCC 的专用修饰符。

GS 段选择器也常用于 TLS 寻址，但是 Linux 的实现方法和 Windows 略有不同。

指令清单 65.5 x86 下的优化 GCC 4.8.1

```
.text:08048460 my_srand         proc near
.text:08048460
.text:08048460 arg_0            = dword ptr 4
.text:08048460
```

```
.text:08048460                  mov     eax, [esp+arg_0]
.text:08048464                  mov     gs:0FFFFFFFCh, eax
.text:0804846A                  retn
.text:0804846A my_srand         endp

.text:08048470 my_rand          proc near
.text:08048470                  imul    eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B                  add     eax, 3C6EF35Fh
.text:08048480                  mov     gs:0FFFFFFFCh, eax
.text:08048486                  and     eax, 7FFFh
.text:0804848B                  retn
.text:0804848B my_rand          endp
```

更多的信息可以查看参考书目 Dre13。

第 66 章　系统调用（syscall-s）

我们都知道，所有在操作系统中运行的进程可以分成两类：一类进程对具有硬件的全部访问权限，运行于内核空间（kernel space）；另一类进程不能直接访问硬件地址，运行于用户空间（user space）。

操作系统的内核以及常规的驱动程序都运行于内核空间。普通的应用程序通常运行于用户空间。

具体来说，Linux 的内核运行于空间；而 Glibc（底层 API）则运行于空间。

这两种空间的隔离措施对于操作系统的安全性至关重要。若没有隔离措施，所有程序都可以干扰其他进程甚至破坏操作系统的内核。另一方面来看，即使实现了两种空间的隔离措施，一旦运行于内核空间的驱动程序发生错误、或者是操作系统的内核组件存在问题，整个内核照样会崩溃甚至发生 BSOD（Black Screen of Death）的“蓝天白云”故障。

虽然 x86 CPU 引入了特权等级的概念，将进程权限分为 ring0～ring3，但是 Linux 和 Windows 只使用了其中的 2 个级别控制进程权限：ring0（内核空间）和 ring3（用户空间）。

由操作系统提供的系统调用（syscall）构成了 ring0 和 ring 3 之间的访问机制。可以说，系统调用就是操作系统为应用程序提供的应用编程接口 API。

而在 Windows NT 环境下，系统调用表位于系统服务分配表 SSDT（System Service Dispatch Table）。

计算机病毒以及 shellcode 大多都会利用系统调用。这是因为系统库函数的寻址过程十分麻烦，而直接调用系统调用却相对简单。虽然系统调用的访问过程并不麻烦，但是由于系统调用本身属于底层 API，因此直接使用系统调用的程序也不好写。另外需要注意的是：系统调用的总数由操作系统和系统版本两个因素共同决定的。

66.1　Linux

Linux 程序通常通过 80 号中断/INT 80 调用系统调用。在调用系统调用时，程序应当通过 EAX 寄存器指定被调用函数的编号，再使用其他寄存器声明系统调用的参数。

指令清单 66.1　使用两次系统调用 syscall 的简单例子

```
section .text
global _start

_start:
        mov     edx,len ; buffer len
        mov     ecx,msg ; buffer
        mov     ebx,1   ; file descriptor. 1 is for stdout
        mov     eax,4   ; syscall number. 4 is for sys_write
        int     0x80

        mov     eax,1   ; syscall number. 4 is for sys_exit
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

编译指令如下所示。

```
nasm -f elf32 1.s
ld 1.o
```

完整的 Linux 系统调用列表可以参考：http://go.yurichev.com/17319。

如需截获或追踪 Linux 系统调用的访问过程，可使用本书 71 章介绍的 strace 程序。

66.2　Windows

Windows 程序可通过 0x2e 号中断/int 0x2e、或 x86 专用指令 SYSENTER 访问系统调用。

这里使用的中断数是 0x2e，或者采用 x86 下的特殊指令 SYSENTER。

完整的 Windows 系统调用列表可以参考：http://go.yurichev.com/17320。

进一步的阅读，可以参阅 Piotr Bania 编写的 Windows Syscall Shellcode 一书：http://go.yurichev.com/17321。

第 67 章 Linux

67.1 位置无关的代码

在分析 Linux 共享库文件（扩展名是 so）时，我们经常会遇到具有下述特征的指令代码：

指令清单 67.1 x86 下的 libc-2.17.so

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near             ; CODE XREF: sub_17350+3
.text:0012D5E3                                             ; sub_173CC+4 ...
.text:0012D5E3                 mov    ebx, [esp+0]
.text:0012D5E6                 retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near                   ; CODE XREF: tmpfile+73

...

.text:000576C0                 push   ebp
.text:000576C1                 mov    ecx, large gs:0
.text:000576C8                 push   edi
.text:000576C9                 push   esi
.text:000576CA                 push   ebx
.text:000576CB                 call   __x86_get_pc_thunk_bx
.text:000576D0                 add    ebx, 157930h
.text:000576D6                 sub    esp, 9Ch

...

.text:000579F0                 lea    eax, (a__gen_tempname - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6                 mov    [esp+0ACh+var_A0], eax
.text:000579FA                 lea    eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/posix/tempname.c"
.text:00057A00                 mov    [esp+0ACh+var_A8], eax
.text:00057A04                 lea    eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid ↙
    ↳ KIND in __gen_tempname\""
.text:00057A0A                 mov    [esp+0ACh+var_A4], 14Ah
.text:00057A12                 mov    [esp+0ACh+var_AC], eax
.text:00057A15                 call   __assert_fail
```

所有字符串指针都被一些常数修正过，并且相关函数都在开始的几条指令里重新调整 EBX 中的值。这类指令称作“位置无关的代码 PIC（Position Independent Code）”。因为进程或对象会被操作系统的链接器加载到任意内存地址，所以代码里的指令无法直接确定（hardcoded）绝对内存地址。

PIC 在早期的计算机系统中非常关键，而目前在没有虚拟内存支持的嵌入式系统中就更为重要了。对于那些没有采用虚拟内存技术的嵌入式设备来说，所有进程都存放于一个连续的内存块中。PIC 至今仍然用于*NIX 系统的共享目标库，因为不同进程可能会链接同一个共享库。库文件只会被操作系统加载一次。当应用程序调用库时，直接把共享的地址复制过来。这种情况下，调用库函数的进程会被加载到不同地址，而库文件的加载地址却固定不变。这些因素决定，共享库的库函数不得使用绝对地址（至少对内部对象而言），否则就不能被多个进程同时调用。

我们来做一个简单的试验：

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

我们在 GCC 4.7.3 下编译一下，然后使用 IDA 打开编译后的.so 文件。

编译的命令行为：

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

指令清单 67.2 GCC 4.7.3

```
.text:00000440                  public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near          ; CODE XREF: _init_proc+4
.text:00000440                                          ; deregister_tm_clones+4 ...
.text:00000440                  mov     ebx, [esp+0]
.text:00000443                  retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                  public f1
.text:00000570 f1               proc near
.text:00000570
.text:00000570 var_1C           = dword ptr -1Ch
.text:00000570 var_18           = dword ptr -18h
.text:00000570 var_14           = dword ptr -14h
.text:00000570 var_8            = dword ptr -8
.text:00000570 var_4            = dword ptr -4
.text:00000570 arg_0            = dword ptr 4
.text:00000570
.text:00000570                  sub     esp, 1Ch
.text:00000573                  mov     [esp+1Ch+var_8], ebx
.text:00000577                  call    __x86_get_pc_thunk_bx
.text:0000057C                  add     ebx, 1A84h
.text:00000582                  mov     [esp+1Ch+var_4], esi
.text:00000586                  mov     eax, ds:(global_variable_ptr - 2000h)[ebx]
.text:0000058C                  mov     esi, [eax]
.text:0000058E                  lea     eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text:00000594                  add     esi, [esp+1Ch+arg_0]
.text:00000598                  mov     [esp+1Ch+var_18], eax
.text:0000059C                  mov     [esp+1Ch+var_1C], 1
.text:000005A3                  mov     [esp+1Ch+var_14], esi
.text:000005A7                  call    __printf_chk
.text:000005AC                  mov     eax, esi
.text:000005AE                  mov     ebx, [esp+1Ch+var_8]
.text:000005B2                  mov     esi, [esp+1Ch+var_4]
.text:000005B6                  add     esp, 1Ch
.text:000005B9                  retn
.text:000005B9 f1               endp
```

上述代码的关键在于：每个函数都在启动之后调整了字符串"returning %d\n" 和 global_variable 的指针。__x86_get_pc_thunk_bx()函数通过 EBX 返回一个指向自身的指针。而位于其后（偏移量 0x57C 处）的指令再次对 ebx 进行了修正。这是一种获取 PC 指针（EIP）的取巧办法。

常数 0x1A84 是函数的启始地址与“全局偏移表（global offset table）GOT”和“过程链接表（Procedure Linkage Table）PLT”之间的地址差。在可执行文件中，GOT、PLT 都有各自的相应段（section）。全局变量 global_variable 的指针正好位于全局偏移量表 GOT 之后。为了便于我们理解偏移量和各表之间的关系，IDA 对显示的偏

移量进行了某种挑战。这部分的原始指令实际上是：

```
.text:00000577                call __x86_get_pc_thunk_bx
.text:0000057C                add ebx, 1A84h
.text:00000582                mov [esp+1Ch+var_4], esi
.text:00000586                mov eax, [ebx-0Ch]
.text:0000058C                mov esi, [eax]
.text:0000058E                lea eax, [ebx-1A30h]
```

EBX 寄存器存储着 GOT PLT 的指针（相应 section 的启始地址）。因此在计算全局变量 global_variable 的指针时（该指针保存在 GOT 中），必须从 EBX 减去地址差，即常数 0xC。同理，在计算“returning %d\n”的字符串指针时，必须从 EBX 减去 0x1A30。

实际上，AMD64 的指令集支持基于 RIP 的相对寻址就是为了简化 PIC 代码的操作。

然后，我们用同样版本的 GCC 把这段 C 代码编译为 64 位目标文件。

IDA 会在显示代码的时候隐藏那些基于 RIP 的寻址细节。因此，我们通过 objdump 查看汇编代码：

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00     mov    rax,QWORD PTR [rip+0x2008b9]       # 200fe0 <_DYNAMIC+0x1d0>
 727:   53                       push   rbx
 728:   89 fb                    mov    ebx,edi
 72a:   48 8d 35 20 00 00 00     lea    rsi,[rip+0x20]         # 751 <_fini+0x9>
 731:   bf 01 00 00 00           mov    edi,0x1
 736:   03 18                    add    ebx,DWORD PTR [rax]
 738:   31 c0                    xor    eax,eax
 73a:   89 da                    mov    edx,ebx
 73c:   e8 df fe ff ff           call   620 <__printf_chk@plt>
 741:   89 d8                    mov    eax,ebx
 743:   5b                       pop    rbx
 744:   c3                       ret
```

我们来看看以上程序代码中的 RIP 后面的两个偏移量：

① 指令 0x720 处。0x2008b9 是该地址与全局变量 global_variable 之间的地址差。

② 指令 0x72A 处。0x20 则是该地址与“returning %d”字符串指针之间的地址差。

可能读者也已经注意到了，经常进行地址重复计算会降低程序的执行效率（虽然在 64 位系统下可能会表现稍好）。因此，注重性能的时候，最好采用使用静态链接的静态库。

67.1.1 Windows

Windows 的 DLL 加载机制不是 PIC 机制。如果 Windows 加载器要把 DLL 加载到另外一个基地址，它就会内存中对 DLL 进行“修补”处理（重定位技术），从而可以正确地处理所有符号地址。这就意味着多个 Windows 进程无法在不同进程内存块的不同地址共享一份 DLL，因为每个被加载在内存里的实例只能访问自己的地址空间。

67.2 在 Linux 下的 LD_PRELOAD

Linux 程序可以加载其他动态库之前、甚至在加载系统库（例如 libc.so.6）之前加载自己的动态库。

借助这项功能，我们能够编写自定义的函数“替换”系统库中的同名函数。进一步说，劫持 time()、read()、write()等系统函数并非难事。

接下来，我们以系统工具 uptime 为例进行演示。我们都知道，该应用可以显示计算机已经工作了多少时间。借助另一款系统工具 strace 可知，uptime 通过/proc/uptime 文件获取计算机的工作时长：

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
```

```
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

其实，/proc/uptime 并不是真正意义上的磁盘文件。它是由 Linux Kernel 产生的虚拟文件。这个文件具有两项数值：

```
$ cat /proc/uptime
416690.91 415152.03
```

查查维基百科，我们可以得到以下的信息：

> 第一项数值显示的是系统已经运行的总时长；第二项数值是计算机处于空闲状态的时间总和。这两项数据都以秒为单位。

我们编写一个声明 open()、read()和 close()函数的自定义动态链接库。

首先要处理的就是 open()函数。它应能判断程序打开的文件是否是我们需要的文件。如果两者相符，那么 open()函数就应当记录并返回文件描述符。接下来要处理的是 read()函数。read()函数应能判断程序打开的是否是我们关注的文件描述符。如果两者相符，那么就用某些数据替代原有文件内容；否则就调用 libc.so.6 里的原有函数。最后需要处理的是 close()函数，它应能正确关闭已经打开的外部文件。

本例通过 dlopen()和 dlsym()函数获取同名函数在 libc.so.6 里的函数地址。虽然本例的确是要劫持系统函数，但是也得将控制权交还给原来的“正牌”函数。

另外一方面，如果我们要劫持 strcmp()函数（字符串比较函数）、以此获取每组对比的字符串，那么我们就得手写一个 strcpm()函数了，而无法继续调用原有函数。

这种劫持功能的程序源代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;
        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
```

```
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

我们用通用的动态库来编译它：

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

最后，我们通过 LD_PRELOAD 指令优先加载自定义的函数库：

```
LD_PRELOAD='pwd'/fool_uptime.so uptime
```

上述指令的输出结果为：

```
01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01, 0.05
```

如果我们在系统的环境变量中设定了 LD_PRELOAD、让它指向我们自定义的动态链接库，那么所有的进程都会在启动之前加载我们自定义的动态链接库。

更多例子请参阅：

① Very simple interception of the strcmp() (Yong Huang) :
https://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
② Fun with LD_PRELOAD (Kevin Pulo): https://yurichev.com/mirrors/LD_PRELOAD/lca2009.pdf。
③ File functions interception for compression/decompression：
ftp://metalab.unc.edu/pub/Linux/libs/compression

第 68 章　Windows NT

68.1　CRT (Win32)

所有程序都是从 main()函数开始执行的吗？事实并非如此。如果用 IDA 或者 HIEW 打开可执行文件，我们可以看到原始入口 OEP（Original Entry Point）总是指向其他的一段代码。这些代码会在启动程序之前进行一些维护和准备工作。这就是所谓的启动代码/startup-code 即 CRT 代码（C RunTime）。

在通过命令行指令启动程序的时候，main()函数通过外来数组获取启动参数及系统的环境变量。然而，实际传递给程序的不是数组而是参数字符串。CRT 代码会根据空格对字符串进行切割。另外，CRT 代码还会通过 envp 数组向 main()函数传递系统的环境变量。在 Win32 的 GUI 程序里，主函数变为了 WinMain()，并且拥有自己的参数传递规格：

```
int CALLBACK WinMain(
 _In_ HINSTANCE hInstance,
 _In_ HINSTANCE hPrevInstance,
 _In_ LPSTR lpCmdLine,
 _In_ int nCmdShow
);
```

上述参数同样是由 CRT 代码准备的。

在程序结束以后，主函数 main()会返回其退出代码。这个退出代码会被传递给 CRT 的 ExitProcess()函数，作为后者的一个参数。

通常来说，不同的编辑器会有不同的 CRT 代码。

以下列出的是 MSVC 2008 特有的 CRT 代码：

```
1  __tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8          push    14h
9          push    offset stru_4092D0
10         call    __SEH_prolog4
11         mov     eax, 5A4Dh
12         cmp     ds:400000h, ax
13         jnz     short loc_401096
14         mov     eax, ds:40003Ch
15         cmp     dword ptr [eax+400000h], 4550h
16         jnz     short loc_401096
17         mov     ecx, 10Bh
18         cmp     [eax+400018h], cx
19         jnz     short loc_401096
20         cmp     dword ptr [eax+400074h], 0Eh
21         jbe     short loc_401096
22         xor     ecx, ecx
23         cmp     [eax+4000E8h], ecx
24         setnz   cl
25         mov     [ebp+var_1C], ecx
26         jmp     short loc_40109A
27
```

```
28
29 loc_401096: ; CODE XREF: __tmainCRTStartup+18
30             ; __tmainCRTStartup+29 ...
31         and     [ebp+var_1C], 0
32
33 loc_40109A: ; CODE XREF: __tmainCRTStartup+50
34         push    1
35         call    __heap_init
36         pop     ecx
37         test    eax, eax
38         jnz     short loc_4010AE
39         push    1Ch
40         call    _fast_error_exit
41         pop     ecx
42
43 loc_4010AE: ; CODE XREF: __tmainCRTStartup+60
44         call    __mtinit
45         test    eax, eax
46         jnz     short loc_4010BF
47         push    10h
48         call    _fast_error_exit
49         pop     ecx
50
51 loc_4010BF: ; CODE XREF: __tmainCRTStartup+71
52         call    sub_401F2B
53         and     [ebp+ms_exc.disabled], 0
54         call    __ioinit
55         test    eax, eax
56         jge     short loc_4010D9
57         push    1Bh
58         call    __amsg_exit
59         pop     ecx
60
61 loc_4010D9: ; CODE XREF: __tmainCRTStartup+8B
62         call    ds:GetCommandLineA
63         mov     dword_40B7F8, eax
64         call    __crtGetEnvironmentStringsA
65         mov     dword_40AC60, eax
66         call    __setargv
67         test    eax, eax
68         jge     short loc_4010FF
69         push    8
70         call    __amsg_exit
71         pop     ecx
72
73 loc_4010FF: ; CODE XREF: __tmainCRTStartup+B1
74         call    __setenvp
75         test    eax, eax
76         jge     short loc_401110
77         push    9
78         call    __amsg_exit
79         pop     ecx
80
81 loc_401110: ; CODE XREF: __tmainCRTStartup+C2
82         push    1
83         call    __cinit
84         pop     ecx
85         test    eax, eax
86         jz      short loc_401123
87         push    eax
88         call    __amsg_exit
89         pop     ecx
90
91 loc_401123: ; CODE XREF: __tmainCRTStartup+D6
```

```
 92        mov     eax, envp
 93        mov     dword_40AC80, eax
 94        push    eax             ; envp
 95        push    argv            ; argv
 96        push    argc            ; argc
 97        call    _main
 98        add     esp, 0Ch
 99        mov     [ebp+var_20], eax
100        cmp     [ebp+var_1C], 0
101        jnz     short $LN28
102        push    eax             ; uExitCode
103        call    $LN32
104
105 $LN28:      ; CODE XREF: __tmainCRTStartup+105
106        call    __cexit
107        jmp     short loc_401186
108
109
110 $LN27:      ; DATA XREF: .rdata:stru_4092D0
111        mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
112        mov     ecx, [eax]
113        mov     ecx, [ecx]
114        mov     [ebp+var_24], ecx
115        push    eax
116        push    ecx
117        call    __XcptFilter
118        pop     ecx
119        pop     ecx
120
121 $LN24:
122        retn
123
124
125 $LN14:      ; DATA XREF: .rdata:stru_4092D0
126        mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 401044
127        mov     eax, [ebp+var_24]
128        mov     [ebp+var_20], eax
129        cmp     [ebp+var_1C], 0
130        jnz     short $LN29
131        push    eax             ; int
132        call    __exit
133
134
135 $LN29:      ; CODE XREF: __tmainCRTStartup+135
136        call    __c_exit
137
138 loc_401186: ; CODE XREF: __tmainCRTStartup+112
139        mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140        mov     eax, [ebp+var_20]
141        call    __SEH_epilog4
142        retn
```

在程序的第 62 行、第 66 行和第 74 行我们分别可以看到的是 GetCommandLineA、setargv()和 setenvp()这三个函数，从这三个函数的名称可以看出它们处理的分别是 argc、argv 和 envp 这三个全局变量。

最后，第 97 行的主函数 main()会获取这些外部参数。

CRT 中的函数名称通常都可以自然解释。例如第 35 行和第 54 行的 heap_init()和 ioinit()这两个函数。

堆的初始化操作是由 CRT 代码完成的。若在没有 CRT 代码的情况下调用内存分配函数 malloc()，就会引发异常退出，并将看到下述错误代码：

```
runtime error R6030
- CRT not initialized
```

在 C++程序中，CRT 代码还要在启动主函数 main()之前初始化全部全局对象。我们可以参考本书的

51.4.1 节。

主函数 main()返回的数值会传递给 cexit()，或者是在$LN32，随后会调用函数 doexit()。

下面一个问题是：我们有没有可能不采用 CRT 呢？这是有可能的，前提是您清楚地知道自己在做什么。

我们可以通过 MSVC 的 Linker 程序的/ENTRY 选项设置程序的入口点。

比如下面的这个程序代码：

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

选用以下的命令行来编译：

```
cl no_crt.c user32.lib /link /entry:main
```

上述指令最终生成一个大小为 2560 字节的可执行文件。该文件中具备标准的 PE 文件头，调用 MessageBox 的指令，其数据段声明了两个字符串，并从库文件 user32.dll 导入 MessageBox 函数。整个可执行文件没有其他的内容了。

虽然这个程序确实可以正常运行，但是这种程序无法获取 WinMain()函数所需的 4 个参数。确切地说，程序确实可以启动得起来，但是在程序启动得时候外部参数没有被准备或传递过来。

不能直接采用包括 4 个参数在内的主函数 WinMain()的方式，而且不采用 main()函数。更加精确一点来说，虽然能传递参数，但是参数不是在程序一执行时就被传递的。

另外，如果通过编译指令限定 PE 段向更小地址对齐（默认值是 4096 字节），那么.编译器将会生成尺寸更小的 exe 文件：

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

链接器 Linker 将会提示：

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

上述指令将生成一个长度为 720 字节的 exe 可执行文件。它可以运行于 x86 构架的 Windows 7 系统，但是却不能运行于 64 位的 Windows 7 系统（执行的时候，系统会给出错误提示）。从这里我们可以看到，虽然我们可以想办法让可执行文件变得更短一些，但是同时兼容性问题也会越来越突出。

68.2 Win32 PE 文件

PE（Portable Executable）格式，是微软 Windows 环境可移植可执行文件（如 exe、dll、vxd、sys 和 vdm 等）的标准文件格式。

与其他格式的 PE 文件不同的是，exe 和 sys 文件通常只有导入表而没有导出表。

和其他的 PE 文件一样，DLL 文件也有一个原始代码入口点 OEP（就是 DllMain()函数的地址）。但是 DLL 的这个函数通常来讲什么也不会做。

sys 文件通常来说是一个系统驱动程序。说到驱动程序，Windows 操作系统需要在 PE 文件里保存其校验和，以验证该文件的正确性（Hiew 就可以验证这个校验和）。

从 Vista 开始，所有的 Windows 驱动程序必须具备数字签名，否则系统会拒绝加载它们。

每个 PE 文件都由一段打印“This program cannot be run in DOS mode.”的 DOS 程序块开始。如果在 DOS 或者 Windows 3.1 环境下运行这个程序，那么只会看到上述字符串。因为 DOS 及 Windows 3.1 系统不能识别 PE 格式的文件。

68.2.1 术语

- 模块 Module：它是指一个单独的 exe 或者 dll 文件。
- 进程 Process：加载到内存中并正在运行的程序，通常由一个 exe 文件和多个 dll 文件组成。
- 进程内存 Process memory：每个进程都有完全属于自己的，进程间独立的，不被干扰的内存空间。通常是模块、堆、栈等数据构成。
- 虚拟地址 VA（Virtual Address）：程序访问存储器所使用的逻辑地址。
- 基地址 Base Address：进程内存中加载模块的首地址。
- 相对虚拟地址 RVA（Relative Virtual Address）：虚拟地址 VA 与基地址 Base Address 的差就是相对虚拟地址 RVA。在 PE 文件表中的很多地址都是相对虚拟地址 RVA。
- 导入地址表 IAT（Import Address Table）：导入符号的地址数组。PE 头里的 IMAGE_DIRECTORY_ENTRY_IAT 指向第一个导入地址表 IAT 的开始位置。值得说明的是，反编译工具 IDA 可能会给 IAT 虚构一个伪段--.idata 段，即使 IAT 是其他地址的一部分。
- 导入符号名称表 INT（Import Name Table）：存储着所需符号名称的数组。

68.2.2 基地址

在开发各自的 DLL 动态链接库文件时，多数开发团队都有意让其他人直接调用自己的动态链接库。然而，具体到“谁的 DLL 到底应该加载到哪个地址”这种问题，却没有一种公开的协议或标准。

因此，当同一个进程的两个 DLL 库具有相同的基地址时，只会有一个 DLL 被真正加载到基地址上。而另外一个 DLL 则会分配到进程内存的某段空闲空间里。在调用后者时，每个虚拟地址都会被重新校对。

通常来说，MSVC 编译成的可执行程序的基地址都是 0x400000，而代码段则从 0x401000 开始。由此可知，这种程序代码段的相对虚拟地址 RVA 的首地址都是 0x1000。而 MSVC 通常把 DLL 的基地址设定为 0x10000000。

操纵系统可能会把模块加载到不同的基地址中，还可能是因为程序自身的要求。当程序“点名”启用地址空间分布的随机化（Address Space Layout Randomization，ASLR）技术的时候，操作系统会把其各个模块加载到随机的基地址上。

ASLR 是 shellcode 的应对策略。shellcode 都会调用系统函数。

在 Windows Vista 之前早期系统里，系统的 DLL（如 kernel32.dll，user32.dll 的加载地址是已知的固定地址。在同一个版本的操作系统里，系统 DLL 里的系统函数地址也几乎一尘不变。也就是说，shellcode 可以根据版本信息直接调用系统函数。

为了避免这个问题，地址空间分布的随机化 ASLR 技术应运而生。它能够将程序以及程序所需模块加载到无法事先确定的随机地址。

在 PE 文件中，我们通过设置一个标识来实现 ASLR。这个标识的名称是：IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE。

68.2.3 子系统

PE 文件有一个子系统字段。这个字段的值通常是下列之一：

- NATIVE（系统驱动程序）。
- console 控制台程序。
- GUI（非控制台程序，也就是最常见的图文界面程序）。

68.2.4 操作系统版本

PE 文件还指定了可加载它的 Windows 操作系统最低版本号。如需查阅版本号码和 Windows 发行名称的完整列表，请参阅：https://en.wikipedia.org/wiki/Windows_NT#Releases。

举个例子，MSVC 2005 编译的.exe 文件只能运行在 Windows NT4（版本号为 4.00）及以后的操作系统上。但是 MSVC 2008 编译调应用程序（版本号是 5.00）不兼容 NT4 系统，只能运行于 Windows 2000 及以后的操作系统。

在 MSVC 2012 生成的.exe 文件里，操作系统版本号的默认值是 6.00。这种程序仅面向 Windows Vista 及后期推出的操作系统。但我们可以编译选项强制编译器生成支持 Windows XP 的应用程序。详情请参阅 https://blogs.msdn.microsoft.com/vcblog/2012/10/08/windows-xp-targeting-with-c-in-visual-studio-2012/。

68.2.5 段

所有的可执行文件都可分解为若干段（sections）。段是代码和数据、以及常量和其他数据的组织形式。

- 带有 IMAGE_SCN_CNT_CODE 或 IMAGE_SCN_MEM_EXECUTE 标识的段，封装的是可执行代码。
- 数据段的标识为 IMAGE_SCN_CNT_INITIALIZED_DATA、IMAGE_SCN_MEM_READ 或 IMAGE_SCN_MEM_WRITE 标记。
- 未初始化的数据的空段的标识为 IMAGE_SCN_CNT_UNINITIALIZED_DATA、IMAGE_SCN_MEM_READ 或 IMAGE_SCN_MEM_WRITE。
- 常数数据段（其中的数据不可被重新赋值）的标识是 IMAGE_SCN_CNT_ INITIALIZED_DATA 以及 IMAGE_SCN_MEM_READ，但是不包括标识 IMAGE_SCN_MEM_WRITE。如果进程试图往这个数据段写入数据，那么整个进程就会崩溃。

PE 可执行文件的每个段都可以拥有一个段名称。然而，名称不是段的重要特征。通常来说，代码段的段名称是.text，数据段的段名称是.data，常数段的段名称.rdata（只读数据）。其他类型的常见段名称还有：

- .idata：导入段。IDA 可能会给这个段分配一个伪名称；详情请参考本书 68.2.1 节。
- .edata：导出段。这个段十分罕见。
- .pdata：这个段存储的是用于异常处理的函数表项。它包含了 Windors NT For MIPS、IA64 以及 x64 所需的全部异常处理信息。详情请参考本书的 68.3.3 节。
- .reloc：（加载）重定向段。
- .bss：未初始化的数据段（BSS）。
- .tls：线程本地存储段（TLS）。
- .rsrc：资源。
- .CRT：在早期版本的 MSVC 编译出的可执行文件里，可能出现这个这个段。

经过加密或者压缩处理之后，PE 文件 section 段的段名称通常会被替换或混淆。

此外，开发人员还可以控制 MSVC 编译器、设定任意段的段名称。有关详情请参阅：https://msdn.microsoft.com/en-us/library/windows/desktop/cc307397.aspx。

部分编译器（例如 MinGW）和链接器可以在生成的可执行文件中加入带有调试符号、或者其他的调试信息的独立段。然而新近版本的 MSVC 不再支持这项功能。为了便于专业人员分析和调试应用程序，MSVC 推出了全称为“程序数据库”的 PDB 文件格式。

PE 格式的 section 段的数据结构大体如下：

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD NumberOfRelocations;
```

```
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

上述代码摘自于：https://msdn.microsoft.com/en-us/library/windows/desktop/ms680341(v=vs.85).aspx。

简单的说，上述 PointerToRawData（指向原始数据）就是段实体的偏移量 Offset，而虚拟地址 VirtualAddress 就是 Hiew 里的 RVA。

68.2.6 重定向段 Relocations(relocs)

至少在 Hiew 中，它也可以表示为 FIXUP-s。

重定位段是自 MS-DOS 时代起一直存在于可执行文件的实体段，几乎所有的可执行文件都有这个段。

前文介绍过，程序模块可能会被加载到不同的基地址。但是如何处理全局变量等局部共享数据呢？程序必须通过地址指针才能访问这类数据，然而程序又不可能事先知道共享数据的存储地址。为解决这个问题，人们推出了“位置无关代码/PIC”（详情请参阅 67.1 节）的解决方案。不过 PIC 用起来并不方便。

于是，人们又推出了基于“重定向表”的地址修正技术。重定向表记录了该文件被加载到不同基地址时需要修正的所有指针。

如果 PE 文件声明了一个地址为 0x410000 的全局变量，那么这个变量的寻址指令大致会是：

```
A1 00 00 41 00        mov        eax,[000410000]
```

此时，模块的基地址是 0x400000（编译器默认值），全局变量的相对虚拟地址 RVA 是 0x10000。

如果这个模块被加载到首地址为 0x500000 的基地址上，那么这个全局变量的真实地址应当被调整为 0x510000。

在上述 opcode 中，变量地址应当是 0xA1（“MOV EAX”指令）之后的那几个字节。为了通知操作系统在重定向时正确处理该地址，PE 文件的重定向表必须收录这四个字节的相对地址。

当操作系统的加载器需要把这个模块加载到不同的基地址时，它会逐一枚举重定位表中所有地址，把这些地址所指向的数据当作 32 位的地址指针，然后减去初始基地址（这样就能获得 RVA，也就是相对虚拟地址），并加上新的基地址。

如果模块被加载于其原始的基地址，那么操作系统就不会进行重定向处理。

所有全局变量的处理过程都是如此。

重定向段的数据结构并不唯一。Windows for x86 程序的重定向段一般采用 IMAGE_REL_BASED_HIGHLOW 的数据结构。

另外，Hiew 用暗色区域显示重定向段，如图 7.12 所示。

而 OllyDbg 会在内存中用下画线的方式标记重定向段，如图 13.11 所示。

68.2.7 导出段和导入段

我们都知道，任何可执行文件都必须或多或少地调用由操作系统提供的服务或者 DLL 动态链接库。

广义地说，由某个模块（一般来说是 DLL）声明的所有函数，最终都会被其他模块（.exe 文件或者其他 DLL）中的某条指令调用，只是调用方式不同而已。

为此，每一个 DLL 动态链接库文件都有一个“导出/exports“表。导出表声明了该模块定义的函数名称和函数的地址。

同时每个 exe 文件和 DLL 文件另有一个“导入”表。这个表声明了执行该模块所需的函数名称、以及相应 DLL 文件的文件名。

在加载完可执行文件主体的.exe 文件之后，操作系统加载器开始处理导入表：它会加载记录在案的 DLL 文件，接着在 DLL 的导出表里查找所需函数的函数地址，最后把这些地址写到.exe 模块的 IAT 导入表。

由此可见，操作系统的加载器在进程的加载过程中要比对大量的函数名称。但是检索字符串的效率不会很高。后来人们引入了基于“排行榜”或者“命中率”（对应相关数据结构里的 Hints 或 Ordinal 字段）的序号表示办法、把函数名称编排为数字，以此摆脱字符串操作的低效率问题。

因此，DLL 文件只需在导出表里标注内部函数的函数“序号”。这显著提升了 DLL 文件的加载速度 。

举例来说，调用 MFC 的程序就能够通过函数“序号”调用 mfc*.dll 动态链接库。而这种程序不再需要导入段的数据表（Import Name Table，INT）中使用字符串存储 MFC 的函数名称。

当我们在 IDA 中加载这样的程序后，IDA 会询问 mfc*.dll 文件的路径以获取外部函数的函数名称。如果我们没有指定 MFC 文件的存储路径，那么函数名称会是 mfc80_123 之类的字符串。

导入段

编译器通常会给导入表及其相关内容分配一个单独的 section 段（例如.idata），但这不是一个强制规定。导入段涉及大量的技术术语，因此理解起来特别困难。本节将通过一个典型的例子进行集中演示。

导入段的主体是数组 IMAGE_IMPORT_DESCRIPTOR。它记录着 PE 文件要导入哪些库文件。

在其元素的数据结构中：

- Name 字段存储着库名称字符串的 RVA 地址；
- OriginalFirstThink 字段是 INT 表的 RVA 地址。逐一读取这个字段对应的 INT 数组的值，可获取相应 IMAGE_IMPORT_BY_NAME 地址（RVA）、进而得到全部函数名称。IMAGE_IMPORT_BY_NAME 表（没有收录在图 68.1 里）里定义了一个 16 位整数的“hint”字段，它正是前文所说函数“序号”。
  在加载模块时，如果可以通过序号检索所需函数名，那么操作系统加载器就不必进行费时的字符串比对操作。

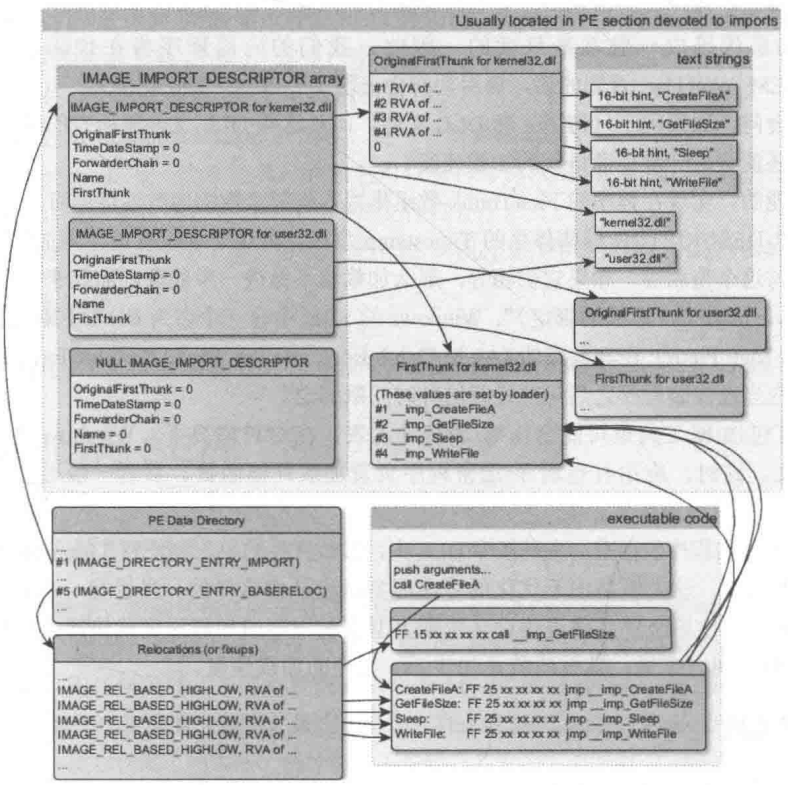


图 68.1 在 PE 范畴内与导入段相关的全部数据结构

- FirstThunk 字段存储的是 IAT 表的表指针。IAT 表的每个成员元素都是由操作系统加载器解析出

来的函数地址（RVA）。IDA 会给这些元素添加“:_imp_CreateFileA”一类的名称标注。

由加载器解析出来的外部函数地址，至少有两种调用方法：

① 代码可通过 call_imp_CreateFileA 形式的指令直接调用外部函数。从某种意义上讲，导入函数的函数地址存储到全局变量的存储空间了。考虑到当前模块可能会被加载于与初始值不同的基地址上，那么只要把 call 指令引用的外部函数目标地址直接追加到 reclos 重定向表里就好了。

但是要把导入函数的函数地址全部追加到重定向段 relocs 里，会显著增加重定向表的数据容量。进一步来说，重定向表越大、程序的加载效率就越低。

② 另一种办法就是对每个调用点进行处理。在调用外部函数的时候，只要 JMP 到“重定向值+外部函数 RVA”就可以调用外部函数了。这种调用方式也叫做“形实转换/thunks”。调用外部函数时，程序可以直接 CALL 相应的 thunk 地址。这种调用方式无需进行重定位运算，因为 CALL 指令本身就能进行相对寻址，因此不必修正 RVA 地址。

编译器能够分派上述两种调用方法。如果外部函数的调用频率很高，那么链接器 Linker 很可能会通过 thunk 调用外部函数。但是在默认情况下，链接器并不创建 thunk。

另外，FirstThunk 字段里的函数指针数组不必在 PE 文件的导入地址段（Input Address Table，IAT section）中。笔者曾经编写过一个令.exe 文件添加外部函数信息的 PE_add_import 工具（https://yurichev.com/PE_add_imports.html）。它就曾经成功的将 PE 文件引用的外部函数替换为另外一个 DLL 文件的其他函数。此时生成的指令是：

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk 字段存储的是外部函数第一条指令的地址。换而言之，在调用 youdll.dll 这类自定义动态链接库的时候，加载器把程序代码的相应位置直接替换为外部函数 function 的函数地址。

需要注意的是代码段一般都是只读的。因此，我们的应用程序将在代码段上增加一个标志 IMAGE_SCN_MEM_WRITE。否则的话，调用期间会出现 5 号错误（禁止访问）。

可能有读者会问，如果程序只调用一套 DLL 文件，而且这些 DLL 文件里所有的函数名称和函数地址保持不变，那么还能否进一步提高进程的加载速度？

答案是：可能的。实现在程序的 FirstThunk 数组里写入外部函数的函数地址即可。另外需要注意的是 IMAGE_IMPORT_DESCRIPTOR 结构体里的 Timestamp 字段。若这个字段有值，则加载器会判断 DLL 文件的时间戳是否与这个值相等。如果它们相等，那么加载器不做进一步处理，加载速度可能会快些。这就是所谓的“old-style binding（古板的绑定）”。Windows 的 SDK 中有一个名为 BIND.EXE 的工具可以专门进行这项绑定设置。Matt Pietrek 在其发布的《An In-Depth Look into the Win32 Portable Executable File Format》建议，终端用户应当在安装程序之后尽快进行这种时间戳绑定。

PE 文件的打包/加密工具也可能会压缩/加密导入表。在这种情况下，Windows 的加载器无法加载全部所需的 DLL。这时，应由打包程序/加密程序负责加载外部函数。后者一般通过 LoadLibrary()和 GetProcAddress()来完成这项任务。

在 Windows 的安装程序的标准动态链接库 DLL 中，多数文件的导入地址表（Input Address Table，IAT）都位于 PE 文件的头部。这大概是出于优化的考虑而刻意设计成这样的。当运行一个 exe 可执行文件时，exe 文件并不会被一次性地全部装载进内存（否则大型安装程序的加载速度就快得太离谱了），而是在访问过程中被分部映射到内存里。其目的就是加快 exe 文件的加载速度。

68.2.8 资源段

位于 PE 文件资源段里的数据无非就是图表、图形、字符串以及对话框描述等界面信息。这些资源与主程序指令分开存储，大概是为了方便实现多语言的支持：操作系统只需要根据系统的语言设置就可以选取相应的文本或图片。

而这其实也带来了一个副作用：因为 PE 可执行文件比较容易编辑，不具备 PE 文件专业知识的人也可以借助工具（ResHack 等）直接修改程序资源。相关的介绍可以参见本书 68.2.11 节。

68.2.9 .NET

.NET 的源程序并不会被编译成机器码，而会被编译成一种特殊的字节码。严格地说，这种.exe 文件由字节码构成，并不是常规意义上的 x86 指令代码。但是这种程序的入口点（OEP）确实是一小段 x86 指令：

```
jmp         mscoree.dll!_CorExeMain
```

.NET 格式的 PE 文件由 mscoree.dll 处理，它同时是.NET 程序的装载器。在 Windows XP 操作系统之前，.net 程序都是通过上述 jmp 指令交由 mscoree.dll 处理的。自 Windows XP 系统起，操作系统的加载器能自动识别.NET 格式的文件，即使没有上述 JMP 指令也可以正常加载.NET 程序。有关详情请参阅：

https://msdn.microsoft.com/en-us/library/xh0859k0(v=vs.110).aspx

68.2.10 TLS 段

这个段里存储了 TLS 数据（第 65 章）的初始化数据（如果需要的话）。当启动一个新的线程时，TLS 的数据就是通过本段的数据来初始化的。

除此之外，PE 文件规范还约定了“TLS!”的初始化规范，即 TLS callbacks/TLS 回调函数。如果程序声明了 TLS 回调函数，那么 TLS 回调函数会先于 OEP 执行。这项技术广泛应用于 PE 文件的压缩和加密程序。

68.2.11 工具

- Objdump（cygwin 版），可转储所有的 PE 文件结构。
- Hiew（可以参考本书第 73 章）。这是一个编辑器。
- Prefile。这是一个用来处理 PE 文件的 Python 库。
- ResHack。它是 Resource Hacker 的简称，是一个资源编辑器。
- PE_add_import。这是一个小工具，利用它可以将符号加入到 PE 可执行文件的导入表中。
- PE_patcher。一个小工具，可以用来给 PE 文件打补丁。
- PE_search_str_refs。一个小工具，可以用来在 PE 可执行文件中寻找函数，这些函数可能有些字符串。

68.2.12 更进一步

- Daniel Pistelli:《.NET 文件格式》：https://www.codeproject.com/articles/12585/the-net-file-format。

68.3 Windows SEH

68.3.1 让我们暂时把 MSVC 放在一边

Windows 操作系统中，结构性例外程序处理机制（Structured Exception Handling，SEH）是用来处理异常情况的响应机制。然而，它是与语言无关的，与 C++或者面向对象的编程语言（Oriented Object Programming，OOP）无任何关联。本节将脱离 C++以及 MSVC 的相关特效，单独分析 SEH 的特性。

每个运行的进程都有一条 SEH 句柄链，线程信息块（Thread Information Block，TIB）有 SHE 的最后一个句柄。当出现异外时（比如出现了被零除、地址访问不正确或者程序主动调用 RaiseException()函数等情况），操作系统会在线程信息块 TIB 里寻找 SEH 的最后一个句柄。并且把出现异常情况时与 CPU 有关的所有状态（包括寄存器的值等数据）传递给那个 SEH 句柄。此时异常处理函数开始判断自己能否应对这种异常情

况。如果答案是肯定的，那么异常处理函数就会着手接管。如果异常处理函数无法处理这种情况，它就会通知操作系统无法处理它，此后操作系统会逐一尝试异常处理链中的其他处理程序，直到找到能够应对这种情况的异常处理程序为止。

在异常处理链的结尾处有一个大家都接触过的异常处理程序：它显示一个标准的对话框，通知用户进程已经崩溃，崩溃时 CPU 的状态信息是什么情况，用户是否愿意把这些信息发送给微软的开发人员。

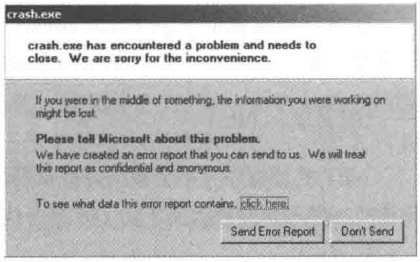


图 68.2 Windows XP 下的崩溃截图

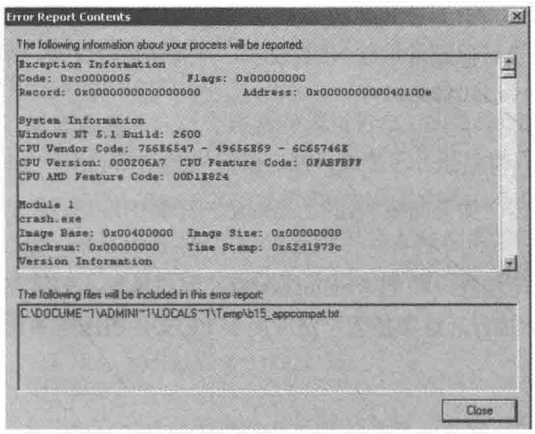


图 68.3 Windows XP 下的崩溃细节

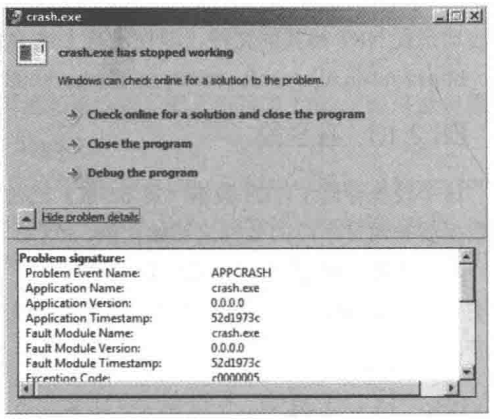


图 68.4 Windows 7 下的崩溃细节

早些时候，这个异常处理程序叫做 Dr. Watson。

另外，一些开发人员会在程序里设计自己的异常处理程序，以便收集程序的崩溃信息。这些都是通过系统函数 SetUnhandledExceptionFilter() 注册的异常处理函数。当操作系统遇到无法应对的异常情况时，它就会调用应用程序自己注册的异常处理函数。Oracle 的 RDBMS 就是十分典型的一个例子：它会在程序崩溃时尽可能地转储 CPU 以及内存数据。

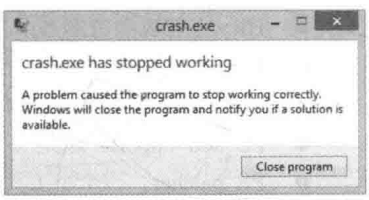


图 68.5 Windows 8.1 下的崩溃截图

接下来，我们研究一个初级的异常处理程序。它摘自于 https://www.microsoft.com/msj/0197/Exception/Exception.aspx：

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
{
        unsigned i;
        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
```

```
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
        {
                printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n");
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer to our handler

        // install exception handler
        __asm
        {                               // make EXCEPTION_REGISTRATION record:
                push    handler         // address of handler function
                push    FS:[0]          // address of previous handler
                mov     FS:[0],ESP      // add new EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                               // remove our EXECEPTION_REGISTRATION record
                mov     eax,[ESP]       // get pointer to previous record
                mov     FS:[0], EAX     // install previous record
                add     esp, 8          // clean our EXECEPTION_REGISTRATION off stack
        }

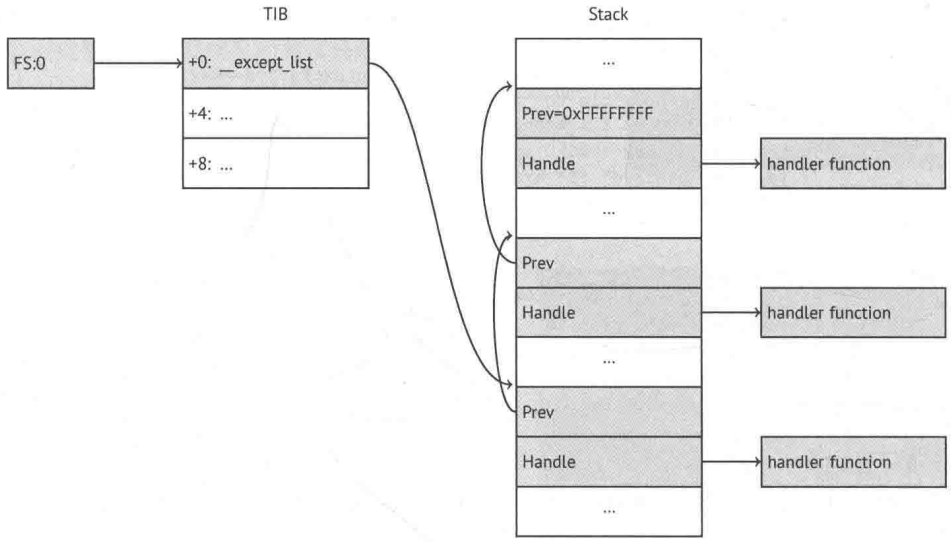
        return 0;
}
```

在 Win32 环境下，FS：段寄存器里的数据就是线程信息块（Thread Information Block，TIB）的指针。而 TIB 中的第一个元素正是异常处理指针链里最后一个处理程序的地址。所谓“注册”异常处理程序就是把自定义的异常处理程序的地址链接到这个异常处理指针链里。在注册异常处理程序时，需要使用一种名为_EXCEPTION_REGISTRATION 的数据结构。它其实只是一个简单的单向链表，使用栈结构存储各项节点的链数据。

指令清单 68.1 MSVC/VC/crt/src/exsup.inc

```
\_EXCEPTION\_REGISTRATION struc
    prev    dd      ?
    handler dd      ?
\_EXCEPTION\_REGISTRATION ends
```

可见，每个节点的“handler”都是一个异常处理程序的启始地址，每个节点的“prev”字段都是上一个节点的地址指针。而最后一个节点的“prev”字段的值为 0xFFFFFFFF(-1)。



在注册好我们自定义的异常处理程序以后，我们调用 RaiseException()函数、触发用户异常的处理过程。异常处理程序首先检查异常代码，如果异常代码是 0xE1223344，它就返回 ExceptionContinueExecution。这个返回值代表“已经纠正 CPU 的状态”（通常通过调整 EIP/ESP 寄存器实现）、“操作系统可以继续执行后续指令”。如果把异常代码修改为其他值，那么处理函数的返回值则会变为 ExceptionContinueSearch。顾名思义， 操作系统就会逐一尝试其他的异常处理程序—万一没有找到有关问题（并不仅仅是错误代码）的异常处理程序，我们就会看到标准的 Windows 进程崩溃对话框。

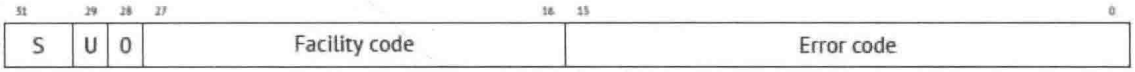
系统异常（system exceptions）和用户异常（user exceptions）之间的区别是什么？系统异常的有关信息是：

由 WinBase.h 定义的异常状态	在 ntstatus.h 里的相应状态	错误编号
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT0x80000002	
EXCEPTION_BREAKPOINTSTATUS_ BREAKPOINT0x80000003		
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D

续表

as defined in WinBase.h	as defined in ntstatus.h	numerical value
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

32 位错误代码的具体含义，如下图所示。



S 是最高的两位（第 30、31 位），为基本的状态代码，一共有 4 种组合，分别是 11、10、01 以及 00。它们分别代码的意义是：11 代表错误，10 代表警告，01 代表信息，00 代表成功。

U 是第 29 位（第 29 位），它只有 0 和 1 两种状态，代表该异常是否属于用户侧异常。

上面我们提到的一个返回值是 0xE1223344。最高的 4 位是 0xE（1110）。这几个比特位代表：①这是属于用户态异常；②这是一个错误信息。实事求是地讲，本例这个程序与这些最高位的值没有关系；而考究的异常处理程序应当能够充分利用错误代码的所有信息。

接着，程序试图读取地址为 0 的内存数据。这个地址实际就是 NULL 指针指向的地址。访问这个地址必将导致系统错误，因为根据 ISO C 标准这个地址不应当存放任何数据（硬性规定）。此时操作系统会优先调用程序自己注册的异常处理程序。后者会判断错误代码是否为 EXCEPTION_ACCESS_VIOLATION，从而得知该异常是否是自己可以处理的问题。

读取地址为 0 的指令大致如下：

指令清单 68.2　MSVC2010

```
...
xor     eax, eax
mov     eax, DWORD PTR [eax] ; exception will occur here
push    eax
push    OFFSET msg
call    _printf
add     esp, 8
...
```

我们的程序是不是可以实时处理这个错误以使得程序能继续执行呢？答案是肯定的。我们的异常处理程序能够修正 EAX 寄存器的值，让操作系统继续执行下去。这就是自定义异常处理程序的功能。字符串显示函数 printf 显示的数值是 1234，因为执行了我们的异常处理函数之后，EAX 的数值不再是 0 了，而是全局变量 new_value 的值。因此执行流程就得到了恢复。

以下就是程序执行的步骤：首先内存管理器检测出由中央处理器 CPU 发出的错误信息，接着 CPU 将此进程挂起，并在 Windows 的内核中检索异常处理程序的句柄。然后依次调用 SEH 链的 handler。

本例是由 MSVC 2010 编译的程序。当然，我们并不能保证其他的编译程序同样会使用 EAX 寄存器存放该指针。

它所演示的地址替换技巧非常的精妙。我经常使用这种技术演示 SEH 的内部构造。不过，我还不曾使用这种技巧实时修复异常错误。

为什么 SEH 相关的记录存储于栈，而不是其他的地方？据说，如此一来操作系统就不需要关心这类数据的释放操作，毕竟函数结束以后这些数据都会被自动释放。但是，笔者也不难 100%保证这种假说的正确

性。这有点像本书 5.2.4 节讲到的 alloca()函数。

68.3.2 让我们重新回到 MSVC

据说，C++语言开发环境已经能够稳妥的处理各种异常情况，只有 C 语言的开发人员才需要关注代码的异常处理机制。所以微软推出了一个面向 MSVC 的非标准 C 扩展。这个扩展组件不适用于 C++程序。

有关详情请参阅：https://msdn.microsoft.com/en-us/library/swezty51.aspx

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

除了“try-except”语句之外，MSVC 还支持“try-finally”语句：

```
__try
{
    ...
}
__finally
{
    ...
}
```

前者中的“filter code”是一个用来判断是否执行“handler code（响应指令）”的表达式。如果代码太长、无法表示为一个表达式，那么就得借助于独立的过滤函数。

Windows 内核就大量使用这种 SEH 结构。以 WRK（Windows Research Kernel）的某段指令为例：

指令清单 68.3 WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

指令清单 68.4 WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                    (PAGE_SIZE - PageOffset) :
                    (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                    &Status ) ) {
```

下面也是一组过滤代码。

指令清单 68.5 WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )

/*++

Routine Description:

    This routine serves as a exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
                    status.
Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

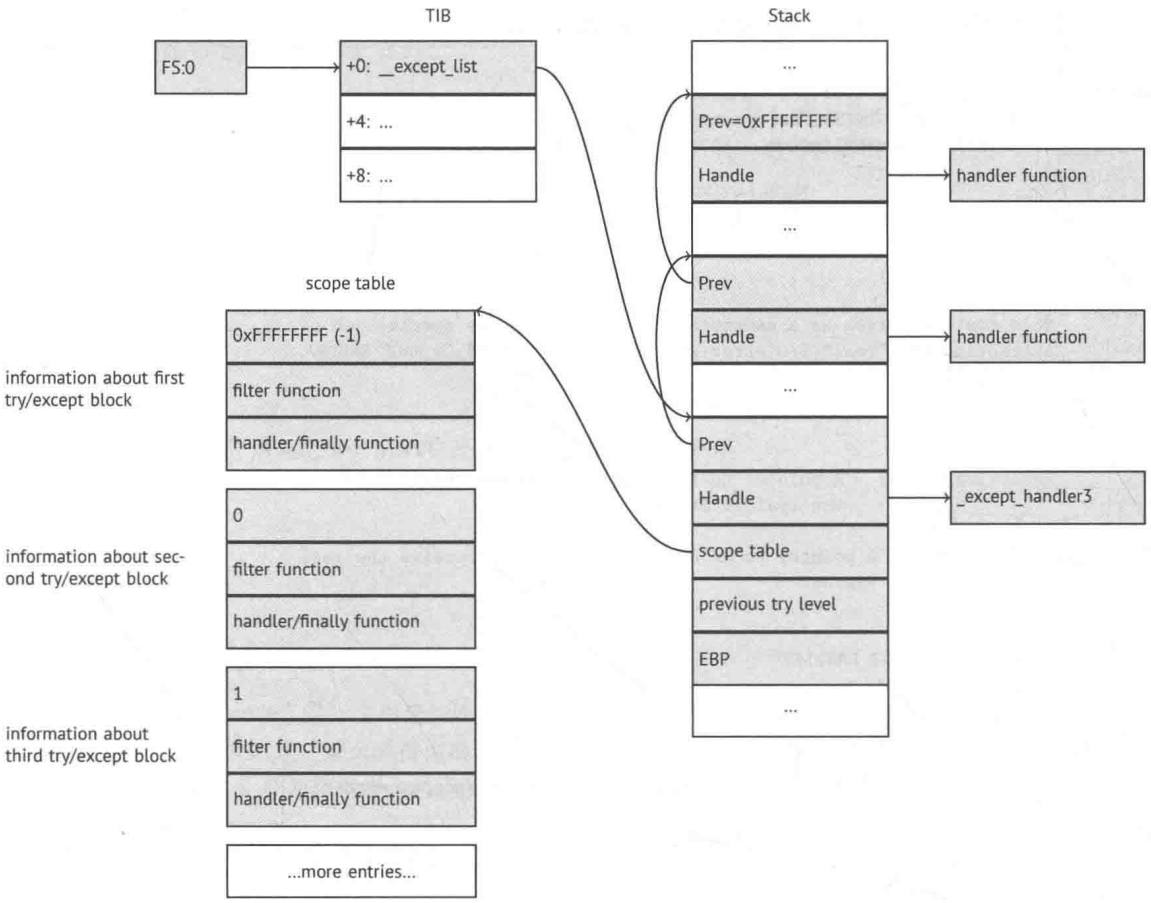
从内部来讲，SEH 是由操作系统支持的异常处理扩展。但是异常处理函数属于_except_handler3（SEH3）或_except_handler4（SEH4）。而且响应代码依赖于 MSVC 编译器。它需要由 MSVC 的库文件或者 msvcr*.dll 的动态链接库提供支持。SEH 是由 MSVC 提供的一种机制，这一点至关重要。其他 Win32 的编译器的异常响应机制可能与 SEH 完全不同。

SEH3

SEH3 定义了一个_except_handler3 的异常处理函数，而且还对_EXCEPTION_REGISTRATION 表进行了扩充、添加了 scope table 和 previous try level 的指针。在此基础上，SEH4 对 scope table 表添加了四个值，以实现缓冲溢出保护。

scope table 是一个表，它的元素都由 filter 表达式和 handler code 块的指针构成。这个表可以正确处理带有嵌套关系的多级 try/except 语句。

本书再次强调，操作系统只关心 SEH 里各节点的 prev 字段和 handle 字段，从不关心任何其他数据。函数_except_handler3 的作用是读取其他的字段以及 scope 表的数值，并且判断什么时间执行哪个异常处理函数。



函数_except_handler3 的源代码并不公开。然而，Sanos 操作系统（与 Win32 系统部分兼容），再现了这个函数。在某种程度上说，由 Sanos 实现的_except_handler3 与 Windows 系统的同名函数十分相似（请参阅其源文件/src/win32/msvcrt/except.c）。除此以外，Wine 平台和 ReactOS 系统也开发了类似的函数。

如果 filter 指针是空指针 NULL，那么 handler 指针将指向“finally”所在的代码块。

在执行过程中，堆栈中的 previous try level 改变了，因此函数_except_handler3 能从目前的嵌套中获取信息，目的是知道使用哪个 scope table 表。

执行期间，栈中的 previous try level 字段将会发生变化。_except_handler3 根据这项数据获取当前嵌套级的信息，进而判断要使用 scope table 中的哪个表项。每一个 try 块都分配了一个唯一的数作为标识，scopetable 表中条目（entry）间的关系则描述了 try 块的嵌套关系。

SEH3: 一个 try/except 块的例子

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13; // causes an access violation exception;
        printf("hello #2!\n");
```

```
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

指令清单 68.6 MSVC 2003

```
$SG74605 DB      'hello #1!', 0aH, 00H
$SG74606 DB      'hello #2!', 0aH, 00H
$SG74608 DB      'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST    SEGMENT
$T74622  DD      0ffffffffH   ; previous try level
         DD      FLAT:$L74617 ; filter
         DD      FLAT:$L74618 ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1                              ; previous try level
    push    OFFSET FLAT:$T74622             ; scope table
    push    OFFSET FLAT:__except_handler3   ; handler
    mov     eax, DWORD PTR fs:__except_list
    push    eax                             ; prev
    mov     DWORD PTR fs:__except_list, esp
    add     esp, -16
; 3 registers to be saved:
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET FLAT:$SG74605 ; 'hello #1!'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET FLAT:$SG74606 ; 'hello #2!'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; previous try level
    jmp     SHORT $L74616

    ; filter code:
$L74617:
$L74627:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74621[ebp], eax
    mov     eax, DWORD PTR $T74621[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
```

```
    inc     eax
$L74619:
$L74626:
    ret     0

    ; handler code:
$L74618:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74608 ; 'access violation, can''t recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back to -1
$L74616:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:__except_list, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main   ENDP
_TEXT   ENDS
END
```

由此可见，SEH 在栈里形成了帧结构。而 Scope table 则是位于文件的 CONST 段，确实如此，scope table 各字段的值确实不会发生变化。值得关注的是 previous try level 字段的变化过程。其初始值是 0xFFFFFFF(-1)。当执行到 try 语句时，专有一条指令把它赋值为 0。而当 try 语句的主体关闭时，它又被赋值为-1。我们也看到了 filter 以及 handler code 的地址。因此，我们能很容易地分析出函数中的 try-except 语句。

函数序言中的 SEH 初始化代码可能会被多个函数共享，有时候编译器会在函数序言直接调用 SEH_prolog()函数，再在函数尾声处调用 SEH_epilog()函数以回收栈空间。

下面我们在 tracer 跟踪器中运行这个例子：

```
tracer.exe -l:2.exe --dump-seh
```

上述指令的输出如下。

指令清单 68.7 tracer.exe 的输出

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070   (2.exe!main+0x60) handler=0x401088 ↙
    ↳ (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1,   filter=0x401531 (2.exe!mainCRTStartup+0x18d) ↙
    ↳ handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xfffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↙
    ↳ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

我们可以看到 SEH 链含有 4 个处理函数/handler。

前两个 handler 是由源代码指定注册的异常处理函数。虽然我们的源代码只定义了一个 handler，但是

CRT 的_mainCRTStartup()函数会自动设置一个配套的 handler。后者的功能不多，至少能够处理一些与 FPU 有关的异常情况。有关源码可以参阅 MSVC 安装目录里的 crt/src/winxfltr.c 文件。

第三个 handler 是由 ntdll.dll 提供的 SEH4，第四个 handler 也位于 ntdll.dll，跟 MSVC 没什么关系，它的函数名是 FinalExceptionHandler。

上述信息表明，SEH 链含有三种类型的处理函数：一种是与 MSVC 彻底无关的自定义 handler（即异常处理指针链中的最后一项），另外两种处理程序是由 MSVC 提供的 SEH3 和 SEH4 函数。

SEH3：两个 try/except 模块例子

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13; // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
                EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

这里，我们可以看到两个 try 块。因此 scope table 会有两个元素，分别存储着各 try 块的相应指针。“Previous try level”字段的值会伴随着进入/退出 try 语句块而发生相应改变。

指令清单 68.8 MSVC 2003

```
$SG74606 DB      'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB      'yes, that is our exception', 0aH, 00H
```

```
$SG74610 DB      'not our exception', 0aH, 00H
$SG74617 DB      'hello!', 0aH, 00H
$SG74619 DB      '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB      'access violation, can''t recover', 0aH, 00H
$SG74623 DB      'user exception caught', 0aH, 00H

_code$ = 8    ; size = 4
_ep$ = 12     ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push   OFFSET FLAT:$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table:
CONST    SEGMENT
$T74644   DD     0ffffffffH   ; previous try level for outer block
          DD     FLAT:$L74634 ; outer block filter
          DD     FLAT:$L74635 ; outer block handler
          DD     00H          ; previous try level for inner block
          DD     FLAT:$L74638 ; inner block filter
          DD     FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT:$T74644
    push   OFFSET FLAT:__except_handler3
    mov    eax, DWORD PTR fs:__except_list
    push   eax
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to 1
```

```
    push    OFFSET FLAT:$SG74617 ; 'hello!'
    call    _printf
    add     esp, 4
    push    0
    push    0
    push    0
    push    1122867 ; 00112233H
    call    DWORD PTR __imp__RaiseException@16
    push    OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level back to 0
    jmp     SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74643[ebp], eax
    mov     eax, DWORD PTR $T74643[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
    inc     eax
$L74640:
$L74648:
    ret     0

; inner block handler:
$L74639:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74621 ; 'access violation, can''t recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level back to 0

$L74615:
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set previous try level back to -1
    jmp     SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74642[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push    ecx
    mov     edx, DWORD PTR $T74642[ebp]
    push    edx
    call    _filter_user_exceptions
    add     esp, 8
$L74636:
$L74649:
    ret     0

; outer block handler:
$L74635:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
```

```
    push    OFFSET FLAT:$SG74623 ; 'user exception caught'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set previous try level back to -1
$L74633:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:__except_list, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main   ENDP
```

只要在 handler 中调用 printf()函数的指令那里设置一个断点，就可以观测到 SEH handler 的添加过程。或许 SEH 的内部处理机制与众不同。而这里我们可以看到 scope table 包含着两个元素：

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

指令清单 68.9 tracer.exe 输出

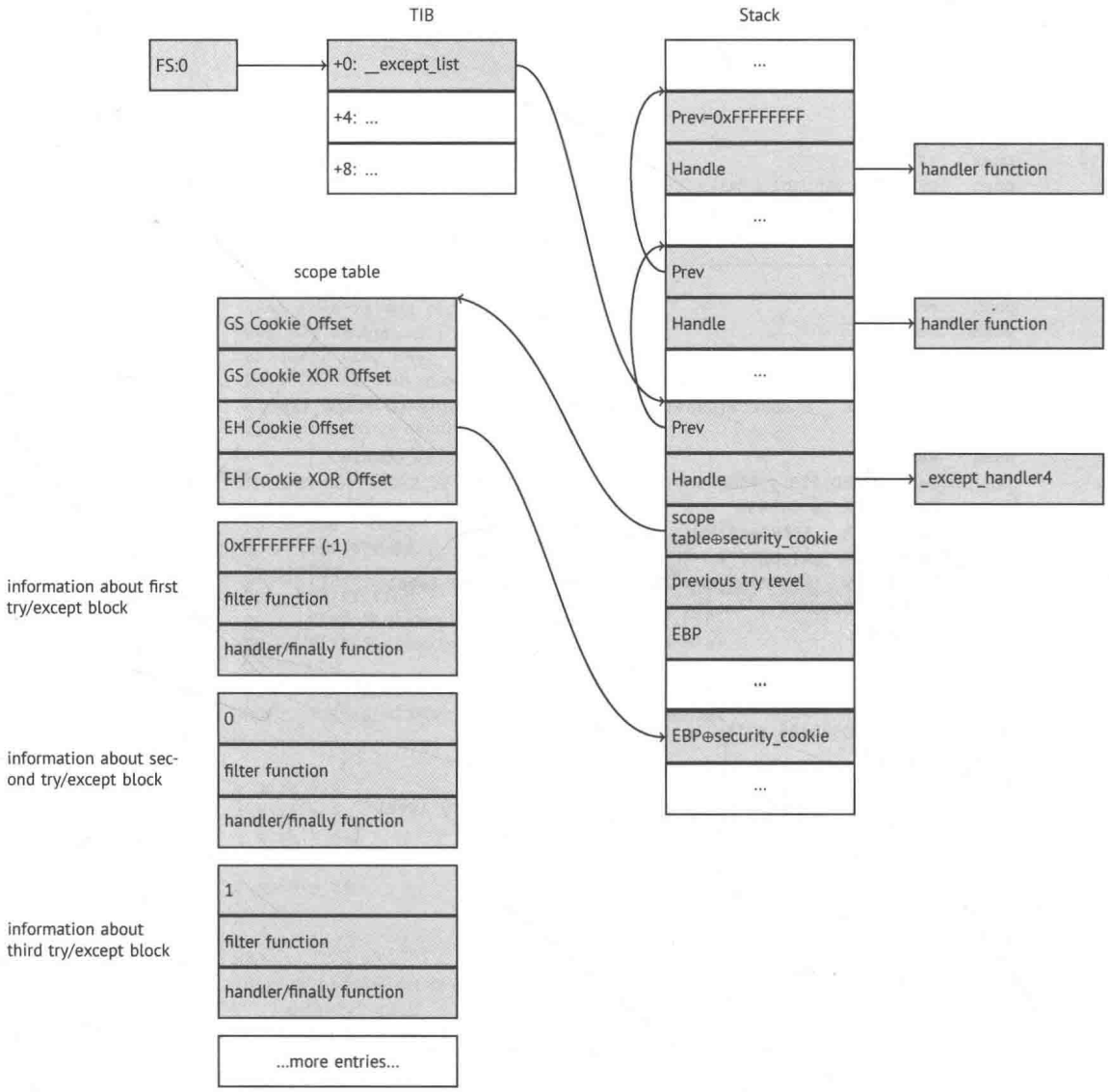
```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b ↙
    ↳ (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100 ↙
    ↳ (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) ↙
    ↳ handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xfffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↙
    ↳ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

SEH4

在遭受缓冲区溢出攻击（请参见本书第 18 章第 2 节）以后，地址表 scope table 的地址可能被重写。MSVC 2005 编译器为 SEH 帧增加了一些缓冲区溢出保护，把 SEH3 升级成了 SEH4。SEH4 的 scope table 表的指针会与 security cookie 进行异或运算，然后才被写到相应的数据结构里。此外 Scope table 新增了一个双指针表头，这两个 EH cookie 指针都是 security cookies 的指针（GS cookies 只有在编译时打开/GS 参数才会出现）。EH Cookie 的偏移量（offset）都是基于栈帧（EBP）的相对地址，其与 security_cookie 的异或运算结果会被保存在栈里，充当校验码。异常处理函数的加载过程会读取这个值并检查其正确性。

由于栈内的 security cookie 是一次性随机值，因此远程攻击者是无法事先预测这个值。

在 SEH4 中最外层的级别（previous try level）是-2，而不是 SEH3 的-1。



这里列出了两个由 MSVC 2012 编译的 SEH4 函数。

指令清单 68.10 MSVC 2012：一个 try 块的例子

```
$SG85485 DB      'hello #1!', 0aH, 00H
$SG85486 DB      'hello #2!', 0aH, 00H
$SG85488 DB      'access violation, can''t recover', 0aH, 00H

; scope table:
xdata$x          SEGMENT
__sehtable$_main DD 0fffffffeH   ; GS Cookie Offset
    DD           00H             ; GS Cookie XOR Offset
    DD           0ffffffccH      ; EH Cookie Offset
    DD           00H             ; EH Cookie XOR Offset
    DD           0fffffffeH      ; previous try level
    DD           FLAT:$LN12@main ; filter
    DD           FLAT:$LN8@main  ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
```

```
_p$ = -32       ; size = 4
tv68 = -28      ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor    eax, ebp
    push   eax                              ; ebp ^ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0

; handler:
$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
```

```
    pop     ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main   ENDP
```

指令清单 68.11 MSVC 2012：两个 try 块的例子

```
$SG85486 DB      'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB      'yes, that is our exception', 0aH, 00H
$SG85490 DB      'not our exception', 0aH, 00H
$SG85497 DB      'hello!', 0aH, 00H
$SG85499 DB      '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB      'access violation, can''t recover', 0aH, 00H
$SG85503 DB      'user exception caught', 0aH, 00H

xdata$x     SEGMENT
__sehtable$_main DD 0ffffffeH           ; GS Cookie Offset
                 DD    00H              ; GS Cookie XOR Offset
                 DD    0ffffffc8H       ; EH Cookie Offset
                 DD    00H              ; EH Cookie Offset
                 DD    0fffffffeH       ; previous try level for outer block
                 DD    FLAT:$LN19@main  ; outer block filter
                 DD    FLAT:$LN9@main   ; outer block handler
                 DD    00H              ; previous try level for inner block
                 DD    FLAT:$LN18@main  ; inner block filter
                 DD    FLAT:$LN13@main  ; inner block handler
xdata$x ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main   PROC
    push    ebp
    mov     ebp, esp
    push    -2 ; initial previous try level
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax ; prev
    add     esp, -24
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR ___security_cookie
    xor     DWORD PTR __$SEHRec$[ebp+16], eax        ; xored pointer to scope table
    xor     eax, ebp                                 ; ebp ^ security_cookie
    push    eax
    lea     eax, DWORD PTR __$SEHRec$[ebp+8]         ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try level=0
    mov     DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try level=1
    push    OFFSET $SG85497 ; 'hello!'
    call    _printf
    add     esp, 4
    push    0
    push    0
    push    0
```

```
    push    1122867 ; 00112233H
    call    DWORD PTR __imp__RaiseException@16
    push    OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level back to 0
    jmp     SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T3[ebp], eax
    cmp     DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne     SHORT $LN5@main
    mov     DWORD PTR tv72[ebp], 1
    jmp     SHORT $LN6@main
$LN5@main:
    mov     DWORD PTR tv72[ebp], 0
$LN6@main:
    mov     eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret     0

; inner block handler:
$LN13@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85501 ; 'access violation, can''t recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try level back to 0
$LN2@main:
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2
    jmp     SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T2[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push    ecx
    mov     edx, DWORD PTR $T2[ebp]
    push    edx
    call    _filter_user_exceptions
    add     esp, 8
$LN10@main:
$LN17@main:
    ret     0

; outer block handler:
$LN9@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85503 ; 'user exception caught'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2
$LN7@main:
    xor     eax, eax
```

```
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:0, ecx
    pop     ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
    _main   ENDP

_code$ = 8 ; size = 4
_ep$ = 12  ; size = 4
_filter_user_exceptions PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call    _printf
    add     esp, 8
    cmp     DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne     SHORT $LN2@filter_use
    push    OFFSET $SG85488 ; 'yes, that is our exception'
    call    _printf
    add     esp, 4
    mov     eax, 1
    jmp     SHORT $LN3@filter_use
    jmp     SHORT $LN3@filter_use
$LN2@filter_use:
    push    OFFSET $SG85490 ; 'not our exception'
    call    _printf
    add     esp, 4
    xor     eax, eax
$LN3@filter_use:
    pop     ebp
    ret     0
_filter_user_exceptions ENDP
```

Cookie Offset 是 EBP 的值（栈帧栈底）与栈内 EBP ⊕ security_cookie 之间的差值。Cookie XOR Offset 是 EBP ⊕ security_cookie 与栈中数值之间的差值。如果上述各值不符合下列条件，那么整个进程将会因为栈损坏而终止运行：

```
security_cookie ⊕ (CookieXOROffset +address_of_saved_EBP) == stack[address_of_saved_EBP + CookieOffset]
```

如果 Cookie Offset 的值是−2，就表示这个 cookie 并不存在（GScookie 一般如此）。

笔者编写的 tracer 程序也能进行 Cookies 的合法性检查，有关详情请访问 https://github.com/dennis714/tracer/blob/master/SEH.c。

在启用“/GS-”选项之后 MSVC 2005 编译器就会分配 SEH3 的函数，但是它依然会分配 SEH4 的 CRT 代码。

68.3.3 Windows x64

和大家想象的一样，每个函数都在序言中设置 SEH 栈帧将会降低运行速度。此外，在程序运行过程中不断调整“previous try level”字段同样会增加时间开销。然而在 x64 程序里，整个情况彻底不同了：所有 try 块的指针、filter 和 handler 函数的指针都单独存储于可执行文件的.pdata 段。操作系统根据.pdata 段获取异常处理的全部信息。

我们把上一个章节的两个程序编译为 x64 程序，可以得到：

指令清单 68.12 MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H
```

```
pdata    SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata    ENDS
pdata    SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
pdata    ENDS
xdata    SEGMENT
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata    ENDS

_TEXT    SEGMENT
main     PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t recover'
        call    printf
        npad    1 ; align next label
$LN8@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main     ENDP
_TEXT    ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x ENDS
```

指令清单 68.13 MSVC 2012

```
$SG86277 DB      'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB      'yes, that is our exception', 0aH, 00H
$SG86281 DB      'not our exception', 0aH, 00H
$SG86288 DB      'hello!', 0aH, 00H
$SG86290 DB      '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB      'access violation, can''t recover', 0aH, 00H
$SG86294 DB      'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD       imagerel $LN6+73
        DD       imagerel $unwind$filter_user_exceptions
$pdata$main DD   imagerel $LN14
        DD       imagerel $LN14+95
        DD       imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD       imagerel main$filt$0+32
        DD       imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD       imagerel main$filt$1+30
        DD       imagerel $unwind$main$filt$1
pdata   ENDS

xdata SEGMENT
$unwind$filter_user_exceptions DD 020601H
        DD       030023206H
$unwind$main DD 020609H
        DD       030023206H
        DD       imagerel __C_specific_handler
        DD       02H
        DD       imagerel $LN14+8
        DD       imagerel $LN14+59
        DD       imagerel main$filt$0
        DD       imagerel $LN14+59
        DD       imagerel $LN14+8
        DD       imagerel $LN14+74
        DD       imagerel main$filt$1
        DD       imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD       050023206H
$unwind$main$filt$1 DD 020601H
        DD       050023206H
Xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14:
        push     rbx
        sub      rsp, 32
        xor      ebx, ebx
        lea      rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call     printf
        xor      r9d, r9d
        xor      r8d, r8d
        xor      edx, edx
        mov      ecx, 1122867 ; 00112233H
        call     QWORD PTR __imp_RaiseException
        lea      rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s crash'
        call     printf
        mov      DWORD PTR [rbx], 13
        jmp      SHORT $LN13@main
$LN11@main:
```

```
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t recover'
        call    printf
        npad    1 ; align next label
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call    printf
        npad    1 ; align next label
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP

text$x SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$:
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1 ; align next label
$LN8@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$1 ENDP
text$x ENDS

_TEXT SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
        call    printf
```

```
        mov     eax, 1
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use:
        lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT ENDS
```

要想查看更多信息，可以参考 Igor Skochinsky 撰写的文章“Compiler Internals:Exceptional and RTTL（编译器内幕:例外与 RTTL）”。

除了例外信息外，.pdata 段还存储着几乎所有函数的起始和结束的地址。由此可见，.pdata 段是自动分析的工具的重点分析对象。

68.3.4 关于 SEH 的更多信息

Igor Skochinsky 编写的文章“Compiler Internals:Exceptional and RTTL（编译器内幕：例外与 RTTL）”。

Matt Pietrek 编写的文章“A Crash Course on the Depths of Win32 Structured Exception Handling（Win32 结构性例外进程的崩溃的深度分析）”。

68.4 Windows NT：临界区段

在任何一个多线程的环境下，临界区段(Critical section)是保护数据一致性和操作互斥性的重要手段。临界区段保证了在同一时间内只会有一个线程访问某些数据，阻止其他进程和中断同期操作相关数据。

在 Windows NT 系统的数据结构中，CRITICAL_SECTION 关键段的定义如下。

指令清单 68.14 (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

下面的代码描述了函数 EnterCriticalSection()的工作原理。

指令清单 68.15 Windows 2008/ntdll.dll/x86（开始）

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr 8
```

```
                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD:
                mov     eax, large fs:18h
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
                pop     ebp
                retn    4

... skipped
```

在代码段中最重要的指令是 BTR（及其 LOCK 前缀）：BTR 指令把第一个操作数的第 0 位复制给 CF 标识位，然后再把这个位清零。由 LOCK 前缀修饰的都是原子性操作，可以让 CPU 阻止其他的系统总线读取或修改相关内存地址。如果 LockCount 的第 0 位值是 1，则将其充值重置并退出函数——CPU 现在正处于临界区；否则，则表示其他线程正在占用临界区，CPU 将等待相关操作结束。

等待期间运行的函数是 WaitForSingleObject()。

下述代码描述了 LeaveCriticalSection()函数的工作机理：

指令清单 68.16 Windows 2008/ntdll.dll/x86（开始）

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr 8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
                lea     edi, [esi+4] ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
```

```
        pop     ebp
        retn    4

... skipped
```

XADD 指令的功能是：先交换操作数的值，然后再进行加法运算。在本例中，它将 LockCount 与数字 1 的和存储于第一个操作数，同时将 LockCount 的初始值传递给 EBX 寄存器。但是 EBX 里的这个值也随即被后面的 INC 指令递增，最终与 LockCount 的值同步。因为它带有 LOCK 前缀，所以属于原子操作。这就意味着所有的其他 CPU（不管是几核的）都不能同时访问那片内存区域。

LOCK 前缀非常重要。不同的 CPU 或者 CPU 核心（core）可能会加载同一个进程的不同线程。若使用无 LOCK 前缀的指令操作临界区段的数据，很可能发生无法预料的情况。

第七部分

常用工具



第 69 章　反汇编工具

69.1　IDA

IDA PRO 简称 IDA（Interactive Disassembler），是一个世界顶级的交互式反汇编工具，由总部位于比利时列日市（Liège）的 Hex-Rayd 公司研发。

Hex-Rayd 为希望了解 IDA 基本功能的用户提供了一个功能有限的免费版本。这个免费版是由 5.0 版精简而来，它的下载地址是：https://www.hex-rays.com/products/ida/support/download_freeware.shtml。

本书的附录 F.1 收录了 IDA 常用的快捷键。

第70章 调试工具

70.1 tracer

我很少使用 debugger，往往用自己研发的 tracer 工具对程序进行跟踪和调试。[①]

近期以来，我完全不用 debugger 了。debugger 就是一个在程序执行期间辨别函数的参数，或者在某个断点查看寄存器状态的工具。每次都使用 debugger 进行调试，未免过于烦琐。所以我就自己编写了 tracer 工具。tracer 程序采用控制台界面，能够从命令行里直接发送命令，同样可以在函数的执行过程中进行中断，并且还能在任意地址设置中断、查看进程状态和修改数据，完成各种各样的任务。

话说回来，在学习和摸索的过程中，初学者还是应当熟悉和掌握 debugger 程序的使用方法，用 debugger 查看寄存器的状态变化[②]，观察标识位、数据，并手动修改各项数据，验证数据对程序的影响。

70.2 OllyDbg

OllyDbg 是一款十分流行的用户模式（user-mode,Ring 3 级）调试程序。它的官方网站是：http://www.ollydbg.de/。

本书的附录 F.2 收录了 OllyDbg 常用的快捷键。

70.3 GDB

GDB 不是一款图形化调试器，因而不太受逆向工程研究人员关注。但是它的功能更为强大，可谓独具特色。

本书的附录 F.5 收录了部分 GDB 常用指令。

---

① tracer 工具的下载地址是 http://yurichev.com/tracer-en.html。

② 经典的 SoftICE、OllyDbg 和 WinDbg 工具能够用高亮信息提示寄存器变化。

第 71 章　系统调用的跟踪工具

71.1　strace/dtruss

Linux 内核提供了一款非常有用的调试工具，它可以跟踪某个进程调用的系统调用（以及该进程所接收到的信号）[①]。这款工具就是 strace。它属于命令行工具，在使用时，我们可以把希望跟踪的应用程序直接指定为命令参数。例如：

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X 的 dtruss 工具与 Linux 下的 strace 功能相同。

Cygwin 环境里同样也有 strace 程序。如果我没搞错的话，Cygwin 里的 strace 只能分析那些在 Cygwin 环境里编译出来的.exe 文件。

① 有关 syscalls 的详细介绍，请参见本书第 66 章。

第 72 章　反编译工具

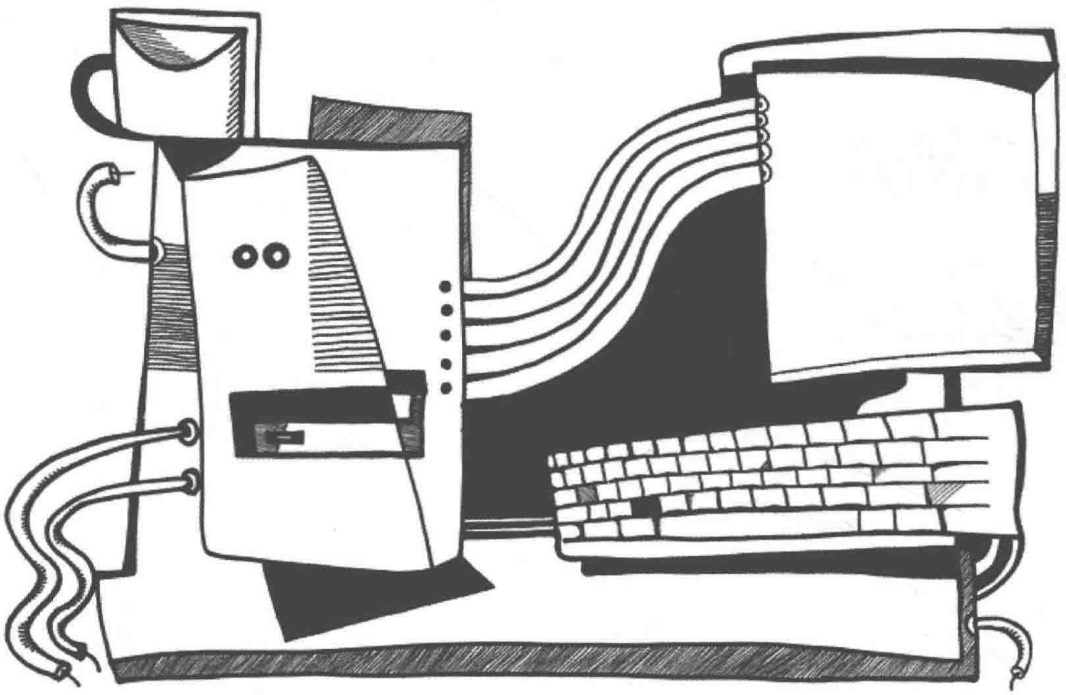
Hex-Rays Decompiler 是唯一一款众所皆知的、公开销售的、品质较高的 C 语言反编译工具。它的官方网站是：https://www.hex-rays.com/products/decompiler/。

第73章 其他工具

- Microsoft Visual Studio Express: 鼎鼎大名的 Visual Studio 的精简版，可用于简单的程序调试。本书的附录 F.3 收集了它的部分选项及功能。其官方网站是：https://www.visualstudio.com/en-US/products/visual-studio-express-vs。
- Hiew2：一款优秀的 16 进制编辑器，可以对应用程序进行反汇编，而且支持对可执行文件的 16 进制代码及汇编语言代码的修改，使用起来非常方便。其官方网站是：http://www.hiew.ru/。
- Binary grep：在海量文件（包括非可执行程序）中搜索常量或任意字节序列的可执行工具。其官方网站是：https://github.com/yurichev/bgrep。

第八部分

更多范例



第 74 章　修改任务管理器（Vista）

如果主机上安装的是四核 CPU，那么 Windows 任务管理器应当显示出 4 个 CPU 的性能统计图表。本章将要稍微 hack 任务管理器，让它显示更多的 CPU 运算核心。

首先需要了解的问题是：任务管理器如何知道 CPU 有多少个运算核心？虽然运行于 win32 用户空间的 GetSystemInfo()函数确实可以反馈这一信息，但是任务管理器 taskmgr.exe 没有直接导入这个函数。它多次调用 NTAPI 中的 NtQuerySystemInformation()函数获取的各种系统信息，也是通过后者了解 CPU 的具体情况。

NtQuerySystemInformation()函数有四个参数：第一个参数是查询的系统信息类型[①]；第二个参数是一个指针，这个指针用来返回系统的 HandleList；第三个参数是程序员指定分配给 HandleList 的内存空间大小；第四个参数是 NtQuerySystemInformation 返回的 HandleList 的大小。

要获取 CPU 信息，就要在调用它的时候把第一个参数设置为常量 SystemBasicInformation[②]。

因此，我们要查找的调用指令大体会是“NtQuerySystemInformation(0, ?, ?, ?)”。第一步当然是用 IDA 打开 taskmgr.exe 文件。在处理微软的官方程序时，IDA 能够下载与之相应的 PDB 文件并显示全部函数名称。显而易见的是，任务管理器是用 C++编写的程序，而且它使用的函数名称和类（class）名称真的是不为人知。在它使用的类名称里，我们可以看到 CAdapter、CNetPage、CPerfPage、CProcInfo、CProcPage、CSvcPage、CTaskPage 和 CUserPage。这些类名称和任务管理器程序窗口的标签（tab）有对应关系。

在跟踪了 NtQuerySystemInformation()函数的每次调用过程之后，我们可以统计出传递给函数的第一个参数。在图 74.1 中可以看到，部分调用过程里的第一个参数值明显不是零，所以被标记上了“Not Zero”。另外，还有一些函数调用的情况非常特殊，本章的第二部分再进行有关讲解。总之，我们要找那些“第一个参数是零”的、NtQuerySystemInformation()函数的调用语句。

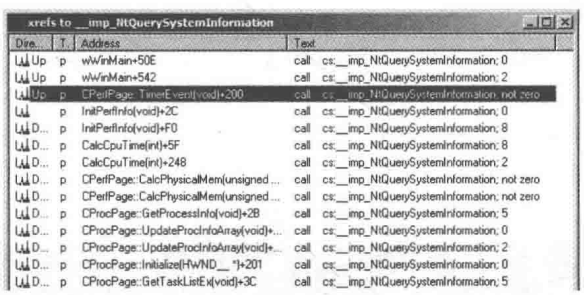


图 74.1　IDA：NtQuerySystemInformation()函数的 xrefs

那些不公开的名称暂且放置一边。

在检索“NtQuerySystemInformation(0, ?, ?, ?)”的调用语句时，我们可以很快地在 InitPerfInfo()里找到如下所示的这种语句。

指令清单 74.1　taskmgr.exe (Windows Vista)

```
.text:10000B4B3          xor     r9d, r9d
.text:10000B4B6          lea     rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB          xor     ecx, ecx
.text:10000B4BD          lea     ebp, [r9+40h]
```

① 后面提到的 HandleList 就是函数应当反馈的类型信息。

② 这个常量的值为零。更多信息请参考 MSDN: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724509(v=vs.85).aspx。

```
.text:10000B4C1          mov     r8d, ebp
.text:10000B4C4          call    cs:__imp_NtQuerySystemInformation ; 0
.text:10000B4CA          xor     ebx, ebx
.text:10000B4CC          cmp     eax, ebx
.text:10000B4CE          jge     short loc_10000B4D7
.text:10000B4D0
.text:10000B4D0 loc_10000B4D0:                          ; CODE XREF: InitPerfInfo(void)+97
.text:10000B4D0                                         ; InitPerInfo(void)+AF
.text:10000B4D0          xor     al, al
.text:10000B4D2          jmp     loc_10000B5EA
.text:10000B4D7 ; ---------------------------------------------------------------------------
.text:10000B4D7
.text:10000B4D7 loc_10000B4D7:                          ; CODE XREF: InitPerfInfo(void)+36
.text:10000B4D7          mov     eax, [rsp+0C78h+var_C50]
.text:10000B4DB          mov     esi, ebx
.text:10000B4DD          mov     r12d, 3E80h
.text:10000B4E3          mov     cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:10000B4E9          shr     eax, 0Ah
.text:10000B4EC          lea     r13, __ImageBase
.text:10000B4F3          imul    eax, [rsp+0C78h+var_C4C]
.text:10000B4F8          cmp     [rsp+0C78h+var_C20], bpl
.text:10000B4FD          mov     cs:?g_MEMMax@@3_JA, rax ; __int64 g_MEMMax
.text:10000B504          movzx   eax, [rsp+0C78h+var_C20] ; no. of CPUs
.text:10000B509          cmova   eax, ebp
.text:10000B50C          cmp     al, bl
.text:10000B50E          mov     cs:?g_cProcessors@@3EA, al ; uchar g_cProcessors
```

从微软的服务器上下载相应的 PDB 文件之后，IDA 就能够给各个变量分配正确的变量名称。我们不难从中找到全局变量 g_cProcessors。

传递给 NtQuerySystemInformation()函数的第二个参数（即接收缓冲区）是 var_C58。var_C20 和 var_C58 之间的地址差值是 0xC58−0xC20=0x38(56)。根据 MSDN 的官方说明，可知返回值的数据格式如下：

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

因为本例是在 x64 系统上的演示，所以 PVOID 占用 8 个字节。两个“reserved”保留字段共占用 24+4×8=56 字节。这意味着 var_C20 很可能就是_SYSTEM_BASIC_INFORMATION 里的 NumberOfProcessors 字段。

下面我们来验证这一推论。把 C:\Windows\System32 里的 taskmgr.exe 复制出来，然后我们在对复制品进行修改，以防 Windows 的文件保护机制自动恢复原始文件。

使用 Hiew 打开复制出来的文件，然后找到图 74.2 所示的程序地址。



图 74.2 Hiew：找到修改点

接下来替换 MOVZX 指令，通过 MOV 指令直接把返回结果改为 64（将 CPU 设为 64 核）。由于修改后的指令比原始指令短 1 个字节，所有我们还需添加 1 个 NOP 指令。如图 74.3 所示。



图 74.3 Hiew：修改程序

修改后的这个程序可以正常运行！当然，图表中的统计信息肯定是不正确的。CPU 的总负载偶尔还会超过 100%。如图 74.4 所示。

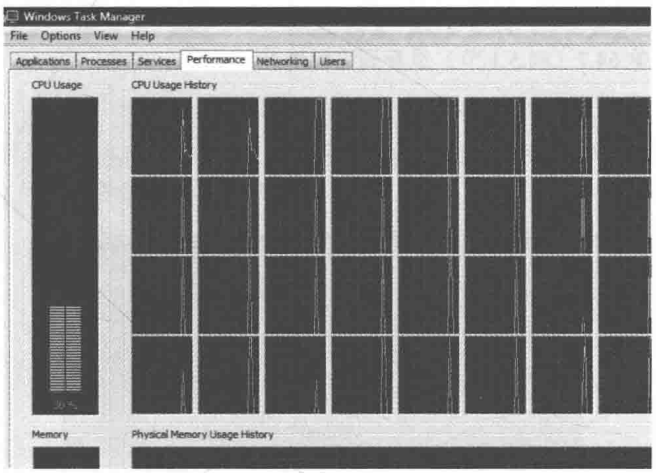


图 74.4　被骗的 Windows 任务管理器

刚才我们把 CPU 运算核心的总数改为了 64（更大的值会使任务管理器崩溃）。显然 Windows Vista 的任务管理器无法在拥有更多运算核心的计算机上运行。这也可能是微软通过静态的数据结构把有关数值限定在 64 以下的原因。

74.1　使用 LEA 指令赋值

任务管理器 taskgmr.exe 传递 NtQuerySystemInformation()第一个参数的指令并非都是 MOV 指令，部分指令是 LEA。

指令清单 74.2　taskmgr.exe (Windows Vista)

```
                xor     r9d, r9d
                div     dword ptr [rsp+4C8h+WndClass.lpfnWndProc]
                lea     rdx, [rsp+4C8h+VersionInformation]
                lea     ecx, [r9+2]     ; put 2 to ECX
                mov     r8d, 138h
                mov     ebx, eax
; ECX=SystemPerformanceInformation
                call    cs:__imp_NtQuerySystemInformation ; 2
                ...

                mov     r8d, 30h
                lea     r9, [rsp+298h+var_268]
                lea     rdx, [rsp+298h+var_258]
                lea     ecx, [r8-2Dh]    ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
                call    cs:__imp_NtQuerySystemInformation ; not zero

                ...

                mov     rbp, [rsi+8]
                mov     r8d, 20h
                lea     r9, [rsp+98h+arg_0]
                lea     rdx, [rsp+98h+var_78]
                lea     ecx, [r8+2Fh]   ; put 0x4F to ECX
                mov     [rsp+98h+var_60], ebx
                mov     [rsp+98h+var_68], rbp
```

```
; ECX=SystemSuperfetchInformation
                call    cs:__imp_NtQuerySystemInformation ; not zero
```

出现这种指令的具体原因不明。但是 MSVC 编译器还是经常如此分配指令。或许 LEA 指令会带来速度或性能方面的好处吧。

您还可以在指令清单 64.7（64.5.1 节）里看到这种情况。

第 75 章　修改彩球游戏

彩球游戏有多个衍生版本。本章采用的是 1997 年发布的 BallTrix 版。这款程序可从 http://go.yurichev.com/17311 公开下载。它的图形界面如图 75.1 所示。

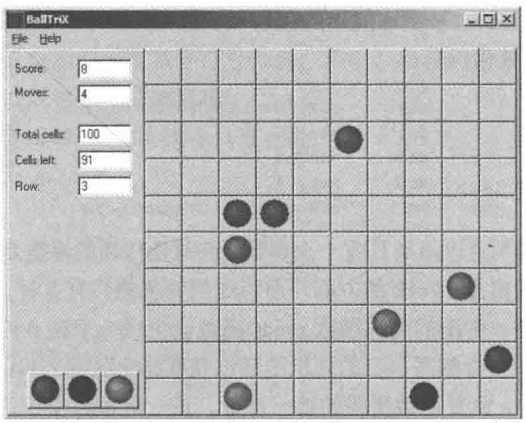


图 75.1　游戏界面

本章关注它的随机生成器，以及修改这个组件的具体方法。IDA 在 balltrix.exe 里识别出了标准函数 _rand。这个函数的地址是 0x00403DA0。不仅如此，IDA 还判断出该函数只会被这一处调用。

```
.text:00402C9C sub_402C9C       proc near               ; CODE XREF: sub_402ACA+52
.text:00402C9C                                          ; sub_402ACA+64 ...
.text:00402C9C
.text:00402C9C arg_0            = dword ptr  8
.text:00402C9C
.text:00402C9C                  push    ebp
.text:00402C9D                  mov     ebp, esp
.text:00402C9F                  push    ebx
.text:00402CA0                  push    esi
.text:00402CA1                  push    edi
.text:00402CA2                  mov     eax, dword_40D430
.text:00402CA7                  imul    eax, dword_40D440
.text:00402CAE                  add     eax, dword_40D5C8
.text:00402CB4                  mov     ecx, 32000
.text:00402CB9                  cdq
.text:00402CBA                  idiv    ecx
.text:00402CBC                  mov     dword_40D440, edx
.text:00402CC2                  call    _rand
.text:00402CC7                  cdq
.text:00402CC8                  idiv    [ebp+arg_0]
.text:00402CCB                  mov     dword_40D430, edx
.text:00402CD1                  mov     eax, dword_40D430
.text:00402CD6                  jmp     $+5
.text:00402CDB                  pop     edi
.text:00402CDC                  pop     esi
.text:00402CDD                  pop     ebx
.text:00402CDE                  leave
.text:00402CDF                  retn
.text:00402CDF sub_402C9C       endp
```

为了便于讨论，我们把_rand 函数的调用方函数叫作“random”。在程序里有三处的代码调用了 random 函数。调用 random 函数的前两处代码是：

```
.text:00402B16                mov     eax, dword_40C03C ; 10 here
.text:00402B1B                push    eax
.text:00402B1C                call    random
.text:00402B21                add     esp, 4
.text:00402B24                inc     eax
.text:00402B25                mov     [ebp+var_C], eax
.text:00402B28                mov     eax, dword_40C040 ; 10 here
.text:00402B2D                push    eax
.text:00402B2E                call    random
.text:00402B33                add     esp, 4
```

调用 random 函数的第三处代码是：

```
.text:00402BBB                mov     eax, dword_40C058 ; 5 here
.text:00402BC0                push    eax
.text:00402BC1                call    random
.text:00402BC6                add     esp, 4
.text:00402BC9                inc     eax
```

综合上述代码，我们可以判定该函数只有一个参数。前两处传递的参数是 10，第三处传递的参数是 5。在观察游戏的界面后，可知棋盘是 10×10 的方阵，而彩球的颜色总共有 5 种。这三处调用 random 的指令，必定是坐标和颜色的生成指令。标准的随机函数 rand()函数会生成一个在 0～0x7FFF 之间的返回值，用起来并不方便。实际上，编程人员会编写自己的随机函数以获取特定区间之内的随机返回值。本例需要的随机数是 0～(n−1)之间的整数，n 就是函数所需的唯一参数。这一假设可由任意一种 debugger 验证。

本章将修改第三处调用指令，让它的第三次返回值永远是 0。为此，我们可把 PUSH/CALL/ADD 这三条指令改为 NOP，然后再添加 XOR EAX,EAX 指令，以清空 EAX 寄存器。

```
.00402BB8: 83C410          add         esp,010
.00402BBB: A158C04000      mov         eax,[00040C058]
.00402BC0: 31C0            xor         eax,eax
.00402BC2: 90              nop
.00402BC3: 90              nop
.00402BC4: 90              nop
.00402BC5: 90              nop
.00402BC6: 90              nop
.00402BC7: 90              nop
.00402BC8: 90              nop
.00402BC9: 40              inc         eax
.00402BCA: 8B4DF8          mov         ecx,[ebp][-8]
.00402BCD: 8D0C49          lea         ecx,[ecx][ecx]*2
.00402BD0: 8B15F4D54000    mov         edx,[00040D5F4]
```

也就是说，我们修改调用 random()函数的有关指令，让它的返回值固定为 0。

修改程序后，它的运行界面如图 75.2 所示。

不得不说我们修改得很成功。我在当初修改这个游戏的时候，希望我的同事明白“没有必要执迷于你肯定会赢的游戏”。可惜我的劝阻没能成功。

另外还有一个问题：为什么 random()函数的参数会是全局变量？这是因为棋盘大小是可调整的变量，不能在程序里把它写成常量。本例中的 5 和 10 只是它的默认值。

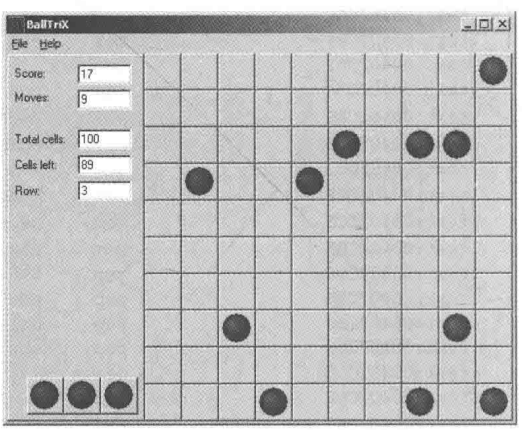


图 75.2 作弊成功

第 76 章　扫雷（Windows XP）

我的扫雷水平不高，所以干脆用 debugger 把雷都显示出来吧！

因为地雷的具体位置是随机的，所以扫雷程序里肯定会用随机函数安置地雷。这种随机函数不是某种自制的随机数生成函数，就是标准的 C 函数 rand()。最为美妙的事情是，微软不仅公开了其产品的 PDB 文件，而且在 PDB 文件里提供了全部的函数名等符号信息。所以，当我们用 IDA 打开 winmine.exe 程序时，它会从微软下载 PDB 文件并且显示所有函数名称。

在 IDA 里可以看到，调用 rand()函数的指令只有一处：

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4          proc near               ; CODE XREF: StartGame()+53
.text:01003940                                         ; StartGame()+61
.text:01003940
.text:01003940 arg_0           = dword ptr 4
.text:01003940
.text:01003940                 call    ds:__imp__rand
.text:01003946                 cdq
.text:01003947                 idiv    [esp+arg_0]
.text:0100394B                 mov     eax, edx
.text:0100394D                 retn    4
.text:0100394D _Rnd@4          endp
```

IDA 把这个函数显示为 rnd() 函数，那就是说扫雷游戏的开发人员给它起的名字就是 rnd。这个函数非常简单：

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

微软的 PDB 文件没有把参数命名为 limit。为了便于讨论，本文给它起名为 limit。可见，rnd()函数返回值是介于 0～limit 之间的整数。

rand()函数的调用方函数也只有一个 StartGame()函数。而且 StartGame()函数应当就是安放地雷的函数：

```
.text:010036C7                 push    _xBoxMac
.text:010036CD                 call    _Rnd@4          ; Rnd(x)
.text:010036D2                 push    _yBoxMac
.text:010036D8                 mov     esi, eax
.text:010036DA                 inc     esi
.text:010036DB                 call    _Rnd@4          ; Rnd(x)
.text:010036E0                 inc     eax
.text:010036E1                 mov     ecx, eax
.text:010036E3                 shl     ecx, 5          ; ECX=ECX*32
.text:010036E6                 test    _rgBlk[ecx+esi], 80h
.text:010036EE                 jnz     short loc_10036C7
.text:010036F0                 shl     eax, 5          ; EAX=EAX*32
.text:010036F3                 lea     eax, _rgBlk[eax+esi]
.text:010036FA                 or      byte ptr [eax], 80h
.text:010036FD                 dec     _cBombStart
.text:01003703                 jnz     short loc_10036C7
```

因为扫雷游戏允许用户设置棋盘大小，所以棋盘的 X(xBoxMac)和 Y(yBoxMac)都是全局变量。Rnd()函数根据这两个参数生成随机坐标，而后 0x10036FA 处的 OR 指令设置地雷。如果这个坐标在以前已经设置过地雷了，那么 0x010036E6 的 TEST 和 JNZ 指令将再次生成一次坐标。

变量 cBombStart 不仅是设置地雷总数的全局变量，还是循环控制变量。

SHL/左移指令意味着棋盘宽度是 32。

全局数组 rgBlk 的容量可通过数据段里 rgBlk 标签的地址与下一个数据的地址推算出来。数组容量应当是这两个地址之间的差值，即 0x360（864）。

```
.data:01005340 _rgBlk              db 360h dup(?)        ; DATA XREF: MainWndProc(x,x,x,x)+574
.data:01005340                                           ; DisplayBlk(x,x)+23
.data:010056A0 _Preferences        dd ?                  ; DATA XREF: FixMenus()+2
...
```

数组的元素数量为：864（总容量）/32＝27。

那么，rgBlk 是否就是 27×32 的数组呢？当我们把棋盘设置为 100×100 的矩阵时，它会自动回滚为 24×30 的棋盘。所以棋盘盘面的最大值就是这个值，而且无论棋盘有多大，程序都把棋盘数据存储在这个数组里。

接下来，我们使用 OllyDbg 进行观察。在 OllyDbg 中运行扫雷游戏，然后在内存窗口里观察 rgBlk 数组（地址为 0x1005340）。①

与这个数组有关的内存数据如下：

```
Address   Hex dump
01005340  10 10 10 10|10 10 10 10|10 10 10 0F|0F 0F 0F 0F|
01005350  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005360  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005370  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005380  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005390  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053A0  10 0F 0F 0F|0F 0F 0F 0F|8F 0F 10 0F|0F 0F 0F 0F|
010053B0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053C0  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
010053D0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053E0  10 0F 0F 0F|0F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
010053F0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005400  10 0F 0F 8F|0F 0F 8F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005410  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005420  10 8F 0F 0F|8F 0F 0F 0F|0F 0F 10 0F|0F 0F 0F 0F|
01005430  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005440  10 8F 0F 0F|0F 0F 8F 0F|0F 8F 10 0F|0F 0F 0F 0F|
01005450  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005460  10 0F 0F 0F|0F 8F 0F 0F|0F 8F 10 0F|0F 0F 0F 0F|
01005470  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
01005480  10 10 10 10|10 10 10 10|10 10 10 0F|0F 0F 0F 0F|
01005490  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054A0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054B0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054C0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
```

与其他的 16 进制编辑程序相似，OllyDbg 也采取了每行 16 字节的显示风格。因此，一个 32 字节的数组对应着 OllyDbg 窗口里的两行数据。

启动程序的时候，我们把游戏设置为了“入门级”难度，所以棋盘大小是 9×9。现在我们可以在每行 0×10 个字节的数据里观测到这种正方形结构。

接下来在 OllyDbg 单击“Run”以运行扫雷程序，然后随意点击、直到触碰地雷为止。此时即可看到棋盘中的全部地雷了，如图 76.1 所示。

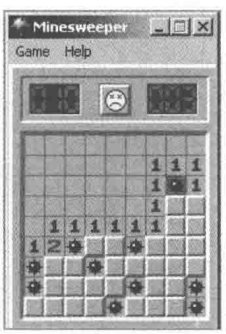


图 76.1 地雷

在比较内存数据之后，我们可得出下列结论：

- 0x10 代表边界。
- 0x0F 代表空白地段。
- 0x8F 代表雷区。

现在我们可对内存数据进行标注了。然后我们再用方括号标注地雷：

① 本章以英文版 Windows XP SP3 中的扫雷游戏为例。如果调试的是其他版本的扫雷游戏，那么内存地址会与本例不同。

```
border:
01005340  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #1:
01005360  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #2:
01005380  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #3:
010053A0  10 0F 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F
010053B0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #4:
010053C0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #5:
010053E0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #6:
01005400  10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F
01005410  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #7:
01005420  10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005430  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #8:
01005440  10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F
01005450  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #9:
01005460  10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F
01005470  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
border:
01005480  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
```

把所有的边界数据（0x10）去除，即可得到地雷的确切位置：

```
 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F[8F]0F
 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F 0F 0F 0F 0F 0F 0F
 0F 0F[8F]0F 0F[8F]0F 0F 0F
[8F]0F 0F[8F]0F 0F 0F 0F
[8F]0F 0F 0F 0F[8F]0F 0F[8F]
 0F 0F 0F 0F[8F]0F 0F 0F[8F]
```

上述数据的行和列与棋盘的相应信息一一对应。

在推导出数据结构之后，在 OllyDbg 里修改数据的尝试更为有趣。如果把所有的 0x8F 都替换成 0x0F，那么扫雷游戏就可以如图 76.2 所示这样玩。

我们还可以把地雷都安置在第一行里，如图 76.3 所示。

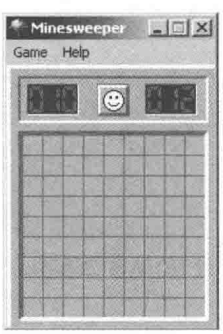


图 76.2　没有地雷的扫雷游戏

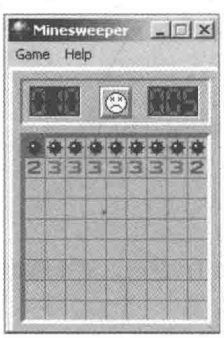


图 76.3　用 debugger 设置地雷

不过，在玩游戏之前，用 OllyDbg 之类的 debugger 查看地雷分布毕竟不够方便。我们不妨写一个专用程序，专门导出棋盘上的地雷分布情况：

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
        int i, j;
        HANDLE h;
        DWORD PID, address, rd;
        BYTE board[27][32];

        if (argc!=3)
        {
                printf ("Usage: %s <PID><address>\n", argv[0]);
                return 0;
        };

        assert (argv[1]!=NULL);
        assert (argv[2]!=NULL);

        assert (sscanf (argv[1], "%d", &PID)==1);
        assert (sscanf (argv[2], "%x", &address)==1);
        h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);

        if (h==NULL)
        {
                DWORD e=GetLastError();
                printf ("OpenProcess error: %08X\n", e);
                return 0;
        };

        if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd)!=TRUE)
        {
                printf ("ReadProcessMemory() failed\n");
                return 0;
        };

        for (i=1; i<26; i++)
        {
                if (board[i][0]==0x10 && board[i][1]==0x10)
                        break; // end of board
                for (j=1; j<31; j++)
                {
                        if (board[i][j]==0x10)
                                break; // board border
                        if (board[i][j]==0x8F)
                                printf ("*");
                        else
                                printf (" ");

                };
                printf ("\n");
        };

        CloseHandle (h);
};
```

指定扫雷游戏的 PID[①]之后，这个程序将会导出 0x01005340 处[②]的地雷分布图。

① PID 即 Program/process ID。Windows 的任务管理器能够查看程序的 PID。
② 不同版本的程序，其地址会发生变化。

上述程序可以把自身绑定到 PID 指定的程序上，然后读取指定程序的数据。

76.1　练习题

- 为什么扫雷游戏里有边界字节 0x10？既然程序不会显示这些数据，那么为何还要保留这些数据？去除这些边界字节会发生什么情况？
- 棋盘上的每个点可被赋予不同的值，以表示“被点击过”“被用户插上棋子”等信息。请找出各个值的具体涵义。
- 请修改本章的最后一个程序，让它以固定的格局分布地雷。
- 请修改本章的最后一个程序，使它在没有 PDB 文件、也不使用预定地址的情况下，自动导出地雷分布图。在扫雷程序运行期间，程序可以在数据段里自动地找到棋盘数据。有关信息可参见附录 G.5.1。

第 77 章　人工反编译与 Z3 SMT 求解法

非专业的密码算法通常都很脆弱。如果密码学专家出手，这些算法将不堪一击。不过，即使密码学专业人士的帮忙，逆向工程分析人员同样可破解密码。

我曾经遇到过一个把 64 位数据转换为另一种数据的单向函数[①]。那时，我们要把 hash 值还原为原始数据。

77.1　人工反编译

在 IDA 中，程序的具体指令如下：

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

ECX 寄存器传递函数的第一个参数，由此可判断这是 GCC 编译的程序。

如果您手头没有 Hex-Rays 一类的反编译程序，或者您根本信不过这些自动化工具，那么您可以自己进行反编译。在进行反编译时，可以把 CPU 寄存器当作 C 语言的变量，然后把汇编语言直接翻译为等效的 C 指令。例如，上述程序可反编译为：

```
uint64_t f(uint64_t input)
{
        uint64_t  rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
```

① 单向函数的有关概念，可参见本书第 34 章的介绍。

```
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

力图谨慎的人，可以把上述代码再次编译为可执行程序。在工作方式上，它应当与最初的程序完全一致。

然后，我们根据寄存器的使用方法，整理刚才写出的 C 代码。这时就需要加倍小心、高度集中注意力，任何细小的纰漏都可能让我们前功尽弃。

首先添加上注释、进行段落划分：

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

接下来整理程序尾部的数学计算指令：

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;
```

```
        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

根据乘法因子的特点，我们应能判断出程序是通过乘法指令等效实现的除法运算。[①]因此，我们可以使用Wolfram Mathematica来计算除数。

指令清单 77.1 Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

那么，原始的运算指令应当是：

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

整理计算指令得：

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
```

① 可参见本书第41章。

```
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

继续简化代码可发现，程序计算的是余数、而非商：

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

最终，我们把程序转化成华丽的 C 语言代码：

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

除了密码专家之外，没什么人能够根据 hash 值逆推原始数据。旋转位左/右移指令足以令人望而却步——它能够保证映射函数不是单满射函数，而且还保留了碰撞的可能性；说得直白一些就是“多个输入可能产生同一个输出”。

由于这个函数采用了 64 位因子，所以暴力破解也不太现实。

77.2　Z3 SMT 求解法

在加密学知识不足的情况下，我们可以使用微软研究团队发布的 Z3 工具[①]尝试破解。虽然它只是个形

① http://z3.codeplex.com/。

式验证工具，但是我们将用它来作 SMT 求解。也就是说，我们要用 Z3 来求解巨型方程式。

我们使用的 Python 源代码如下：

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

程序的第 7 行声明了各个变量。这些都是 64 位变量。其中，i1～i6 都是中间变量（形参），在各个指令之间传递寄存器的值。

第 10～15 行之间是我们添加的约束条件。这些条件之中，第 17 行限定的约束条件最为重要：在使用这个函数时，我们要查找输出值为 10816636949158156260 的输入值。

本质上说，基于 SMT 的求解方法可以搜索满足全部限定条件的所有输入值。

上述程序中的 RotateRight、RotateLeft 和 URem 都是 Z3 提供的 Python API。它们都不是 Python 语言提供的标准指令。

然后运行上述程序：

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

程序输出中的“sat”是“satisfiable（满足条件的值）”的缩写。这就是说，我们的求解方法至少可以找到一个解。程序用方括号把最终解标注了出来。屏幕输出的最后两行是用 16 进制显示的输入、输出值。如果把 0x12EE577B63E80B73 代入原函数的输入变量，那么它的输出值与我们指定的值相符。

另外需要注意的是，因为原函数不是单满射函数，所以可能存在多个符合条件的输入值。不过，网上公开的 Z3 SMT 求解程序只会计算出一组解。为此，我们对上面的程序稍加修改，添加了第 19 行，让程序“探寻其他的解”：

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
```

```
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

s.add(inp!=0x12EE577B63E80B73)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

这样一来，它就可以求得另一组解：

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
 outp=0x961C69FF0AEFD7E4
```

人工排除已知解的方法不太先进。其实程序可以自动地修改约束条件，并且自行排除已知解，以便自动化地求得所有解。自行求得全部解的程序十分精巧：

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

# copypasted   from http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation
result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not supported")
```

```
32              # create a constant from declaration
33              c=d()
34              if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
35                  raise Z3Exception("arrays and uninterpreted sorts are not supported")
36              block.append(c != m[d])
37          s.add(Or(block))
38      else:
39              print "results total=",len(result)
40              break
```

运行上述程序，可得：

```
1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16
```

可见，总共有 16 个输入值满足条件“输出值为 0x92EE577B63E80B73”。

第二个解是 1234567890。在编写本文时，笔者使用的输入值正是这个数。

接下来，我们要更深入地讨论程序的算法。在这些解里面，有没有低 32 位与输出值的低 32 位相等的解？为此，我们修改本章的第一个脚本程序，对第 17 行的限定条件进行修改：

```
 1 from z3 import *
 2
 3 C1=0x5D7E0D1F2E0F1F84
 4 C2=0x388D76AEE8CB1500
 5 C3=0xD2E9EE7E83C4285B
 6
 7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8
 9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

上述程序证明，确实存在这种解：

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
```

```
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

在此基础上，我们再添加一个约束条件——验证是否存在“最后 16 位是 0x1234”的解：

```
 1 from z3 import *
 2
 3 C1=0x5D7E0D1F2E0F1F84
 4 C2=0x388D76AEE8CB1500
 5 C3=0xD2E9EE7E83C4285B
 6
 7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8
 9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18 s.add(outp & 0xFFFF == 0x1234)
19
20 print s.check()
21 m=s.model()
22 print m
23 print (" inp=0x%X" % m[inp].as_long())
24 print ("outp=0x%X" % m[outp].as_long())
```

即使有如此苛刻的约束条件，程序仍然算出了一个解：

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 函数的破解速度非常快。这说明原始算法十分脆弱，根本算不上加密算法。业余人员自制的算法多数都是这样。

这种方法是否可以破解那些真正专业的加密算法呢？虽然像 AES、RSA 这样的加密算法都同样属于大规模方程组，但是其计算规模非常之大，以至于未来几年的计算机系统都不可能对其进行破解。当然，加密学专家非常清楚这件事。

总而言之，SMT/SAT 求解程序（例如 Z3）可以处理非专业的加密算法。

另外，我还写过一篇与 Z3 有关的博客。有兴趣的读者可查阅参考文献[Yur12]。

第78章 加 密 狗

78.1 例1：PowerPC 平台的 MacOS Classic 程序

本例研究的是一款运行在 PowerPC 平台上的 MacOS Classic 程序。研发这款程序的公司早已经不知去向，所以买家十分害怕加密狗出什么意外。

在不插入加密狗的情况下，程序会显示“Invalid Security Device”信息。非常幸运的是，这个字符串就在程序的可执行文件里。

虽然那时我即不熟悉 Mac OS Classic 系统，也没怎么用过 PowerPC，但是还是放手一搏。

IDA 可以毫无困难地打开这个程序。它判断该文件类型为“PEF (Mac OS or Be OS executable)”。标准的 Mac OS Classic 的程序文件的确采用了这种文件格式。

接下来，在文件里搜索错误信息的字符串时，我找到了下述指令：

```
...

seg000:000C87FC 38 60 00 01              li      %r3, 1
seg000:000C8800 48 03 93 41              bl      check1
seg000:000C8804 60 00 00 00              nop
seg000:000C8808 54 60 06 3F              clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40              bne     OK
seg000:000C8810 80 62 9F D8              lwz     %r3, TC_aInvalidSecurityDevice

...
```

这些都是 PowerPC 平台的指令。这款 CPU 是 20 世纪 90 年代出产的一款典型的 32 位 RISC CPU。它的每条指令都占用 4 个字节（与 MIPS 和 ARM 的指令相似），指令名称还与 MIPS 指令相似。

为了便于演示，我把函数名称改为 check1()。BL 是 Brach Link 指令，常用于调用子函数。上述程序的关键点是 BNE 指令，在程序通过了加密狗认证的情况下进行跳转，否则就会在 r3 寄存器里加载字符串，然后报错。

在参阅了参考文献［SK95］之后，我发现 r3 寄存器用于存储返回值。如果返回值是 64 位数据，那么 r4 寄存器也会用于回传返回值。

另外，CLRLWI 指令[①]还是当时的盲点。后来我阅读了参考文献［IBM00］，获悉它是清除和传递数据的复合指令。本例的这个指令会清除 r3 寄存器的高 24 位，把结果存储在 r0 寄存器里。所以，它不仅相当于 x86 的 MOVZX 指令（可参见本书 15.1.1 节），而且还能设置相应标识位，向后面的 BNE 指令传递标识信息。

接下来，我们探索一下 check 1()函数：

```
seg000:00101B40                  check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40                          # sub_64070+160p ...
seg000:00101B40
seg000:00101B40                  .set arg_8, 8
seg000:00101B40
seg000:00101B40 7C 08 02 A6              mflr    %r0
seg000:00101B44 90 01 00 08              stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0              stwu    %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39              bl      check2
seg000:00101B50 60 00 00 00              nop
```

① CLRLWI 是 Clear left word immediate 的缩写，用于清除指定的高/前 *n* 位，再把结果复制到目标操作符。

```
seg000:00101B54 80 01 00 48            lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40            addi    %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6            mtlr    %r0
seg000:00101B60 4E 80 00 20            blr
seg000:00101B60                # End of function check1
```

在 IDA 中，我们可以清楚地观察到：虽然程序的多个指令都调用了这个函数，但是调用方函数在调用结束之后只访问了 r3 寄存器的值。这个函数只起到调用其他函数的功能，因此它就是形实转换函数：即使函数序言和函数尾声都十分完整，但是对 r3 寄存器完全没有操作。据此判断，check1()函数的返回值与 check()函数一致。

BLR 指令[①]似乎是函数返回语句，可作为划分函数模块的标识。但是 IDA 能够识别并划分函数体，因此我们可以先不管它。由于本程序采用的是 RISC（精简指令集）指令，调用方函数会通过链接寄存器（Link Register）向被调用方函数传递返回地址。就这些特征来看，PowerPC 的程序与 ARM 程序有很多共同点。

check2()函数略为复杂：

```
seg000:00118684            check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684            .set var_18, -0x18
seg000:00118684            .set var_C, -0xC
seg000:00118684            .set var_8, -8
seg000:00118684            .set var_4, -4
seg000:00118684            .set arg_8,  8
seg000:00118684
seg000:00118684 93 E1 FF FC   stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6   mflr    %r0
seg000:0011868C 83 E2 95 A8   lwz     %r31, off_1485E8 # dword_24B704
seg000:00118690               .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8   stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4   stw     %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78   mr      %r29, %r3
seg000:0011869C 90 01 00 08   stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01   cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0   stwu    %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C   bne     loc_1186B8
seg000:001186B0 38 60 00 01   li      %r3, 1
seg000:001186B4 48 00 00 6C   b       exit
seg000:001186B8
seg000:001186B8            loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5   bl      sub_118A8C
seg000:001186BC 60 00 00 00   nop
seg000:001186C0 3B C0 00 00   li      %r30, 0
seg000:001186C4
seg000:001186C4            skip:       # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F   clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18   beq     loc_1186E0
seg000:001186CC 38 61 00 38   addi    %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00   lwz     %r4, dword_24B704
seg000:001186D4 48 00 C0 55   bl      .RBEFINDNEXT
seg000:001186D8 60 00 00 00   nop
seg000:001186DC 48 00 00 1C   b       loc_1186F8
seg000:001186E0
seg000:001186E0            loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00   lwz     %r5, dword_24B704
seg000:001186E4 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2   li      %r3, 0x1234
seg000:001186EC 48 00 BF 99   bl      .RBEFINDFIRST
seg000:001186F0 60 00 00 00   nop
seg000:001186F4 3B C0 00 01   li     %r30, 1
```

① BLR 是 Branch to Link Register 的缩写。

```
seg000:001186F8
seg000:001186F8                loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F    clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C    beq     must_jump
seg000:00118700 38 60 00 00    li      %r3, 0          # error
seg000:00118704 48 00 00 1C    b       exit
seg000:00118708
seg000:00118708                must_jump:  # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78    mr      %r3, %r29
seg000:0011870C 48 00 00 31    bl      check3
seg000:00118710 60 00 00 00    nop
seg000:00118714 54 60 06 3F    clrlwi. %r0,%r3, 24
seg000:00118718 41 82 FF AC    beq     skip
seg000:0011871C 38 60 00 01    li      %r3, 1
seg000:00118720
seg000:00118720                exit:       # CODE XREF: check2+30j
seg000:00118720                            # check2+80j
seg000:00118720 80 01 00 58    lwz     %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50    addi    %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC    lwz     %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6    mtlr    %r0
seg000:00118730 83 C1 FF F8    lwz     %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4    lwz     %r29, var_C(%sp)
seg000:00118738 4E 80 00 20    blr
seg000:00118738                # End of function check2
```

因为可执行文件保留了部分函数名称，所以分析的难度并非很高。例如，程序文件里有.RBEFINDNEXT()和 .RBEFINDFIRST()等函数名。这可能是编译器留下的调试符号。虽然无法确定这种情况的具体原因，但是在不了解文件格式的情况下，我们可以参考与之类似的 PE 文件格式（可参考 68.2.7 节）。而这些函数最终都调用了.GetNextDeviceViaUSB()函数和.USBSendPKT()函数。从函数名称可以判断，它们都是访问 USB 加密狗的函数。

程序里甚至还直接调用了.GetNextEve3Device()函数。这个函数在 20 世纪 90 年代就非常著名。程序往往通过这个函数访问 Mac 设备上的 ADB 口，最终访问 Sentinel Eve3 加密狗。

我们首先要把其他问题搁置一边，重点关注 r3 寄存器在函数返回前的赋值过程。在分析前面的指令时，我们已经知道，如果 r3 寄存器的值为零，那么程序将转向错误提示信息的信息窗口；所以，我们关注的是对 r3 寄存器进行非零赋值的指令。

上述程序中，有两条“li %r3, {非零值}”指令，有一条“li %r3, 0”指令。LI 是 Load Immediate 的缩写，可见 li 指令的作用是“令寄存器加载立即数”。第一条指令的地址是 0x001186B0。坦白地讲，如果要了解它的具体作用，还需要进一步学习 PowerPC 平台的汇编语言。

然而下一处简明易懂。它调用.RBEFINDFIRST()函数。如果函数验证失败，则 r3 的值为 0，程序将跳转到 exit（退出）；否则，继续调用 check3()函数。如果 check3()函数的验证失败，那么程序将调用.RBEFINDNEXT()函数，大概是检测下一个 USB 口的意思吧。

前文我们介绍过“clrlwi %r0, %r3, 16”的具体功能了。要注意的是，它清除的是 16 位数据。这就代表着.REBFINDFIRST()函数的返回值多半也是 16 位数据。

此外，B（branch）指令是无条件转移指令，BEQ 的触发条件和 BNE 相反。这些指令就不再介绍了。

下面来分析 check3()函数：

```
seg000:0011873C                check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C                .set var_18, -0x18
seg000:0011873C                .set var_C, -0xC
seg000:0011873C                .set var_8, -8
seg000:0011873C                .set var_4, -4
seg000:0011873C                .set arg_8,  8
seg000:0011873C
seg000:0011873C 93 E1 FF FC    stw     %r31, var_4(%sp)
```

```
seg000:00118740 7C 08 02 A6   mflr   %r0
seg000:00118744 38 A0 00 00   li     %r5, 0
seg000:00118748 93 C1 FF F8   stw    %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8   lwz    %r30, off_1485E8 # dword_24B704
seg000:00118750               .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4   stw    %r29, var_C(%sp)
seg000:00118754 3B A3 00 00   addi   %r29, %r3, 0
seg000:00118758 38 60 00 00   li     %r3, 0
seg000:0011875C 90 01 00 08   stw    %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0   stwu   %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00   lwz    %r6, dword_24B704
seg000:00118768 38 81 00 38   addi   %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D   bl     .RBEREAD
seg000:00118770 60 00 00 00   nop
seg000:00118774 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C   beq    loc_118784
seg000:0011877C 38 60 00 00   li     %r3, 0
seg000:00118780 48 00 02 F0   b      exit
seg000:00118784
seg000:00118784               loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2   cmplwi %r0, 0x1100
seg000:0011878C 41 82 00 0C   beq    loc_118798
seg000:00118790 38 60 00 00   li     %r3, 0
seg000:00118794 48 00 02 DC   b      exit
seg000:00118798
seg000:00118798               loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00   lwz    %r6, dword_24B704
seg000:0011879C 38 81 00 38   addi   %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01   li     %r3, 1
seg000:001187A4 38 A0 00 00   li     %r5, 0
seg000:001187A8 48 00 C0 21   bl     .RBEREAD
seg000:001187AC 60 00 00 00   nop
seg000:001187B0 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C   beq    loc_1187C0
seg000:001187B8 38 60 00 00   li     %r3, 0
seg000:001187BC 48 00 02 B4   b      exit
seg000:001187C0
seg000:001187C0               loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B   cmplwi %r0, 0x09AB
seg000:001187C8 41 82 00 0C   beq    loc_1187D4
seg000:001187CC 38 60 00 00   li     %r3, 0
seg000:001187D0 48 00 02 A0   b      exit
seg000:001187D4
seg000:001187D4               loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9   bl     sub_B7BAC
seg000:001187D8 60 00 00 00   nop
seg000:001187DC 54 60 06 3E   clrlwi %r0, %r3, 24
seg000:001187E0 2C 00 00 05   cmpwi  %r0, 5
seg000:001187E4 41 82 01 00   beq    loc_1188E4
seg000:001187E8 40 80 00 10   bge    loc_1187F8
seg000:001187EC 2C 00 00 04   cmpwi  %r0, 4
seg000:001187F0 40 80 00 58   bge    loc_118848
seg000:001187F4 48 00 01 8C   b      loc_118980
seg000:001187F8
seg000:001187F8               loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B   cmpwi  %r0, 0xB
seg000:001187FC 41 82 00 08   beq    loc_118804
seg000:00118800 48 00 01 80   b      loc_118980
seg000:00118804
seg000:00118804               loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00   lwz    %r6, dword_24B704
seg000:00118808 38 81 00 38   addi   %r4, %sp, 0x50+var_18
```

```
seg000:0011880C 38 60 00 08   li     %r3, 8
seg000:00118810 38 A0 00 00   li     %r5, 0
seg000:00118814 48 00 BF B5   bl     .RBEREAD
seg000:00118818 60 00 00 00   nop
seg000:0011881C 54 60 04 3F   clrlwi.%r0, %r3, 16
seg000:00118820 41 82 00 0C   beq    loc_11882C
seg000:00118824 38 60 00 00   li     %r3, 0
seg000:00118828 48 00 02 48   b      exit
seg000:0011882C
seg000:0011882C               loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30   cmplwi %r0, 0xFEA0
seg000:00118834 41 82 00 0C   beq    loc_118840
seg000:00118838 38 60 00 00   li     %r3, 0
seg000:0011883C 48 00 02 34   b      exit
seg000:00118840
seg000:00118840               loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01   li     %r3, 1
seg000:00118844 48 00 02 2C   b      exit
seg000:00118848
seg000:00118848               loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00   lwz    %r6, dword_24B704
seg000:0011884C 38 81 00 38   addi   %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A   li     %r3, 0xA
seg000:00118854 38 A0 00 00   li     %r5, 0
seg000:00118858 48 00 BF 71   bl     .RBEREAD
seg000:0011885C 60 00 00 00   nop
seg000:00118860 54 60 04 3F   clrlwi.%r0, %r3, 16
seg000:00118864 41 82 00 0C   beq    loc_118870
seg000:00118868 38 60 00 00   li     %r3, 0
seg000:0011886C 48 00 02 04   b      exit
seg000:00118870
seg000:00118870               loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3   cmplwi %r0, 0xA6E1
seg000:00118878 41 82 00 0C   beq    loc_118884
seg000:0011887C 38 60 00 00   li     %r3, 0
seg000:00118880 48 00 01 F0   b      exit
seg000:00118884
seg000:00118884               loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E   clrlwi %r31, %r29, 24
seg000:00118888 28 1F 00 02   cmplwi %r31, 2
seg000:0011888C 40 82 00 0C   bne    loc_118898
seg000:00118890 38 60 00 01   li     %r3, 1
seg000:00118894 48 00 01 DC   b      exit
seg000:00118898
seg000:00118898               loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00   lwz    %r6, dword_24B704
seg000:0011889C 38 81 00 38   addi   %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B   li     %r3, 0xB
seg000:001188A4 38 A0 00 00   li     %r5, 0
seg000:001188A8 48 00 BF 21   bl     .RBEREAD
seg000:001188AC 60 00 00 00   nop
seg000:001188B0 54 60 04 3F   clrlwi.%r0, %r3, 16
seg000:001188B4 41 82 00 0C   beq    loc_1188C0
seg000:001188B8 38 60 00 00   li     %r3, 0
seg000:001188BC 48 00 01 B4   b      exit
seg000:001188C0
seg000:001188C0               loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C   cmplwi %r0, 0x1C20
seg000:001188C8 41 82 00 0C   beq    loc_1188D4
seg000:001188CC 38 60 00 00   li     %r3, 0
seg000:001188D0 48 00 01 A0   b      exit
seg000:001188D4
```

```
seg000:001188D4                 loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03       cmplwi %r31, 3
seg000:001188D8 40 82 01 94       bne    error
seg000:001188DC 38 60 00 01       li     %r3, 1
seg000:001188E0 48 00 01 90       b      exit
seg000:001188E4
seg000:001188E4                 loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00       lwz    %r6, dword_24B704
seg000:001188E8 38 81 00 38       addi   %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C       li     %r3, 0xC
seg000:001188F0 38 A0 00 00       li     %r5, 0
seg000:001188F4 48 00 BE D5       bl     .RBEREAD
seg000:001188F8 60 00 00 00       nop
seg000:001188FC 54 60 04 3F       clrlwi.%r0, %r3, 16
seg000:00118900 41 82 00 0C       beq    loc_11890C
seg000:00118904 38 60 00 00       li     %r3, 0
seg000:00118908 48 00 01 68       b      exit
seg000:0011890C
seg000:0011890C                 loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38       lhz    %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40       cmplwi %r0, 0x40FF
seg000:00118914 41 82 00 0C       beq    loc_118920
seg000:00118918 38 60 00 00       li     %r3, 0
seg000:0011891C 48 00 01 54       b      exit
seg000:00118920
seg000:00118920                 loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E       clrlwi %r31, %r29, 24
seg000:00118924 28 1F 00 02       cmplwi %r31, 2
seg000:00118928 40 82 00 0C       bne    loc_118934
seg000:0011892C 38 60 00 01       li     %r3, 1
seg000:00118930 48 00 01 40       b      exit
seg000:00118934
seg000:00118934                 loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00       lwz    %r6, dword_24B704
seg000:00118938 38 81 00 38       addi   %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D       li     %r3, 0xD
seg000:00118940 38 A0 00 00       li     %r5, 0
seg000:00118944 48 00 BE 85       bl     .RBEREAD
seg000:00118948 60 00 00 00       nop
seg000:0011894C 54 60 04 3F       clrlwi.%r0, %r3, 16
seg000:00118950 41 82 00 0C       beq    loc_11895C
seg000:00118954 38 60 00 00       li     %r3, 0
seg000:00118958 48 00 01 18       b      exit
seg000:0011895C
seg000:0011895C                 loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38       lhz    %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF       cmplwi %r0, 0xFC7
seg000:00118964 41 82 00 0C       beq    loc_118970
seg000:00118968 38 60 00 00       li     %r3, 0
seg000:0011896C 48 00 01 04       b      exit
seg000:00118970
seg000:00118970                 loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03       cmplwi %r31, 3
seg000:00118974 40 82 00 F8       bne    error
seg000:00118978 38 60 00 01       li     %r3, 1
seg000:0011897C 48 00 00 F4       b      exit
seg000:00118980
seg000:00118980                 loc_118980: # CODE XREF: check3+B8j
seg000:00118980                             # check3+C4j
seg000:00118980 80 DE 00 00       lwz    %r6, dword_24B704
seg000:00118984 38 81 00 38       addi   %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00       li     %r31, 0
seg000:0011898C 38 60 00 04       li     %r3, 4
seg000:00118990 38 A0 00 00       li     %r5, 0
seg000:00118994 48 00 BE 35       bl     .RBEREAD
```

```
seg000:00118998 60 00 00 00   nop
seg000:0011899C 54 60 04 3F   clrlwi.%r0, %r3, 16
seg000:001189A0 41 82 00 0C   beq    loc_1189AC
seg000:001189A4 38 60 00 00   li     %r3, 0
seg000:001189A8 48 00 00 C8   b      exit
seg000:001189AC
seg000:001189AC               loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A   cmplwi %r0, 0xAED0
seg000:001189B4 40 82 00 0C   bne    loc_1189C0
seg000:001189B8 3B E0 00 01   li     %r31, 1
seg000:001189BC 48 00 00 14   b      loc_1189D0
seg000:001189C0
seg000:001189C0               loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28   cmplwi %r0, 0x2818
seg000:001189C4 41 82 00 0C   beq    loc_1189D0
seg000:001189C8 38 60 00 00   li     %r3, 0
seg000:001189CC 48 00 00 A4   b      exit
seg000:001189D0
seg000:001189D0               loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0                           # check3+288j
seg000:001189D0 57 A0 06 3E   clrlwi %r0, %r29, 24
seg000:001189D4 28 00 00 02   cmplwi %r0, 2
seg000:001189D8 40 82 00 20   bne    loc_1189F8
seg000:001189DC 57 E0 06 3F   clrlwi.%r0, %r31, 24
seg000:001189E0 41 82 00 10   beq    good2
seg000:001189E4 48 00 4C 69   bl     sub_11D64C
seg000:001189E8 60 00 00 00   nop
seg000:001189EC 48 00 00 84   b      exit
seg000:001189F0
seg000:001189F0               good2:     # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01   li     %r3, 1
seg000:001189F4 48 00 00 7C   b      exit
seg000:001189F8
seg000:001189F8               loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00   lwz    %r6, dword_24B704
seg000:001189FC 38 81 00 38   addi   %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05   li     %r3, 5
seg000:00118A04 38 A0 00 00   li     %r5, 0
seg000:00118A08 48 00 BD C1   bl     .RBEREAD
seg000:00118A0C 60 00 00 00   nop
seg000:00118A10 54 60 04 3F   clrlwi.%r0, %r3, 16
seg000:00118A14 41 82 00 0C   beq    loc_118A20
seg000:00118A18 38 60 00 00   li     %r3, 0
seg000:00118A1C 48 00 00 54   b      exit
seg000:00118A20
seg000:00118A20               loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38   lhz    %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3   cmplwi %r0, 0xD300
seg000:00118A28 40 82 00 0C   bne    loc_118A34
seg000:00118A2C 3B E0 00 01   li     %r31, 1
seg000:00118A30 48 00 00 14   b      good1
seg000:00118A34
seg000:00118A34               loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB   cmplwi %r0, 0xEBA1
seg000:00118A38 41 82 00 0C   beq    good1
seg000:00118A3C 38 60 00 00   li     %r3, 0
seg000:00118A40 48 00 00 30   b      exit
seg000:00118A44
seg000:00118A44               good1:     # CODE XREF: check3+2F4j
seg000:00118A44                          # check3+2FCj
seg000:00118A44 57 A0 06 3E   clrlwi %r0, %r29, 24
seg000:00118A48 28 00 00 03   cmplwi %r0, 3
seg000:00118A4C 40 82 00 20   bne    error
seg000:00118A50 57 E0 06 3F   clrlwi.%r0, %r31, 24
```

```
seg000:00118A54 41 82 00 10    beq     good
seg000:00118A58 48 00 4B F5    bl      sub_11D64C
seg000:00118A5C 60 00 00 00    nop
seg000:00118A60 48 00 00 10    b       exit
seg000:00118A64
seg000:00118A64                good:   # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01    li      %r3, 1
seg000:00118A68 48 00 00 08    b       exit
seg000:00118A6C
seg000:00118A6C                error:  # CODE XREF: check3+19Cj
seg000:00118A6C                        # check3+238j ...
seg000:00118A6C 38 60 00 00    li      %r3, 0
seg000:00118A70
seg000:00118A70                exit:   # CODE XREF: check3+44j
seg000:00118A70                        # check3+58j ...
seg000:00118A70 80 01 00 58    lwz     %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50    addi    %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC    lwz     %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6    mtlr    %r0
seg000:00118A80 83 C1 FF F8    lwz     %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4    lwz     %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20    blr
seg000:00118A88                # End of function check3
```

此函数多次调用了.RBEREAD()函数。在调用后面的这个函数之后，check3()函数又大量使用 CMPLWI 指令将返回值与特定的固定值进行比较。由此可见.RBEREAD()函数大体是从加密狗读取数据的函数。

另外，在调用.RBEREAD()函数之前，r3 寄存器的取值不外乎 0、1、8、0xA、0xB、0xC、0xD、4 和 5。这很可能是内存地址一类的信息。

如果使用 google 引擎搜索这些函数名，google 的查询结果多数就是 Sentinel Eve 3 加密狗开发手册。

其实，在不了解其他 PowerPC 指令的情况下，我们照样可以分析加密狗的有关操作。毕竟认证函数就是这几个，而且“认证成功”的返回值肯定是 1，“认证失败”的返回值又肯定是 0。

综合上述分析，只要让 check1()函数的返回值固定为 1 或者其他某个非零值，即可破解软件狗认证。但是因为我并不熟悉 PowerPC 的指令，所以我决定采取保守的修改方案：修改 check2()函数中 0x001186FC 处和 0x00118718 处的转移指令。

我把 0x001186FC 的指令改为 0x48 和 0，即把 BEQ 指令替换为无条件转移指令 B。即使不参阅[IBM00]的参考手册，我们也能在程序里找到 B 指令的 opcode。

另外，我把 0x00118718 处修改为一个 0x60 和三个 0 字节，将有关指令改为 NOP 指令。当然，这也是从原程序里找的 opcode。

进行上述修改之后，在不插入加密狗的情况下，程序仍然可正常运行。

总之，借助 IDA 和部分汇编知识，任何人都可以小规模地修改程序。

78.2 例 2: SCO OpenServer

本例研究的程序是 1997 年开发的面向 SCO OpenServer 的程序。因为年代过于久远，买家早就找不到开发商了。

这款软件的加密狗驱动程序是定制程序。这个驱动程序里有“Copyright 1989, Rainbow Technologies, Inc., Irvine, CA”和“Sentinel Integrated Driver Ver. 3.0”的字样。

在 SCO OpenServer 上安装驱动程序之后，硬件加密狗将会加载到文件系统的/dev 目录里：

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

如果不插入加密狗，程序将会报错。而且错误信息不在可执行程序里。

好在 IDA 可以加载 SCO OpenServer 的 COFF 程序。

在 IDA 里搜索“rbsl”，然后找到了下述指令：

```
.text:00022AB8          public SSQC
.text:00022AB8 SSQC     proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr  8
.text:00022AB8
.text:00022AB8          push    ebp
.text:00022AB9          mov     ebp, esp
.text:00022ABB          sub     esp, 44h
.text:00022ABE          push    edi
.text:00022ABF          mov     edi, offset unk_4035D0
.text:00022AC4          push    esi
.text:00022AC5          mov     esi, [ebp+arg_0]
.text:00022AC8          push    ebx
.text:00022AC9          push    esi
.text:00022ACA          call    strlen
.text:00022ACF          add     esp, 4
.text:00022AD2          cmp     eax, 2
.text:00022AD7          jnz     loc_22BA4
.text:00022ADD          inc     esi
.text:00022ADE          mov     al, [esi-1]
.text:00022AE1          movsx   eax, al
.text:00022AE4          cmp     eax, '3'
.text:00022AE9          jz      loc_22B84
.text:00022AEF          cmp     eax, '4'
.text:00022AF4          jz      loc_22B94
.text:00022AFA          cmp     eax, '5'
.text:00022AFF          jnz     short loc_22B6B
.text:00022B01          movsx   ebx, byte ptr [esi]
.text:00022B04          sub     ebx, '0'
.text:00022B07          mov     eax, 7
.text:00022B0C          add     eax, ebx
.text:00022B0E          push    eax
.text:00022B0F          lea     eax, [ebp+var_44]
.text:00022B12          push    offset aDevSlD ; "/dev/sl%d"
.text:00022B17          push    eax
.text:00022B18          call    nl_sprintf
.text:00022B1D          push    0               ; int
.text:00022B1F          push    offset aDevRbsl8 ; char *
.text:00022B24          call    _access
.text:00022B29          add     esp, 14h
.text:00022B2C          cmp     eax, 0FFFFFFFFh
.text:00022B31          jz      short loc_22B48
.text:00022B33          lea     eax, [ebx+7]
.text:00022B36          push    eax
.text:00022B37          lea     eax, [ebp+var_44]
.text:00022B3A          push    offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F          push    eax
.text:00022B40          call    nl_sprintf
.text:00022B45          add     esp, 0Ch
.text:00022B48
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j
.text:00022B48          mov     edx, [edi]
.text:00022B4A          test    edx, edx
.text:00022B4C          jle     short loc_22b57
.text:00022B4E          push    edx             ; int
.text:00022B4F          call    _close
.text:00022B54          add     esp, 4
.text:00022B57
```

```
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57        push    2               ; int
.text:00022B59        lea     eax, [ebp+var_44]
.text:00022B5C        push    eax             ; char *
.text:00022B5D        call    _open
.text:00022B62        add     esp, 8
.text:00022B65        test    eax, eax
.text:00022B67        mov     [edi], eax
.text:00022B69        jge     short loc_22B78
.text:00022B6B
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B        mov     eax, 0FFFFFFFFh
.text:00022B70        pop     ebx
.text:00022B71        pop     esi
.text:00022B72        pop     edi
.text:00022B73        mov     esp,ebp
.text:00022B75        pop     ebp
.text:00022B76        retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78        pop     ebx
.text:00022B79        pop     esi
.text:00022B7A        pop     edi
.text:00022B7B        xor     eax, eax
.text:00022B7D        mov     esp, ebp
.text:00022B7F        pop     ebp
.text:00022B80        retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84        mov     al, [esi]
.text:00022B86        pop     ebx
.text:00022B87        pop     esi
.text:00022B88        pop     edi
.text:00022B89        mov     ds:byte_407224, al
.text:00022B8E        mov     esp, ebp
.text:00022B90        xor     eax, eax
.text:00022B92        pop     ebp
.text:00022B93        retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94        mov     al, [esi]
.text:00022B96        pop     ebx
.text:00022B97        pop     esi
.text:00022B98        pop     edi
.text:00022B99        mov     ds:byte_407225, al
.text:00022B9E        mov     esp, ebp
.text:00022BA0        xor     eax, eax
.text:00022BA2        pop     ebp
.text:00022BA3        retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4        movsx   eax, ds:byte_407225
.text:00022BAB        push    esi
.text:00022BAC        push    eax
.text:00022BAD        movsx   eax, ds:byte_407224
.text:00022BB4        push    eax
.text:00022BB5        lea     eax, [ebp+var_44]
.text:00022BB8        push    offset a46CCS   ; "46%c%c%s"
.text:00022BBD        push    eax
.text:00022BBE        call    nl_sprintf
.text:00022BC3        lea     eax, [ebp+var_44]
.text:00022BC6        push    eax
.text:00022BC7        call    strlen
.text:00022BCC        add     esp, 18h
.text:00022BCF        cmp     eax, 1Bh
.text:00022BD4        jle     short loc_22BDA
```

```
.text:00022BD6        mov     [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA        lea     eax, [ebp+var_44]
.text:00022BDD        push    eax
.text:00022BDE        call    strlen
.text:00022BE3        push    eax             ; unsigned int
.text:00022BE4        lea     eax, [ebp+var_44]
.text:00022BE7        push    eax             ; void *
.text:00022BE8        mov     eax, [edi]
.text:00022BEA        push    eax             ; int
.text:00022BEB        call    _write
.text:00022BF0        add     esp, 10h
.text:00022BF3        pop     ebx
.text:00022BF4        pop     esi
.text:00022BF5        pop     edi
.text:00022BF6        mov     esp, ebp
.text:00022BF8        pop     ebp
.text:00022BF9        retn
.text:00022BFA        db 0Eh dup(90h)
.text:00022BFA SSQC   endp
```

果然，该程序要和驱动程序通信。

而且，只有下面这个形实转换函数调用了 SSQC()函数：

```
.text:0000DBE8        public SSQ
.text:0000DBE8 SSQ    proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                  ; sys_info+CBp...
.text:0000DBE8
.text:0000DBE8 arg_0  = dword ptr  8
.text:0000DBE8
.text:0000DBE8        push    ebp
.text:0000DBE9        mov     ebp, esp
.text:0000DBEB        mov     edx, [ebp+arg_0]
.text:0000DBEE        push    edx
.text:0000DBEF        call    SSQC
.text:0000DBF4        add     esp, 4
.text:0000DBF7        mov     esp, ebp
.text:0000DBF9        pop     ebp
.text:0000DBFA        retn
.text:0000DBFB SSQ    endp
```

调用 SSQ()函数的指令至少有两处。

其中一处是：

```
.data:0040169C _51_52_53        dd offset aPressAnyKeyT_0 ; DATA XREF: init_sys+392r
.data:0040169C                                            ; sys_info+A1r
.data:0040169C                                            ; "PRESS ANY KEY TO CONTINUE: "
.data:004016A0                  dd offset a51             ; "51"
.data:004016A4                  dd offset a52             ; "52"
.data:004016A8                  dd offset a53             ; "53"

...

.data:004016B8 _3C_or_3E        dd offset a3c             ; DATA XREF: sys_info:loc_D67Br
.data:004016B8                                            ; "3C"
.data:004016BC                  dd offset a3e             ; "3E"

; these names we gave to the labels:
.data:004016C0 answers1         dd 6B05h                  ; DATA XREF: sys_info+E7r
.data:004016C4                  dd 3D87h
.data:004016C8 answers2         dd 3Ch                    ; DATA XREF: sys_info+F2r
.data:004016CC                  dd 832h
.data:004016D0 _C_and_B         db 0Ch                    ; DATA XREF: sys_info+BAr
.data:004016D0                                            ; sys_info:OKr
.data:004016D1 byte_4016D1      db 0Bh                    ; DATA XREF: sys_info+FDr
```

```
.data:004016D2                  db    0

...

.text:0000D652                  xor     eax, eax
.text:0000D654                  mov     al, ds:ctl_port
.text:0000D659                  mov     ecx, _51_52_53[eax*4]
.text:0000D660                  push    ecx
.text:0000D661                  call    SSQ
.text:0000D666                  add     esp, 4
.text:0000D669                  cmp     eax, 0FFFFFFFFh
.text:0000D66E                  jz      short loc_D6D1
.text:0000D670                  xor     ebx, ebx
.text:0000D672                  mov     al, _C_and_B
.text:0000D677                  test    al, al
.text:0000D679                  jz      short loc_D6C0
.text:0000D67B
.text:0000D67B loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B                  mov     eax, _3C_or_3E[ebx*4]
.text:0000D682                  push    eax
.text:0000D683                  call    SSQ
.text:0000D688                  push    offset a4g    ; "4G"
.text:0000D68D                  call    SSQ
.text:0000D692                  push    offset a0123456789 ; "0123456789"
.text:0000D697                  call    SSQ
.text:0000D69C                  add     esp, 0Ch
.text:0000D69F                  mov     edx, answers1[ebx*4]
.text:0000D6A6                  cmp     eax, edx
.text:0000D6A8                  jz      short OK
.text:0000D6AA                  mov     ecx, answers2[ebx*4]
.text:0000D6B1                  cmp     eax, ecx
.text:0000D6B3                  jz      short OK
.text:0000D6B5                  mov     al, byte_4016D1[ebx]
.text:0000D6BB                  inc     ebx
.text:0000D6BC                  test    al, al
.text:0000D6BE                  jnz     short loc_D67B
.text:0000D6C0
.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0                  inc     ds:ctl_port
.text:0000D6C6                  xor     eax, eax
.text:0000D6C8                  mov     al, ds:ctl_port
.text:0000D6CD                  cmp     eax, edi
.text:0000D6CF                  jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1                  ; sys_info+B6j
.text:0000D6D1                  mov     edx, [ebp+var_8]
.text:0000D6D4                  inc     edx
.text:0000D6D5                  mov     [ebp+var_8], edx
.text:0000D6D8                  cmp     edx, 3
.text:0000D6DB                  jle     loc_D641
.text:0000D6E1
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1                  ; sys_info+51j ...
.text:0000D6E1                  pop     ebx
.text:0000D6E2                  pop     edi
.text:0000D6E3                  mov     esp, ebp
.text:0000D6E5                  pop     ebp
.text:0000D6E6                  retn
.text:0000D6E8 OK:              ; CODE XREF: sys_info+F0j
.text:0000D6E8                  ; sys_info+FBj
.text:0000D6E8                  mov     al, _C_and_B[ebx]
.text:0000D6EE                  pop     ebx
.text:0000D6EF                  pop     edi
.text:0000D6F0                  mov     ds:ctl_model, al
```

```
.text:0000D6F5                  mov     esp, ebp
.text:0000D6F7                  pop     ebp
.text:0000D6F8                  retn
.text:0000D6F8 sys_info         endp
```

“3C”和“3E”听起来很熟：它是私有的、单功能加密－哈希函数，用于 Rainbow 公司生产的一款没有内存的 Sentinel Pro 加密狗。

本书的第 34 章详细介绍过哈希函数。

就这个程序而言，我们可以据此判断它仅检测加密狗的“有/无”信息。这款加密狗上不具备内存芯片，也就无法存储信息；换句话说这个程序不会向加密狗写数据。后面连续出现的双字符代码是指令代码，由 SSQC()函数接收和处理。其他字符串的哈希值都以 16 位数字的形式存储在加密狗里。因为开发厂商的加密算法是私有算法，所以编写驱动程序替身、或者仿制硬件加密狗的做法都行不通。但是，“截获所有访问加密狗的操作、继而找到程序核对的哈希值”确实行得通。不怕麻烦的话，我们还可以摸索程序逻辑、基于私有的加密哈希函数再建一个软件，从而使用自制软件替代原有软件对数据文件进行加解密。

代码 51/52/53 用于选择 LPT 打印机接口。3x/4x 用于选择相应的加密狗“系列”。不同类型的 Sentinel Pro 加密狗可以接在同一个 LPT 接口上，而区分加密狗的工作则由应用程序完成。

除了字符串“0123456789”以外，传递给哈希函数的值都是双字符的字符串。然后，函数返回的哈希值与一系列有效值进行比较。如果该值有效，全局变量 ctl_model 将被赋值为 0xC 或 0xB。

此外，程序还定义了字符串“PRESS ANY KEY TO CONTINUE:”，这应当是通过加密狗认证之后的提示信息。不过整个程序没有调用过这个字符串，恐怕这属于源程序的 bug 吧。

接下来，我们要关注全局变量 ctl_mode 的读取指令。

其中一处是：

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14    = dword ptr -14h
.text:0000D708 var_10    = byte ptr -10h
.text:0000D708 var_8     = dword ptr -8
.text:0000D708 var_2     = word ptr -2
.text:0000D708
.text:0000D708           push    ebp
.text:0000D709           mov     eax, ds:net_env
.text:0000D70E           mov     ebp, esp
.text:0000D710           sub     esp, 1Ch
.text:0000D713           test    eax, eax
.text:0000D715           jnz     short loc_D734
.text:0000D717           mov     al, ds:ctl_model
.text:0000D71C           test    al, al
.text:0000D71E           jnz     short loc_D77E
.text:0000D720           mov     [ebp+var_8], offset aIeCvulnvvOkgT_ ; "Ie-cvulnvV\\\bOKG]T_"
.text:0000D727           mov     edx, 7
.text:0000D72C           jmp     loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7           ; prep_sys+33j
.text:0000D7E7           push    edx
.text:0000D7E8           mov     edx, [ebp+var_8]
.text:0000D7EB           push    20h
.text:0000D7ED           push    edx
.text:0000D7EE           push    16h
.text:0000D7F0           call    err_warn
.text:0000D7F5           push    offset station_sem
.text:0000D7FA           call    ClosSem
.text:0000D7FF           call    startup_err
```

如果 ctl_mode 为 0，那么一条经过加密处理的错误信息将被传递到解密程序，从而出现在屏幕上。这个错误信息的解密方法就是简单的 XOR 算法：

```
.text:0000A43C err_warn         proc near               ; CODE XREF: prep_sys+E8p
.text:0000A43C                                          ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55           = byte ptr -55h
.text:0000A43C var_54           = byte ptr -54h
.text:0000A43C arg_0            = dword ptr  8
.text:0000A43C arg_4            = dword ptr  0Ch
.text:0000A43C arg_8            = dword ptr  10h
.text:0000A43C arg_C            = dword ptr  14h
.text:0000A43C
.text:0000A43C                  push    ebp
.text:0000A43D                  mov     ebp, esp
.text:0000A43F                  sub     esp, 54h
.text:0000A442                  push    edi
.text:0000A443                  mov     ecx, [ebp+arg_8]
.text:0000A446                  xor     edi, edi
.text:0000A448                  test    ecx, ecx
.text:0000A44A                  push    esi
.text:0000A44B                  jle     short loc_A466
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:                                ; CODE XREF: err_warn+28j
.text:0000A453                  xor     eax, eax
.text:0000A455                  mov     al, [edx+edi]
.text:0000A458                  xor     eax, esi
.text:0000A45A                  add     esi, 3
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
.text:0000A466
.text:0000A466 loc_A466:                                ; CODE XREF: err_warn+Fj
.text:0000A466                  mov     [ebp+edi+var_54], 0
.text:0000A46B                  mov     eax, [ebp+arg_0]
.text:0000A46E                  cmp     eax, 18h
.text:0000A473                  jnz     short loc_A49C
.text:0000A475                  lea     eax, [ebp+var_54]
.text:0000A478                  push    eax
.text:0000A479                  call    status_line
.text:0000A47E                  add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:                                ; CODE XREF: err_warn+72j
.text:0000A481                  push    50h
.text:0000A483                  push    0
.text:0000A485                  lea     eax, [ebp+var_54]
.text:0000A488                  push    eax
.text:0000A489                  call    memset
.text:0000A48E                  call    pcv_refresh
.text:0000A493                  add     esp, 0Ch
.text:0000A496                  pop     esi
.text:0000A497                  pop     edi
.text:0000A498                  mov     esp, ebp
.text:0000A49A                  pop     ebp
.text:0000A49B                  retn
.text:0000A49C
.text:0000A49C loc_A49C:                                ; CODE XREF: err_warn+37j
.text:0000A49C                  push    0
.text:0000A49E                  lea     eax, [ebp+var_54]
.text:0000A4A1                  mov     edx, [ebp+arg_0]
.text:0000A4A4                  push    edx
```

```
.text:0000A4A5                  push    eax
.text:0000A4A6                  call    pcv_lputs
.text:0000A4AB                  add     esp, 0Ch
.text:0000A4AE                  jmp     short loc_A481
.text:0000A4AE err_warn         endp
```

因为程序对提示信息进行了加密处理（这是常见手段），所以我无法在可执行程序里直接找到错误的提示信息。

此外，程序里还有一个调用哈希函数 SSQ()的地方。在调用过程中，该处指令向哈希函数传递了字符串“offln”。后续指令将返回值与 0xFE81 和 0x12A9 进行比对。如果两个值不匹配，程序将启用 timer()函数（貌似是等待用户重新插入加密狗），然后在屏幕上显示另一个错误提示信息：

```
.text:0000DA55 loc_DA55:                                ; CODE XREF: sync_sys+24Cj
.text:0000DA55                  push    offset aOffln   ; "offln"
.text:0000DA5A                  call    SSQ
.text:0000DA5F                  add     esp, 4
.text:0000DA62                  mov     dl, [ebx]
.text:0000DA64                  mov     esi, eax
.text:0000DA66                  cmp     dl, 0Bh
.text:0000DA69                  jnz     short loc_DA83
.text:0000DA6B                  cmp     esi, 0FE81h
.text:0000DA71                  jz      OK
.text:0000DA77                  cmp     esi, 0FFFFF8EFh
.text:0000DA7D                  jz      OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF: sync_sys+201j
.text:0000DA83                  mov     cl, [ebx]
.text:0000DA85                  cmp     cl, 0Ch
.text:0000DA88                  jnz     short loc_DA9F
.text:0000DA8A                  cmp     esi, 12A9h
.text:0000DA90                  jz      OK
.text:0000DA96                  cmp     esi, 0FFFFFFF5h
.text:0000DA99                  jz      OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE XREF: sync_sys+220j
.text:0000DA9F                  mov     eax, [ebp+var_18]
.text:0000DAA2                  test    eax, eax
.text:0000DAA4                  jz      short loc_DAB0
.text:0000DAA6                  push    24h
.text:0000DAA8                  call    timer
.text:0000DAAD                  add     esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0:                                ; CODE XREF: sync_sys+23Cj
.text:0000DAB0                  inc     edi
.text:0000DAB1                  cmp     edi, 3
.text:0000DAB4                  jle     short loc_DA55
.text:0000DAB6                  mov     eax, ds:net_env
.text:0000DABB                  test    eax, eax
.text:0000DABD                  jz      short error

...

.text:0000DAF7 error:                                   ; CODE XREF: sync_sys+255j
.text:0000DAF7                                          ; sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name I gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                          ; sync_sys+2ABj
```

```
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
.text:0000D9BD                  mov     edx, [ebp+var_C] ; key
.text:0000D9C0                  mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                  push    edx
.text:0000D9C4                  push    20h
.text:0000D9C6                  push    ecx
.text:0000D9C7                  push    18h
.text:0000D9C9                  call    err_warn
.text:0000D9CE                  push    0Fh
.text:0000D9D0                  push    190h
.text:0000D9D5                  call    sound
.text:0000D9DA                  mov     [ebp+var_18], 1
.text:0000D9E1                  add     esp, 18h
.text:0000D9E4                  call    pcv_kbhit
.text:0000D9E9                  test    eax, eax
.text:0000D9EB                  jz      short loc_D9FB

...

; this name I gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah, 49h, 4Ch, 2 dup(47h)
.data:00401736                  db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, 33h, 36h, 76h
.data:00401736                  db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, 1Ah
```

可见，破解加密狗的工作十分简单：我们只需要找到相关的 CMP 指令，把它后面的转移指令替换为无条件转移指令即可。当然，编写自制的 SCO OpenServer 驱动程序也不失为一种方法。

解密错误信息

我们还能够破解源程序中的错误信息。程序里 err_warn()函数所采用的解密算法非常简单。

指令清单 78.1 Decryption function

```
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453                  xor     eax, eax
.text:0000A455                  mov     al, [edx+edi] ; load encrypted byte
.text:0000A458                  xor     eax, esi      ; decrypt it
.text:0000A45A                  add     esi, 3        ; change key for the next byte
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
```

由此可见，程序不仅向解密函数传递了加密后的字符串，而且向它传递了加密密钥：

```
.text:0000DAF7 error:                                   ; CODE XREF: sync_sys+255j
.text:0000DAF7                                          ; sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name we gave to label manually:
.text:0000D9B6 decrypt_end_print_message:               ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                          ; sync_sys+2ABj
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
```

```
.text:0000D9BD                mov     edx, [ebp+var_C] ; key
.text:0000D9C0                mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                push    edx
.text:0000D9C4                push    20h
.text:0000D9C6                push    ecx
.text:0000D9C7                push    18h
.text:0000D9C9                call    err_warn
```

这就是简单的 XOR 算法：每个字节都与密钥进行 XOR 运算，而且每解密一个字节密钥就增加 3。为了验证这个猜想，我专门编写了一个 Python 脚本程序：

指令清单 78.2 Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
        sys.stdout.write ("%c" % (i^tmp))
        tmp=tmp+3
sys.stdout.flush()
```

它解密出来的字符串正是“check security device connection”。

程序还用到了其他的加密字符串和相应密钥。然而我们不需要原始密钥就可以进行密文解密。首先，密钥实际只有 1 个字节。解密核心的 XOR 指令以字节为操作单位。其次，虽然密钥存储于 ESI 寄存器，但是它只用了 ESI 寄存器地址最低的那个字节。即使密钥大于 255，但是参与演算的密钥还是不会大于 1 个字节的值。

最终，我们可以使用 0～255 之间的密钥暴力破解加密字符串。与此同时，我们要排除那些含有控制字符的解密结果。

指令清单 78.3 Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]
```

```
def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
        print ("message #%d" % cnt)
        for key in range(0,256):
                result=[]
                tmp=key
                for i in msg:
                        result.append (i^tmp)
                        tmp=tmp+3
                if is_string_printable (result):
                        print ("key=", key, "value=", "".join(list(map(chr, result))))
        cnt=cnt+1
```

上述程序的运行结果如下。

指令清单 78.4　Results

```
message #1
key= 20 value= 'eb^h%|''hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!'nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!'!#!
message #3
key= 7 value= Bk<waoqNUpu$'yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%gxqd''_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml. 'mwhj
key= 10 value= Ol!td'tMhwx'efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{'whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#'juknuhydk!vrbsp!Zy'dbefe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!'!
```

虽然出现了人类语言之外的字符串，但是我们还是能够找到英语字符串。

另外，因为程序采用的解密算法是非常简单的 XOR 算法，所以它的加密函数也不会复杂到哪去。如果有必要的话，我们甚至可以用它的算法加密自己的字符串，然后把自制的密文放在程序里面。

78.3　例 3: MS-DOS

本例研究的是一款 1995 年研发的 MS-DOS 程序，联络不上开发商了。

在从前的那个坚守 DOS 阵地的时代，所有的 MS-DOS 程序基本都在 16 位的 8086 或者 80286 CPU 上运行。因此大批的程序都是 16 位程序。这种程序的指令与本书介绍过的汇编指令大体相同，只是寄存器是 16 位寄存器、而且指令集略微小些罢了。

MS-DOS 系统没有系统驱动程序，全部程序都可以直接访问硬件端口。故而程序中大量出现了 OUT/IN 指令。就当今的操作系统来说，基本上只有驱动程序才会使用这些指令，而且应用程序已经不能直接访问硬件端口了。

在当时的技术条件下，访问加密狗的 MS-DOS 程序必须直接访问 LPT 打印端口。那么我们就搜索这些端口操作指令好了：

```
seg030:0034              out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034                                ; sent_pro+2Ap ...
seg030:0034
seg030:0034              arg_0    =byteptr 6
seg030:0034
seg030:0034 55                    push   bp
seg030:0035 8B EC                 mov    bp, sp
seg030:0037 8B 16 7E E7           mov    dx, _out_port ; 0x378
seg030:003B 8A 46 06              mov    al, [bp+arg_0]
seg030:003E EE                    out    dx, al
seg030:003F 5D                    pop    bp
seg030:0040 CB                    retf
seg030:0040              out_port endp
```

上述指令的标签名称全部是笔者自行添加的。我们发现，只有下面这个函数调用了 out_port()函数：

```
seg030:0041              sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041
seg030:0041              var_3    = byte ptr -3
seg030:0041              var_2    = word ptr -2
seg030:0041              arg_0    = dword ptr  6
seg030:0041
seg030:0041 C8 04 00 00           enter    4, 0
seg030:0045 56                    push     si
seg030:0046 57                    push     di
seg030:0047 8B 16 82 E7           mov      dx, _in_port_1 ; 0x37A
seg030:004B EC                    in       al, dx
seg030:004C 8A D8                 mov      bl, al
seg030:004E 80 E3 FE              and      bl, 0FEh
seg030:0051 80 CB 04              or       bl, 4
seg030:0054 8A C3                 mov      al, bl
seg030:0056 88 46 FD              mov      [bp+var_3], al
seg030:0059 80 E3 1F              and      bl, 1Fh
seg030:005C 8A C3                 mov      al, bl
seg030:005E EE                    out      dx, al
seg030:005F 68 FF 00              push     0FFh
seg030:0062 0E                    push     cs
seg030:0063 E8 CE FF              call     near ptr out_port
seg030:0066 59                    pop      cx
seg030:0067 68 D3 00              push     0D3h
seg030:006A 0E                    push     cs
seg030:006B E8 C6 FF              call     near ptr out_port
seg030:006E 59                    pop      cx
seg030:006F 33 F6                 xor      si, si
seg030:0071 EB 01                 jmp      short loc_359D4
seg030:0073
seg030:0073              loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46                    inc      si
seg030:0074
seg030:0074              loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00           cmp      si, 96h
seg030:0078 7C F9                 jl       short loc_359D3
seg030:007A 68 C3 00              push     0C3h
seg030:007D 0E                    push     cs
seg030:007E E8 B3 FF              call     near ptr out_port
seg030:0081 59                    pop      cx
seg030:0082 68 C7 00              push     0C7h
seg030:0085 0E                    push     cs
seg030:0086 E8 AB FF              call     near ptr out_port
seg030:0089 59                    pop      cx
seg030:008A 68 D3 00              push     0D3h
seg030:008D 0E                    push     cs
seg030:008E E8 A3 FF              call     near ptr out_port
seg030:0091 59                    pop      cx
seg030:0092 68 C3 00              push     0C3h
seg030:0095 0E                    push     cs
```

```
seg030:0096 E8 9B FF            call    near ptr out_port
seg030:0099 59                  pop     cx
seg030:009A 68 C7 00            push    0C7h
seg030:009D 0E                  push    cs
seg030:009E E8 93 FF            call    near ptr out_port
seg030:00A1 59                  pop     cx
seg030:00A2 68 D3 00            push    0D3h
seg030:00A5 0E                  push    cs
seg030:00A6 E8 8B FF            call    near ptr out_port
seg030:00A9 59                  pop     cx
seg030:00AA BF FF FF            mov     di, 0FFFFh
seg030:00AD EB 40               jmp     short loc_35A4F
seg030:00AF
seg030:00AF             loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00            mov     si, 4
seg030:00B2
seg030:00B2             loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7               shl     di, 1
seg030:00B4 8B 16 80 E7         mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC                  in      al, dx
seg030:00B9 A8 80               test    al, 80h
seg030:00BB 75 03               jnz     short loc_35A20
seg030:00BD 83 CF 01            or      di, 1
seg030:00C0
seg030:00C0             loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+        test    [bp+var_2], 8
seg030:00C5 74 05               jz      short loc_35A2C
seg030:00C7 68 D7 00            push    0D7h ; '+'
seg030:00CA EB 0B               jmp     short loc_35A37
seg030:00CC
seg030:00CC             loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00            push    0C3h
seg030:00CF 0E                  push    cs
seg030:00D0 E8 61 FF            call    near ptr out_port
seg030:00D3 59                  pop     cx
seg030:00D4 68 C7 00            push    0C7h
seg030:00D7
seg030:00D7             loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E                  push    cs
seg030:00D8 E8 59 FF            call    near ptr out_port
seg030:00DB 59                  pop     cx
seg030:00DC 68 D3 00            push    0D3h
seg030:00DF 0E                  push    cs
seg030:00E0 E8 51 FF            call    near ptr out_port
seg030:00E3 59                  pop     cx
seg030:00E4 8B 46 FE            mov     ax, [bp+var_2]
seg030:00E7 D1 E0               shl     ax, 1
seg030:00E9 89 46 FE            mov     [bp+var_2], ax
seg030:00EC 4E                  dec     si
seg030:00ED 75 C3               jnz     short loc_35A12
seg030:00EF
seg030:00EF             loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06            les     bx, [bp+arg_0]
seg030:00F2 FF 46 06            inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07            mov     al, es:[bx]
seg030:00F8 98                  cbw
seg030:00F9 89 46 FE            mov     [bp+var_2], ax
seg030:00FC 0B C0               or      ax, ax
seg030:00FE 75 AF               jnz     short loc_35A0F
seg030:0100 68 FF 00            push    0FFh
seg030:0103 0E                  push    cs
seg030:0104 E8 2D FF            call    near ptr out_port
seg030:0107 59                  pop     cx
seg030:0108 8B 16 82 E7         mov     dx, _in_port_1 ; 0x37A
seg030:010C EC                  in      al, dx
```

```
seg030:010D 8A C8               mov     cl, al
seg030:010F 80 E1 5F            and     cl, 5Fh
seg030:0112 8A C1               mov     al, cl
seg030:0114 EE                  out     dx, al
seg030:0115 EC                  in      al, dx
seg030:0116 8A C8               mov     cl, al
seg030:0118 F6 C1 20            test    cl, 20h
seg030:011B 74 08               jz      short loc_35A85
seg030:011D 8A 5E FD            mov     bl, [bp+var_3]
seg030:0120 80 E3 DF            and     bl, 0DFh
seg030:0123 EB 03               jmp     short loc_35A88
seg030:0125
seg030:0125             loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD            mov     bl, [bp+var_3]
seg030:0128
seg030:0128             loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80            test    cl, 80h
seg030:012B 74 03               jz      short loc_35A90
seg030:012D 80 E3 7F            and     bl, 7Fh
seg030:0130
seg030:0130             loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7         mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3               mov     al, bl
seg030:0136 EE                  out     dx, al
seg030:0137 8B C7               mov     ax, di
seg030:0139 5F                  pop     di
seg030:013A 5E                  pop     si
seg030:013B C9                  leave
seg030:013C CB                  retf
seg030:013C             sent_pro endp
```

可见，这个加密狗还是 Sentinel Pro 出品的“哈希验证”型加密狗。通过观察程序中传递的文本字符串，我们能够查到这种加密狗信息。而且它的 16 位返回值最终要和固定值进行比较。

输出端口地址通常是 0x378，即 USB 问世之前、老式打印机才用的打印终端 LPT 接口。在设计这种接口时，恐怕没有人会想到要从打印机接收数据，因此这种接口是单向通信接口[①]。应用程序能够通过 0x379 端口访问打印机的状态寄存器，获取“缺纸”“确定/ack”“繁忙”之类的信号信息。也就是说，打印机的状态寄存器是主机了解打印机工作状态的唯一途径。因此，加密狗肯定通过这个寄存器获取反馈信息，逐次轮训有关比特位罢了。

源程序的_in_port_2 和_in_port_1 标签，分别访问了状态字寄存器（0x379）和控制寄存器（0x37A）。

看来，通过 seg030:00B9 的指令，程序通过“繁忙”标志为获取返回信息：每个比特位都存储于 DI 寄存器，由函数尾部的指令返回。

那么，发送给输出端口的这些字节都有什么涵义？笔者没有进行深究，只知道是发送给加密狗的指令。而且就这种任务来说，也不必把各个控制指令搞清楚。

检测加密狗的指令如下：

```
00000000 struct_0        struc ; (sizeof=0x1B)
00000000 field_0         db 25 dup(?)            ; string(C)
00000019 _A              dw?
0000001B struct_0        ends

dseg:3CBC 61 63 72 75+_Q    struc_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+        ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+      struc_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+      struc_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+      struc_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+      struc_0 <'paper', 0FFDFh>
```

① 本文指的是 LPT 并行接口。实际上，IEEE 1284 标准允许打印机回传数据。

```
dseg:3E6C 63 6F 6B 65+      struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+      struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+      struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+      struct_0 <'copy', 0F557h>


seg030:0145              check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145              var_6 = dword ptr -6
seg030:0145              var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00        enter   6, 0
seg030:0149 56                 push    si
seg030:014A 66 6A 00           push    large 0          ; newtime
seg030:014D 6A 00              push    0                ; cmd
seg030:014F 9A C1 18 00+       call    _biostime
seg030:0154 52                 push    dx
seg030:0155 50                 push    ax
seg030:0156 66 58              pop     eax
seg030:0158 83 C4 06           add     sp, 6
seg030:015B 66 89 46 FA        mov     [bp+var_6], eax
seg030:015F 66 3B 06 D8+       cmp     eax, _expiration
seg030:0164 7E 44              jle     short loc_35B0A
seg030:0166 6A 14              push    14h
seg030:0168 90                 nop
seg030:0169 0E                 push    cs
seg030:016A E8 52 00           call    near ptr get_rand
seg030:016D 59                 pop     cx
seg030:016E 8B F0              mov     si, ax
seg030:0170 6B C0 1B           imul    ax, 1Bh
seg030:0173 05 BC 3C           add     ax, offset _Q
seg030:0176 1E                 push    ds
seg030:0177 50                 push    ax
seg030:0178 0E                 push    cs
seg030:0179 E8 C5 FE           call    near ptr sent_pro
seg030:017C 83 C4 04           add     sp, 4
seg030:017F 89 46 FE           mov     [bp+var_2], ax
seg030:0182 8B C6              mov     ax, si
seg030:0184 6B C0 12           imul    ax, 18
seg030:0187 66 0F BF C0        movsx   eax, ax
seg030:018B 66 8B 56 FA        mov     edx, [bp+var_6]
seg030:018F 66 03 D0           add     edx, eax
seg030:0192 66 89 16 D8+       mov     _expiration, edx
seg030:0197 8B DE              mov     bx, si
seg030:0199 6B DB 1B           imul    bx, 27
seg030:019C 8B 87 D5 3C        mov     ax, _Q._A[bx]
seg030:01A0 3B 46 FE           cmp     ax, [bp+var_2]
seg030:01A3 74 05              jz      short loc_35B0A
seg030:01A5 B8 01 00           mov     ax, 1
seg030:01A8 EB 02              jmp     short loc_35B0C
seg030:01AA
seg030:01AA              loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                         ; check_dongle+5Ej
seg030:01AA 33 C0              xor     ax, ax
seg030:01AC
seg030:01AC              loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E                 pop     si
seg030:01AD C9                 leave
seg030:01AE CB                 retf
seg030:01AE              check_dongle endp
```

一般来说，应用程序在执行重要功能之前都会检验加密狗。受并行打印口和加密狗硬件特性的共同制约，验证 LPT 加密狗的核实操作肯定非常非常慢。所以多数软件都会存储验证结果，在一段时间内免去验证的烦恼。它们多数会通过 boiostime()函数获取当前时间。

另外，程序还使用了标准 C 函数 get_rand()：

```
seg030:01BF              get_rand proc far ; CODE XREF: check_dongle+25p
seg030:01BF
seg030:01BF              arg_0    =word ptr 6
seg030:01BF
seg030:01BF 55                    push    bp
seg030:01C0 8B EC                 mov     bp, sp
seg030:01C2 9A 3D 21 00+          call    _rand
seg030:01C7 66 0F BF C0           movsx   eax, ax
seg030:01CB 66 0F BF 56+          movsx   edx, [bp+arg_0]
seg030:01D0 66 0F AF C2           imul    eax, edx
seg030:01D4 66 BB 00 80+          mov     ebx, 8000h
seg030:01DA 66 99                 cdq
seg030:01DC 66 F7 FB              idiv    ebx
seg030:01DF 5D                    pop     bp
seg030:01E0 CB                    retf
seg030:01E0              get_rand endp
```

可见，程序会把随机文本字符串发送到加密狗上，然后把加密狗的返回值与正确的哈希值进行比对。

加密狗检测主函数的调用方法如下：

```
seg033:087B 9A 45 01 96+     call    check_dongle
seg033:0880 0B C0            or      ax, ax
seg033:0882 74 62            jz      short OK
seg033:0884 83 3E 60 42+     cmp     word_620E0, 0
seg033:0889 75 5B            jnz     short OK
seg033:088B FF 06 60 42      inc     word_620E0
seg033:088F 1E               push    ds
seg033:0890 68 22 44         push    offset aTrupcRequiresA ; "This Software Requires a Software Lock\n"
seg033:0893 1E               push    ds
seg033:0894 68 60 E9         push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+     call    _strcpy
seg033:089C 83 C4 08         add     sp, 8
seg033:089F 1E               push    ds
seg033:08A0 68 42 44         push    offset aPleaseContactA ; "Please Contact ..."
seg033:08A3 1E               push    ds
seg033:08A4 68 60 E9         push    offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+     call    _strcat
```

由此可知，破解加密狗的方法十分简单；我们只要把 check_dongle()函数的返回值强制设置为 0 即可。例如，不妨在代码的开头处加上下列指令：

```
mov ax,0
retf
```

细心的读者可能会想起 C 语言的 strcpy()函数仅仅需要 2 个参数而已，但是这个指令却传递了 4 个值。

```
seg033:088F 1E                    push    ds
seg033:0890 68 22 44              push    offset aTrupcRequiresA ; "This Softwar ↙
    ↳ Requires a Software Lock\n"
seg033:0893 1E                    push    ds
seg033:0894 68 60 E9              push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+          call    _strcpy
seg033:089C 83 C4 08              add     sp, 8
```

这是 MS-DOS 特有的寻址方式。如需了解详情，请参见第 94 章。

如您所见，在使用 strcpy()和其他函数时，操作系统都是把指针分解为一对 16 位值、然后再进行传递的。

在程序启动时，DS 寄存器的值被设置为当前程序数据段的地址。程序里的字符串信息都存储在数据段里。

在 send_pro()函数里，字符串的每个字节都被加载到 seg030:00EF。与此同时，LES 指令从外部传递的参数里加载 ES:BX 对。地址 seg030:00F5 处的 MOV 指令，在内存中 ES:BX 对所描述的地址上读取字节。

而 seg030:00F2 的指令只对 16 位值进行递增处理，却没有处理的值。这意味着传递给函数的字符串位于两个数据段的地址之外。

第 79 章　“QR9”：魔方态加密模型

非专业的加密系统偶尔会让人感到匪夷所思。

本章将与您共同逆向分析一款已经找不到源代码的数据加密软件。

首先利用 IDA 的导出功能，导出这款程序的指令清单：

```
.text:00541000 set_bit          proc near               ; CODE XREF: rotate1+42
.text:00541000                                          ; rotate2+42 ...
.text:00541000
.text:00541000 arg_0            = dword ptr  4
.text:00541000 arg_4            = dword ptr  8
.text:00541000 arg_8            = dword ptr  0Ch
.text:00541000 arg_C            = byte ptr  10h
.text:00541000
.text:00541000                  mov     al, [esp+arg_C]
.text:00541004                  mov     ecx, [esp+arg_8]
.text:00541008                  push    esi
.text:00541009                  mov     esi, [esp+4+arg_0]
.text:0054100D                  test    al, al
.text:0054100F                  mov     eax, [esp+4+arg_4]
.text:00541013                  mov     dl, 1
.text:00541015                  jz      short loc_54102B
.text:00541017                  shl     dl, cl
.text:00541019                  mov     cl, cube64[eax+esi*8]
.text:00541020                  or      cl, dl
.text:00541022                  mov     cube64[eax+esi*8], cl
.text:00541029                  pop     esi
.text:0054102A                  retn
.text:0054102B
.text:0054102B loc_54102B:                              ; CODE XREF: set_bit+15
.text:0054102B                  shl     dl, cl
.text:0054102D                  mov     cl, cube64[eax+esi*8]
.text:00541034                  not     dl
.text:00541036                  and     cl, dl
.text:00541038                  mov     cube64[eax+esi*8], cl
.text:0054103F                  pop     esi
.text:00541040                  retn
.text:00541040 set_bit          endp
.text:00541040
.text:00541041                  align 10h
.text:00541050
.text:00541050 ; =============== S U B R O U T I N E =============
.text:00541050
.text:00541050
.text:00541050 get_bit          proc near               ; CODE XREF: rotate1+16
.text:00541050                                          ; rotate2+16 ...
.text:00541050
.text:00541050 arg_0            = dword ptr  4
.text:00541050 arg_4            = dword ptr  8
.text:00541050 arg_8            = byte ptr  0Ch
.text:00541050
.text:00541050                  mov     eax, [esp+arg_4]
.text:00541054                  mov     ecx, [esp+arg_0]
.text:00541058                  mov     al, cube64[eax+ecx*8]
.text:0054105F                  mov     cl, [esp+arg_8]
.text:00541063                  shr     al, cl
.text:00541065                  and     al, 1
```

```
.text:00541067                  retn
.text:00541067 get_bit          endp
.text:00541067
.text:00541068                  align 10h
.text:00541070
.text:00541070 ; =========== S U B R O U T I N E ============
.text:00541070
.text:00541070
.text:00541070 rotate1          proc near               ; CODE XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0            = dword ptr  4
.text:00541070
.text:00541070                  sub     esp, 40h
.text:00541073                  push    ebx
.text:00541074                  push    ebp
.text:00541075                  mov     ebp, [esp+48h+arg_0]
.text:00541079                  push    esi
.text:0054107A                  push    edi
.text:0054107B                  xor     edi, edi        ; EDI is loop1 counter
.text:0054107D                  lea     ebx, [esp+50h+internal_array_64]
.text:00541081
.text:00541081 first_loop1_begin:                       ; CODE XREF: rotate1+2E
.text:00541081                  xor     esi, esi        ; ESI is loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:                       ; CODE XREF: rotate1+25
.text:00541083                  push    ebp             ; arg_0
.text:00541084                  push    esi
.text:00541085                  push    edi
.text:00541086                  call    get_bit
.text:0054108B                  add     esp, 0Ch
.text:0054108E                  mov     [ebx+esi], al   ; store to internal array
.text:00541091                  inc     esi
.text:00541092                  cmp     esi, 8
.text:00541095                  jl      short first_loop2_begin
.text:00541097                  inc     edi
.text:00541098                  add     ebx, 8
.text:0054109B                  cmp     edi, 8
.text:0054109E                  jl      short first_loop1_begin
.text:005410A0                  lea     ebx, [esp+50h+internal_array_64]
.text:005410A4                  mov     edi, 7          ; EDI is loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:                      ; CODE XREF: rotate1+57
.text:005410A9                  xor     esi, esi        ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:                      ; CODE XREF: rotate1+4E
.text:005410AB                  mov     al, [ebx+esi]   ; value from internal array
.text:005410AE                  push    eax
.text:005410AF                  push    ebp             ; arg_0
.text:005410B0                  push    edi
.text:005410B1                  push    esi
.text:005410B2                  call    set_bit
.text:005410B7                  add     esp, 10h
.text:005410BA                  inc     esi             ; increment loop2 counter
.text:005410BB                  cmp     esi, 8
.text:005410BE                  jl      short second_loop2_begin
.text:005410C0                  dec     edi             ; decrement loop2 counter
.text:005410C1                  add     ebx, 8
.text:005410C4                  cmp     edi, 0FFFFFFFFh
.text:005410C7                  jg      short second_loop1_begin
.text:005410C9                  pop     edi
.text:005410CA                  pop     esi
.text:005410CB                  pop     ebp
.text:005410CC                  pop     ebx
.text:005410CD                  add     esp, 40h
```

```
.text:005410D0                  retn
.text:005410D0 rotate1          endp
.text:005410D0
.text:005410D1                  align 10h
.text:005410E0
.text:005410E0 ; ============ S U B R O U T I N E =========
.text:005410E0
.text:005410E0
.text:005410E0 rotate2          proc near               ; CODE XREF: rotate_all_with_password+7A
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0            = dword ptr  4
.text:005410E0
.text:005410E0                  sub     esp, 40h
.text:005410E3                  push    ebx
.text:005410E4                  push    ebp
.text:005410E5                  mov     ebp, [esp+48h+arg_0]
.text:005410E9                  push    esi
.text:005410EA                  push    edi
.text:005410EB                  xor     edi, edi        ; loop1 counter
.text:005410ED                  lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:                              ; CODE XREF: rotate2+2E
.text:005410F1                  xor     esi, esi        ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:                              ; CODE XREF: rotate2+25
.text:005410F3                  push    esi             ; loop2
.text:005410F4                  push    edi             ; loop1
.text:005410F5                  push    ebp             ; arg_0
.text:005410F6                  call    get_bit
.text:005410FB                  add     esp, 0Ch
.text:005410FE                  mov     [ebx+esi], al   ; store to internal array
.text:00541101                  inc     esi             ; increment loop1 counter
.text:00541102                  cmp     esi, 8
.text:00541105                  jl      short loc_5410F3
.text:00541107                  inc     edi             ; increment loop2 counter
.text:00541108                  add     ebx, 8
.text:0054110B                  cmp     edi, 8
.text:0054110E                  jl      short loc_5410F1
.text:00541110                  lea     ebx, [esp+50h+internal_array_64]
.text:00541114                  mov     edi, 7          ; loop1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:                              ; CODE XREF: rotate2+57
.text:00541119                  xor     esi, esi        ; loop2 counter
.text:0054111B
.text:0054111B loc_54111B:                              ; CODE XREF: rotate2+4E
.text:0054111B                  mov     al, [ebx+esi]   ; get byte from internal array
.text:0054111E                  push    eax
.text:0054111F                  push    edi             ; loop1 counter
.text:00541120                  push    esi             ; loop2 counter
.text:00541121                  push    ebp             ; arg_0
.text:00541122                  call    set_bit
.text:00541127                  add     esp, 10h
.text:0054112A                  inc     esi             ; increment loop2 counter
.text:0054112B                  cmp     esi, 8
.text:0054112E                  jl      short loc_54111B
.text:00541130                  dec     edi             ; decrement loop2 counter
.text:00541131                  add     ebx, 8
.text:00541134                  cmp     edi, 0FFFFFFFFh
.text:00541137                  jg      short loc_541119
.text:00541139                  pop     edi
.text:0054113A                  pop     esi
.text:0054113B                  pop     ebp
.text:0054113C                  pop     ebx
.text:0054113D                  add     esp, 40h
```

```
.text:00541140                  retn
.text:00541140 rotate2          endp
.text:00541140
.text:00541141                  align 10h
.text:00541150
.text:00541150 ; ============ S U B R O U T I N E ===========
.text:00541150
.text:00541150
.text:00541150 rotate3          proc near               ; CODE XREF: rotate_all_with_password+66
.text:00541150
.text:00541150 var_40           = byte ptr -40h
.text:00541150 arg_0            = dword ptr  4
.text:00541150
.text:00541150                  sub     esp, 40h
.text:00541153                  push    ebx
.text:00541154                  push    ebp
.text:00541155                  mov     ebp, [esp+48h+arg_0]
.text:00541159                  push    esi
.text:0054115A                  push    edi
.text:0054115B                  xor     edi, edi
.text:0054115D                  lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161:                              ; CODE XREF: rotate3+2E
.text:00541161                  xor     esi, esi
.text:00541163
.text:00541163 loc_541163:                              ; CODE XREF: rotate3+25
.text:00541163                  push    esi
.text:00541164                  push    ebp
.text:00541165                  push    edi
.text:00541166                  call    get_bit
.text:0054116B                  add     esp, 0Ch
.text:0054116E                  mov     [ebx+esi], al
.text:00541171                  inc     esi
.text:00541172                  cmp     esi, 8
.text:00541175                  jl      short loc_541163
.text:00541177                  inc     edi
.text:00541178                  add     ebx, 8
.text:0054117B                  cmp     edi, 8
.text:0054117E                  jl      short loc_541161
.text:00541180                  xor     ebx, ebx
.text:00541182                  lea     edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:                              ; CODE XREF: rotate3+54
.text:00541186                  mov     esi, 7
.text:0054118B
.text:0054118B loc_54118B:                              ; CODE XREF: rotate3+4E
.text:0054118B                  mov     al, [edi]
.text:0054118D                  push    eax
.text:0054118E                  push    ebx
.text:0054118F                  push    ebp
.text:00541190                  push    esi
.text:00541191                  call    set_bit
.text:00541196                  add     esp, 10h
.text:00541199                  inc     edi
.text:0054119A                  dec     esi
.text:0054119B                  cmp     esi, 0FFFFFFFFh
.text:0054119E                  jg      short loc_54118B
.text:005411A0                  inc     ebx
.text:005411A1                  cmp     ebx, 8
.text:005411A4                  jl      short loc_541186
.text:005411A6                  pop     edi
.text:005411A7                  pop     esi
.text:005411A8                  pop     ebp
.text:005411A9                  pop     ebx
.text:005411AA                  add     esp, 40h
.text:005411AD                  retn
```

```
.text:005411AD rotate3          endp
.text:005411AD
.text:005411AE                  align 10h
.text:005411B0
.text:005411B0 ; ============ S U B R O U T I N E ===========
.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near          ; CODE XREF: crypt+1F
.text:005411B0                                             ; decrypt+36
.text:005411B0
.text:005411B0 arg_0            = dword ptr  4
.text:005411B0 arg_4            = dword ptr  8
.text:005411B0
.text:005411B0                  mov     eax, [esp+arg_0]
.text:005411B4                  push    ebp
.text:005411B5                  mov     ebp, eax
.text:005411B7                  cmp     byte ptr [eax], 0
.text:005411BA                  jz      exit
.text:005411C0                  push    ebx
.text:005411C1                  mov     ebx, [esp+8+arg_4]
.text:005411C5                  push    esi
.text:005411C6                  push    edi
.text:005411C7
.text:005411C7 loop_begin:                                 ; CODE XREF: rotate_all_with_password+9F
.text:005411C7                  movsx   eax, byte ptr [ebp+0]
.text:005411CB                  push    eax             ; C
.text:005411CC                  call    _tolower
.text:005411D1                  add     esp, 4
.text:005411D4                  cmp     al, 'a'
.text:005411D6                  jl      short next_character_in_password
.text:005411D8                  cmp     al, 'z'
.text:005411DA                  jg      short next_character_in_password
.text:005411DC                  movsx   ecx, al
.text:005411DF                  sub     ecx, 'a'
.text:005411E2                  cmp     ecx, 24
.text:005411E5                  jle     short skip_subtracting
.text:005411E7                  sub     ecx, 24
.text:005411EA
.text:005411EA skip_subtracting:                           ; CODE XREF: rotate_all_with_password+35
.text:005411EA                  mov     eax, 55555556h
.text:005411EF                  imul    ecx
.text:005411F1                  mov     eax, edx
.text:005411F3                  shr     eax, 1Fh
.text:005411F6                  add     edx, eax
.text:005411F8                  mov     eax, ecx
.text:005411FA                  mov     esi, edx
.text:005411FC                  mov     ecx, 3
.text:00541201                  cdq
.text:00541202                  idiv    ecx
.text:00541204                  sub     edx, 0
.text:00541207                  jz      short call_rotate1
.text:00541209                  dec     edx
.text:0054120A                  jz      short call_rotate2
.text:0054120C                  dec     edx
.text:0054120D                  jnz     short next_character_in_password
.text:0054120F                  test    ebx, ebx
.text:00541211                  jle     short next_character_in_password
.text:00541213                  mov     edi, ebx
.text:00541215
.text:00541215 call_rotate3:                               ; CODE XREF: rotate_all_with_password+6F
.text:00541215                  push    esi
.text:00541216                  call    rotate3
.text:0054121B                  add     esp, 4
.text:0054121E                  dec     edi
.text:0054121F                  jnz     short call_rotate3
```

```
.text:00541221                  jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:                                    ; CODE XREF: rotate_all_with_password+5A
.text:00541223                  test    ebx, ebx
.text:00541225                  jle     short next_character_in_password
.text:00541227                  mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:                                      ; CODE XREF: rotate_all_with_password+83
.text:00541229                  push    esi
.text:0054122A                  call    rotate2
.text:0054122F                  add     esp, 4
.text:00541232                  dec     edi
.text:00541233                  jnz     short loc_541229
.text:00541235                  jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:                                    ; CODE XREF: rotate_all_with_password+57
.text:00541237                  test    ebx, ebx
.text:00541239                  jle     short next_character_in_password
.text:0054123B                  mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:                                      ; CODE XREF: rotate_all_with_password+97
.text:0054123D                  push    esi
.text:0054123E                  call    rotate1
.text:00541243                  add     esp, 4
.text:00541246                  dec     edi
.text:00541247                  jnz     short loc_54123D
.text:00541249
.text:00541249 next_character_in_password:                      ; CODE XREF: rotate_all_with_password+26
.text:00541249                                                  ; rotate_all_with_password+2A ...
.text:00541249                  mov     al, [ebp+1]
.text:0054124C                  inc     ebp
.text:0054124D                  test    al, al
.text:0054124F                  jnz     loop_begin
.text:00541255                  pop     edi
.text:00541256                  pop     esi
.text:00541257                  pop     ebx
.text:00541258
.text:00541258 exit:                                            ; CODE XREF: rotate_all_with_password+A
.text:00541258                  pop     ebp
.text:00541259                  retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:0054125A                  align 10h
.text:00541260
.text:00541260 ; ============= S U B R O U T I N E ===============
.text:00541260
.text:00541260
.text:00541260 crypt            proc near               ; CODE XREF: crypt_file+8A
.text:00541260
.text:00541260 arg_0            = dword ptr  4
.text:00541260 arg_4            = dword ptr  8
.text:00541260 arg_8            = dword ptr  0Ch
.text:00541260
.text:00541260                  push    ebx
.text:00541261                  mov     ebx, [esp+4+arg_0]
.text:00541265                  push    ebp
.text:00541266                  push    esi
.text:00541267                  push    edi
.text:00541268                  xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:                                      ; CODE XREF: crypt+41
.text:0054126A                  mov     eax, [esp+10h+arg_8]
.text:0054126E                  mov     ecx, 10h
.text:00541273                  mov     esi, ebx
.text:00541275                  mov     edi, offset cube64
```

```
.text:0054127A                push    1
.text:0054127C                push    eax
.text:0054127D                rep movsd
.text:0054127F                call    rotate_all_with_password
.text:00541284                mov     eax, [esp+18h+arg_4]
.text:00541288                mov     edi, ebx
.text:0054128A                add     ebp, 40h
.text:0054128D                add     esp, 8
.text:00541290                mov     ecx, 10h
.text:00541295                mov     esi, offset cube64
.text:0054129A                add     ebx, 40h
.text:0054129D                cmp     ebp, eax
.text:0054129F                rep movsd
.text:005412A1                jl      short loc_54126A
.text:005412A3                pop     edi
.text:005412A4                pop     esi
.text:005412A5                pop     ebp
.text:005412A6                pop     ebx
.text:005412A7                retn
.text:005412A7 crypt          endp
.text:005412A7
.text:005412A8                align 10h
.text:005412B0
.text:005412B0 ; =========== S U B R O U T I N E ============
.text:005412B0
.text:005412B0
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt        proc near               ; CODE XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0          = dword ptr  4
.text:005412B0 arg_4          = dword ptr  8
.text:005412B0 Src            = dword ptr  0Ch
.text:005412B0
.text:005412B0                mov     eax, [esp+Src]
.text:005412B4                push    ebx
.text:005412B5                push    ebp
.text:005412B6                push    esi
.text:005412B7                push    edi
.text:005412B8                push    eax             ; Src
.text:005412B9                call    __strdup
.text:005412BE                push    eax             ; Str
.text:005412BF                mov     [esp+18h+Src], eax
.text:005412C3                call    __strrev
.text:005412C8                mov     ebx, [esp+18h+arg_0]
.text:005412CC                add     esp, 8
.text:005412CF                xor     ebp, ebp
.text:005412D1
.text:005412D1 loc_5412D1:                            ; CODE XREF: decrypt+58
.text:005412D1                mov     ecx, 10h
.text:005412D6                mov     esi, ebx
.text:005412D8                mov     edi, offset cube64
.text:005412DD                push    3
.text:005412DF                rep movsd
.text:005412E1                mov     ecx, [esp+14h+Src]
.text:005412E5                push    ecx
.text:005412E6                call    rotate_all_with_password
.text:005412EB                mov     eax, [esp+18h+arg_4]
.text:005412EF                mov     edi, ebx
.text:005412F1                add     ebp, 40h
.text:005412F4                add     esp, 8
.text:005412F7                mov     ecx, 10h
.text:005412FC                mov     esi, offset cube64
.text:00541301                add     ebx, 40h
.text:00541304                cmp     ebp, eax
.text:00541306                rep movsd
.text:00541308                jl      short loc_5412D1
```

```
.text:0054130A                  mov     edx, [esp+10h+Src]
.text:0054130E                  push    edx             ; Memory
.text:0054130F                  call    _free
.text:00541314                  add     esp, 4
.text:00541317                  pop     edi
.text:00541318                  pop     esi
.text:00541319                  pop     ebp
.text:0054131A                  pop     ebx
.text:0054131B                  retn
.text:0054131B decrypt          endp
.text:0054131B
.text:0054131C                  align 10h
.text:00541320
.text:00541320 ; ============= S U B R O U T I N E ============
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file       proc near               ; CODE XREF: _main+42
.text:00541320
.text:00541320 Str              = dword ptr  4
.text:00541320 Filename         = dword ptr  8
.text:00541320 password         = dword ptr  0Ch
.text:00541320
.text:00541320                  mov     eax, [esp+Str]
.text:00541324                  push    ebp
.text:00541325                  push    offset Mode     ; "rb"
.text:0054132A                  push    eax             ; Filename
.text:0054132B                  call    _fopen          ; open file
.text:00541330                  mov     ebp, eax
.text:00541332                  add     esp, 8
.text:00541335                  test    ebp, ebp
.text:00541337                  jnz     short loc_541348
.text:00541339                  push    offset Format   ; "Cannot open input file!\n"
.text:0054133E                  call    _printf
.text:00541343                  add     esp, 4
.text:00541346                  pop     ebp
.text:00541347                  retn
.text:00541348
.text:00541348 loc_541348:                              ; CODE XREF: crypt_file+17
.text:00541348                  push    ebx
.text:00541349                  push    esi
.text:0054134A                  push    edi
.text:0054134B                  push    2               ; Origin
.text:0054134D                  push    0               ; Offset
.text:0054134F                  push    ebp             ; File
.text:00541350                  call    _fseek
.text:00541355                  push    ebp             ; File
.text:00541356                  call    _ftell          ; get file size
.text:0054135B                  push    0               ; Origin
.text:0054135D                  push    0               ; Offset
.text:0054135F                  push    ebp             ; File
.text:00541360                  mov     [esp+2Ch+Str], eax
.text:00541364                  call    _fseek          ; rewind to start
.text:00541369                  mov     esi, [esp+2Ch+Str]
.text:0054136D                  and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370                  add     esi, 40h        ; align size to 64-byte border
.text:00541373                  push    esi             ; Size
.text:00541374                  call    _malloc
.text:00541379                  mov     ecx, esi
.text:0054137B                  mov     ebx, eax        ; allocated buffer pointer -> to EBX
.text:0054137D                  mov     edx, ecx
.text:0054137F                  xor     eax, eax
.text:00541381                  mov     edi, ebx
.text:00541383                  push    ebp             ; File
.text:00541384                  shr     ecx, 2
.text:00541387                  rep stosd
```

```
.text:00541389                  mov     ecx, edx
.text:0054138B                  push    1                       ; Count
.text:0054138D                  and     ecx, 3
.text:00541390                  rep stosb                       ; memset (buffer, 0, aligned_size)
.text:00541392                  mov     eax, [esp+38h+Str]
.text:00541396                  push    eax                     ; ElementSize
.text:00541397                  push    ebx                     ; DstBuf
.text:00541398                  call    _fread                  ; read file
.text:0054139D                  push    ebp                     ; File
.text:0054139E                  call    _fclose
.text:005413A3                  mov     ecx, [esp+44h+password]
.text:005413A7                  push    ecx                     ; password
.text:005413A8                  push    esi                     ; aligned size
.text:005413A9                  push    ebx                     ; buffer
.text:005413AA                  call    crypt                   ; do crypt
.text:005413AF                  mov     edx, [esp+50h+Filename]
.text:005413B3                  add     esp, 40h
.text:005413B6                  push    offset aWb              ; "wb"
.text:005413BB                  push    edx                     ; Filename
.text:005413BC                  call    _fopen
.text:005413C1                  mov     edi, eax
.text:005413C3                  push    edi                     ; File
.text:005413C4                  push    1                       ; Count
.text:005413C6                  push    3                       ; Size
.text:005413C8                  push    offset aQr9             ; "QR9"
.text:005413CD                  call    _fwrite                 ; write file signature
.text:005413D2                  push    edi                     ; File
.text:005413D3                  push    1                       ; Count
.text:005413D5                  lea     eax, [esp+30h+Str]
.text:005413D9                  push    4                       ; Size
.text:005413DB                  push    eax                     ; Str
.text:005413DC                  call    _fwrite                 ; write original file size
.text:005413E1                  push    edi                     ; File
.text:005413E2                  push    1                       ; Count
.text:005413E4                  push    esi                     ; Size
.text:005413E5                  push    ebx                     ; Str
.text:005413E6                  call    _fwrite                 ; write encrypted file
.text:005413EB                  push    edi                     ; File
.text:005413EC                  call    _fclose
.text:005413F1                  push    ebx                     ; Memory
.text:005413F2                  call    _free
.text:005413F7                  add     esp, 40h
.text:005413FA                  pop     edi
.text:005413FB                  pop     esi
.text:005413FC                  pop     ebx
.text:005413FD                  pop     ebp
.text:005413FE                  retn
.text:005413FE crypt_file       endp
.text:005413FE
.text:005413FF                  align 10h
.text:00541400
.text:00541400 ; ========== S U B R O U T I N E ==============
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file     proc near               ; CODE XREF: _main+6E
.text:00541400
.text:00541400 Filename         = dword ptr  4
.text:00541400 arg_4            = dword ptr  8
.text:00541400 Src              = dword ptr  0Ch
.text:00541400
.text:00541400                  mov     eax, [esp+Filename]
.text:00541404                  push    ebx
.text:00541405                  push    ebp
.text:00541406                  push    esi
.text:00541407                  push    edi
```

```
.text:00541408                push    offset aRb        ; "rb"
.text:0054140D                push    eax               ; Filename
.text:0054140E                call    _fopen
.text:00541413                mov     esi, eax
.text:00541415                add     esp, 8
.text:00541418                test    esi, esi
.text:0054141A                jnz     short loc_54142E
.text:0054141C                push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                call    _printf
.text:00541426                add     esp, 4
.text:00541429                pop     edi
.text:0054142A                pop     esi
.text:0054142B                pop     ebp
.text:0054142C                pop     ebx
.text:0054142D                retn
.text:0054142E
.text:0054142E loc_54142E:                              ; CODE XREF: decrypt_file+1A
.text:0054142E                push    2                 ; Origin
.text:00541430                push    0                 ; Offset
.text:00541432                push    esi               ; File
.text:00541433                call    _fseek
.text:00541438                push    esi               ; File
.text:00541439                call    _ftell
.text:0054143E                push    0                 ; Origin
.text:00541440                push    0                 ; Offset
.text:00541442                push    esi               ; File
.text:00541443                mov     ebp, eax
.text:00541445                call    _fseek
.text:0054144A                push    ebp               ; Size
.text:0054144B                call    _malloc
.text:00541450                push    esi               ; File
.text:00541451                mov     ebx, eax
.text:00541453                push    1                 ; Count
.text:00541455                push    ebp               ; ElementSize
.text:00541456                push    ebx               ; DstBuf
.text:00541457                call    _fread
.text:0054145C                push    esi               ; File
.text:0054145D                call    _fclose
.text:00541462                add     esp, 34h
.text:00541465                mov     ecx, 3
.text:0054146A                mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                mov     esi, ebx
.text:00541471                xor     edx, edx
.text:00541473                repe cmpsb
.text:00541475                jz      short loc_541489
.text:00541477                push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C                call    _printf
.text:00541481                add     esp, 4
.text:00541484                pop     edi
.text:00541485                pop     esi
.text:00541486                pop     ebp
.text:00541487                pop     ebx
.text:00541488                retn
.text:00541489
.text:00541489 loc_541489:                              ; CODE XREF: decrypt_file+75
.text:00541489                mov     eax, [esp+10h+Src]
.text:0054148D                mov     edi, [ebx+3]
.text:00541490                add     ebp, 0FFFFFFF9h
.text:00541493                lea     esi, [ebx+7]
.text:00541496                push    eax               ; Src
.text:00541497                push    ebp               ; int
.text:00541498                push    esi               ; int
.text:00541499                call    decrypt
.text:0054149E                mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                push    offset aWb_0      ; "wb"
.text:005414A7                push    ecx               ; Filename
```

```
.text:005414A8                  call    _fopen
.text:005414AD                  mov     ebp, eax
.text:005414AF                  push    ebp             ; File
.text:005414B0                  push    1               ; Count
.text:005414B2                  push    edi             ; Size
.text:005414B3                  push    esi             ; Str
.text:005414B4                  call    _fwrite
.text:005414B9                  push    ebp             ; File
.text:005414BA                  call    _fclose
.text:005414BF                  push    ebx             ; Memory
.text:005414C0                  call    _free
.text:005414C5                  add     esp, 2Ch
.text:005414C8                  pop     edi
.text:005414C9                  pop     esi
.text:005414CA                  pop     ebp
.text:005414CB                  pop     ebx
.text:005414CC                  retn
.text:005414CC decrypt_file     endp
```

笔者在分析过程中逐步添加了各标签的名称。

我们从文件头开始分析。第一个函数读取两个文件名和一个密码：

```
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near
.text:00541320
.text:00541320 Str             = dword ptr  4
.text:00541320 Filename        = dword ptr  8
.text:00541320 password        = dword ptr  0Ch
.text:00541320
```

如果不能成功打开明文文件，程序就会进行异常处理：

```
.text:00541320                  mov     eax, [esp+Str]
.text:00541324                  push    ebp
.text:00541325                  push    offset Mode         ; "rb"
.text:0054132A                  push    eax                 ; Filename
.text:0054132B                  call    _fopen              ; open file
.text:00541330                  mov     ebp, eax
.text:00541332                  add     esp, 8
.text:00541335                  test    ebp, ebp
.text:00541337                  jnz     short loc_541348
.text:00541339                  push    offset Format       ; "Cannot open input file!\n"
.text:0054133E                  call    _printf
.text:00541343                  add     esp, 4
.text:00541346                  pop     ebp
.text:00541347                  pop     ebp
.text:00541348
.text:00541348 loc_541348:
```

通过 fseek()/ftell()函数获取文件大小：

```
.text:00541348 push    ebx
.text:00541349 push    esi
.text:0054134A push    edi
.text:0054134B push    2               ; Origin
.text:0054134D push    0               ; Offset
.text:0054134F push    ebp             ; File

; move current file position to the end
.text:00541350 call    _fseek
.text:00541355 push    ebp             ; File
.text:00541356 call    _ftell          ; get current file position
.text:0054135B push    0               ; Origin
.text:0054135D push    0               ; Offset
.text:0054135F push    ebp             ; File
.text:00541360 mov     [esp+2Ch+Str], eax
```

```
; move current file position to the start
.text:00541364 call    _fseek
```

上述指令把文件大小向 64 字节边界对齐。程序所采用的加密算法只能处理 64 字节消息块。它的算法相当直白：把文件尺寸除以 64，舍弃余数，然后把整除结果乘以 64。下述指令的“与”运算起到整除 64 并清除余数的作用，然后加法运算指令把上述整除的商再加上 64。这组指令将使文件大小向 64 字节边界对齐。

```
.text:00541369 mov     esi, [esp+2Ch+Str]
; reset all lowest 6 bits
.text:0054136D and     esi, 0FFFFFFC0h
; align size to 64-byte border
.text:00541370 add     esi, 40h
```

接下来按照上述结果分配缓冲区：

```
.text:00541373                  push    esi             ; Size
.text:00541374                  call    _malloc
```

调用 memset()，即清除缓冲区数据：[①]

```
.text:00541379 mov     ecx, esi
.text:0054137B mov     ebx, eax            ; allocated buffer pointer -> to EBX
.text:0054137D mov     edx, ecx
.text:0054137F xor     eax, eax
.text:00541381 mov     edi, ebx
.text:00541383 push    ebp                 ; File
.text:00541384 shr     ecx, 2
.text:00541387 rep stosd
.text:00541389 mov     ecx, edx
.text:0054138B push    1                   ; Count
.text:0054138D and     ecx, 3
.text:00541390 rep stosb                   ; memset (buffer, 0, aligned_size)
```

调用标准 C 函数 fread()读取文件：

```
.text:00541392                  mov     eax, [esp+38h+Str]
.text:00541396                  push    eax             ; ElementSize
.text:00541397                  push    ebx             ; DstBuf
.text:00541398                  call    _fread          ; read file
.text:0054139D                  push    ebp             ; File
.text:0054139E                  call    _fclose
```

调用 crypt()函数，并且向这个函数传递缓冲区、缓冲区尺寸及密码字符串：

```
.text:005413A3                  mov     ecx, [esp+44h+password]
.text:005413A7                  push    ecx             ; password
.text:005413A8                  push    esi             ; aligned size
.text:005413A9                  push    ebx             ; buffer
.text:005413AA                  call    crypt           ; do crypt
```

创建输出文件。虽然研发人员确实检测了能否成功打开文件，但是他们忘记了检测能否正确创建文件：

```
.text:005413AF                  mov     edx, [esp+50h+Filename]
.text:005413B3                  add     esp, 40h
.text:005413B6                  push    offset aWb      ; "wb"
.text:005413BB                  push    edx             ; Filename
.text:005413BC                  call    _fopen
.text:005413C1                  mov     edi, eax
```

新建文件的句柄（handle）存储在 EDI 寄存器里。然后写上签名“QR9”：

```
.text:005413C3                  push    edi             ; File
.text:005413C4                  push    1               ; Count
```

① 实际上 calloc()函数可替代 malloc()和 memset()两个函数。

```
.text:005413C6                  push    3               ; Size
.text:005413C8                  push    offset aQr9     ; "QR9"
.text:005413CD                  call    _fwrite         ; write file signature
```

标注原始文件的实际大小（未经数据对齐处理的原始值）：

```
.text:005413D2                  push    edi             ; File
.text:005413D3                  push    1               ; Count
.text:005413D5                  lea     eax, [esp+30h+Str]
.text:005413D9                  push    4               ;Size
.text:005413DB                  push    eax             ; Str
.text:005413DC                  call    _fwrite         ; write original file size
```

写入密文的缓冲区：

```
.text:005413E1                  push    edi             ; File
.text:005413E2                  push    1               ; Count
.text:005413E4                  push    esi             ; Size
.text:005413E5                  push    ebx             ; Str
.text:005413E6                  call    _fwrite         ; write encrypted file
```

关闭文件、释放缓冲区：

```
.text:005413EB                  push    edi             ; File
.text:005413EC                  call    _fclose
.text:005413F1                  push    ebx             ; Memory
.text:005413F2                  call    _free
.text:005413F7                  add     esp, 40h
.text:005413FA                  pop     edi
.text:005413FB                  pop     esi
.text:005413FC                  pop     ebx
.text:005413FD                  pop     ebp
.text:005413FE                  retn
.text:005413FE crypt_file       endp
```

通过上面的分析，我们可整理出源代码如下：

```
void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");
```

```
        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);
};
```

解密过程几乎如出一辙：

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file     proc near
.text:00541400
.text:00541400 Filename         = dword ptr  4
.text:00541400 arg_4            = dword ptr  8
.text:00541400 Src              = dword ptr  0Ch
.text:00541400
.text:00541400                  mov     eax, [esp+Filename]
.text:00541404                  push    ebx
.text:00541405                  push    ebp
.text:00541406                  push    esi
.text:00541407                  push    edi
.text:00541408                  push    offset aRb          ; "rb"
.text:0054140D                  push    eax                 ; Filename
.text:0054140E                  call    _fopen
.text:00541413                  mov     esi, eax
.text:00541415                  add     esp, 8
.text:00541418                  test    esi, esi
.text:0054141A                  jnz     short loc_54142E
.text:0054141C                  push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                  call    _printf
.text:00541426                  add     esp, 4
.text:00541429                  pop     edi
.text:0054142A                  pop     esi
.text:0054142B                  pop     ebp
.text:0054142C                  pop     ebx
.text:0054142D                  retn
.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E                  push    2                   ; Origin
.text:00541430                  push    0                   ; Offset
.text:00541432                  push    esi                 ; File
.text:00541433                  call    _fseek
.text:00541438                  push    esi                 ; File
.text:00541439                  call    _ftell
.text:0054143E                  push    0                   ; Origin
.text:00541440                  push    0                   ; Offset
.text:00541442                  push    esi                 ; File
.text:00541443                  mov     ebp, eax
.text:00541445                  call    _fseek
.text:0054144A                  push    ebp                 ; Size
.text:0054144B                  call    _malloc
.text:00541450                  push    esi                 ; File
.text:00541451                  mov     ebx, eax
.text:00541453                  push    1                   ; Count
.text:00541455                  push    ebp                 ; ElementSize
.text:00541456                  push    ebx                 ; DstBuf
.text:00541457                  call    _fread
.text:0054145C                  push    esi                 ; File
.text:0054145D                  call    _fclose
```

检测前三个字节的程序签名：

```
.text:00541462                  add     esp, 34h
```

```
.text:00541465                  mov     ecx, 3
.text:0054146A                  mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                  mov     esi, ebx
.text:00541471                  xor     edx, edx
.text:00541473                  repe cmpsb
.text:00541475                  jz      short loc_541489
```

如果签名有误，则进行错误提示：

```
.text:00541477                  push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C                  call    _printf
.text:00541481                  add     esp, 4
.text:00541484                  pop     edi
.text:00541485                  pop     esi
.text:00541486                  pop     ebp
.text:00541487                  pop     ebx
.text:00541488                  retn
.text:00541489
.text:00541489 loc_541489:
```

调用 decrypt()函数：

```
.text:00541489                  mov     eax, [esp+10h+Src]
.text:0054148D                  mov     edi, [ebx+3]
.text:00541490                  add     ebp, 0FFFFFFF9h
.text:00541493                  lea     esi, [ebx+7]
.text:00541496                  push    eax             ; Src
.text:00541497                  push    ebp             ; int
.text:00541498                  push    esi             ; int
.text:00541499                  call    decrypt
.text:0054149E                  mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                  push    offset aWb_0    ; "wb"
.text:005414A7                  push    ecx             ; Filename
.text:005414A8                  call    _fopen
.text:005414AD                  mov     ebp, eax
.text:005414AF                  push    ebp             ; File
.text:005414B0                  push    1               ; Count
.text:005414B2                  push    edi             ; Size
.text:005414B3                  push    esi             ; Str
.text:005414B4                  call    _fwrite
.text:005414B9                  push    ebp             ; File
.text:005414BA                  call    _fclose
.text:005414BF                  push    ebx             ; Memory
.text:005414C0                  call    _free
.text:005414C5                  add     esp, 2Ch
.text:005414C8                  pop     edi
.text:005414C9                  pop     esi
.text:005414CA                  pop     ebp
.text:005414CB                  pop     ebx
.text:005414CC                  retn
.text:005414CC decrypt_file     endp
```

通过上面的分析，我们可整理出 decrypt_file()的源代码如下：

```
void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };
```

```
        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not encrypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};
```

接下来，我们深入研究 crypt()函数：

```
.text:00541260 crypt           proc near
.text:00541260
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:
```

这段指令从输入缓冲区中读取部分数据（即消息块），把它传递到内部数组。为了方便讨论，笔者把内部数组叫作“cube64”。信息的尺寸存储在 ECX 寄存器里。MOVSD 代表 move 32-bit dword。而 16 个 32 位 dwords 数据就是 64 个字节。

```
.text:0054126A                 mov     eax, [esp+10h+arg_8]
.text:0054126E                 mov     ecx, 10h
.text:00541273                 mov     esi, ebx    ; EBX is pointer within input buffer
.text:00541275                 mov     edi, offset cube64
.text:0054127A                 push    1
.text:0054127C                 push    eax
.text:0054127D                 rep movsd
```

调用 rotate_all_with_password():

```
.text:0054127F                 call    rotate_all_with_password
```

把加密内容从 cube64 复制到缓冲区：

```
.text:00541284                 mov     eax, [esp+18h+arg_4]
```

```
.text:00541288                  mov     edi, ebx
.text:0054128A                  add     ebp, 40h
.text:0054128D                  add     esp, 8
.text:00541290                  mov     ecx, 10h
.text:00541295                  mov     esi, offset cube64
.text:0054129A                  add     ebx, 40h   ; add 64 to input buffer pointer
.text:0054129D                  cmp     ebp, eax   ; EBP contain amount of encrypted data.
.text:0054129F                  rep movsd
```

如果 EBP 不大于传入函数的文件大小，那么继续处理下一个消息块：

```
.text:005412A1                  jl      short loc_54126A
.text:005412A3                  pop     edi
.text:005412A4                  pop     esi
.text:005412A5                  pop     ebp
.text:005412A6                  pop     ebx
.text:005412A7                  retn
.text:005412A7 crypt            endp
```

通过上面的分析，我们可整理出 crypt()的源代码如下：

```
void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};
```

然后我们再分析 rotate_all_with_password()函数。它有两个参数：密码字符串和数字参数。在加密 crypt()函数里数字参数是 1，而在同样调用 rotate_all_with_password()函数的解密 decrypt()函数里，这个数字参数的值为 3。

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0            = dword ptr 4
.text:005411B0 arg_4            = dword ptr 8
.text:005411B0
.text:005411B0                  mov     eax, [esp+arg_0]
.text:005411B4                  push    ebp
.text:005411B5                  mov     ebp, eax
```

检测密码中的当前字符。如果是零字节，就退出：

```
.text:005411B7                  cmp     byte ptr [eax], 0
.text:005411BA                  jz      exit
.text:005411C0                  push    ebx
.text:005411C1                  mov     ebx, [esp+8+arg_4]
.text:005411C5                  push    esi
.text:005411C6                  push    edi
.text:005411C7
.text:005411C7 loop_begin:
```

调用标准 C 函数 tolower()：

```
.text:005411C7                  movsx   eax, byte ptr [ebp+0]
.text:005411CB                  push    eax             ; C
.text:005411CC                  call    _tolower
.text:005411D1                  add     esp, 4
```

这个函数会忽略（跳过）密码中的非拉丁字符。在进行程序测试的时候，加密工具确实忽略了非拉丁字符。

```
.text:005411D4                  cmp     al, 'a'
.text:005411D6                  jl      short next_character_in_password
.text:005411D8                  cmp     al, 'z'
.text:005411DA                  jg      short next_character_in_password
.text:005411DC                  movsx   ecx, al
```

从当前字符的 ASCII 值中减去 97（即字母“a”）：

```
.text:005411DF                  sub     ecx, 'a' ; 97
```

经过刚才的处理，字母 *a* 的值就变成了 0，*b* 的值为 1，*z* 的值为 25:

```
.text:005411E2                  cmp     ecx, 24
.text:005411E5                  jle     short skip_subtracting
.text:005411E7                  sub     ecx, 24
```

似乎“y”和“z”是特例。在执行上述指令之后，“y”变成了 0，“z”变成了 1。这也就是说，26 个拉丁字母最终被转换为了 0～23 之间（共计 24 个）的数字。

```
.text:005411EA
.text:005411EA skip_subtracting:                        ;CODE XREF: rotate_all_ with_password+35
```

接着，它以乘法指令实现除法运算。本书第 41 章介绍过“除以 9”的详细步骤。下面这段指令把密码字符的值除以 3：

```
.text:005411EA                  mov     eax, 55555556h
.text:005411EF                  imul    ecx
.text:005411F1                  mov     eax, edx
.text:005411F3                  shr     eax, 1Fh
.text:005411F6                  add     edx, eax
.text:005411F8                  mov     eax, ecx
.text:005411FA                  mov     esi, edx
.text:005411FC                  mov     ecx, 3
.text:00541201                  cdq
.text:00541202                  idiv    ecx
```

除法运算的余数存储在 EDX 寄存器里：

```
.text:00541204 sub      edx, 0
.text:00541207 jz       short call_rotate1 ; if remainder is zero, go to rotate1
.text:00541209 dec      edx
.text:0054120A jz       short call_rotate2 ; .. if it is 1, go to rotate2
.text:0054120C dec      edx
.text:0054120D jnz      short next_character_in_password
.text:0054120F test     ebx, ebx
.text:00541211 jle      short next_character_in_password
.text:00541213 mov      edi, ebx
```

如果余数为 2，那么就会调用 rotate3()。此时，EDI 寄存器存储的是 rotate_all_with_password()函数的第二个参数。前文介绍过，在加密过程中这个值为 1，在解密过程中这个值为 3。函数进行了循环处理。在加密时，rotate1/2/3 的调用次数与函数的第一个参数相等。

```
.text:00541215 call_rotate3:
.text:00541215                  push    esi
.text:00541216                  call    rotate3
.text:0054121B                  add     esp, 4
.text:0054121E                  dec     edi
.text:0054121F                  jnz     short call_rotate3
.text:00541221                  jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223                  test    ebx, ebx
.text:00541225                  jle     short next_character_in_password
.text:00541227                  mov     edi, ebx
```

```
.text:00541229
.text:00541229 loc_541229:
.text:00541229                  push    esi
.text:0054122A                  call    rotate2
.text:0054122F                  add     esp, 4
.text:00541232                  dec     edi
.text:00541233                  jnz     short loc_541229
.text:00541235                  jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:
.text:00541237                  test    ebx, ebx
.text:00541239                  jle     short next_character_in_password
.text:0054123B                  mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:
.text:0054123D                  push    esi
.text:0054123E                  call    rotate1
.text:00541243                  add     esp, 4
.text:00541246                  dec     edi
.text:00541247                  jnz     short loc_54123D
.text:00541249
```

从密码字符串中提取第二个字符：

```
.text:00541249 next_character_in_password:
.text:00541249                  mov     al, [ebp+1]
```

密码字符串指针递增：

```
.text:0054124C                  inc     ebp
.text:0054124D                  test    al, al
.text:0054124F                  jnz     loop_begin
.text:00541255                  pop     edi
.text:00541256                  pop     esi
.text:00541257                  pop     ebx
.text:00541258
.text:00541258 exit:
.text:00541258                  pop     ebp
.text:00541259                  retn
.text:00541259 rotate_all_with_password endp
```

通过上面的分析，我们可整理出 rotate_all ()的源代码如下：

```
void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
```

```
                        case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
                        };
                };

                p++;
        };
};
```

然后我们再分析 rotate1/2/3 函数。这三个函数都会调用它们之外的两个函数。根据函数功能，笔者把它们命名为 set_bit()和 get_bit()。

首先来分析 get_bit():

```
.text:00541050 get_bit          proc near
.text:00541050
.text:00541050 arg_0            = dword ptr  4
.text:00541050 arg_4            = dword ptr  8
.text:00541050 arg_8            = byte ptr  0Ch
.text:00541050
.text:00541050                  mov     eax, [esp+arg_4]
.text:00541054                  mov     ecx, [esp+arg_0]
.text:00541058                  mov     al, cube64[eax+ecx*8]
.text:0054105F                  mov     cl, [esp+arg_8]
.text:00541063                  shr     al, cl
.text:00541065                  and     al, 1
.text:00541067                  retn
.text:00541067 get_bit          endp
```

上述指令先计算 cube64 的数组索引“arg_4+arg0*8”，然后将数组中的单个字节向右位移 arg_8 位，隔离最低位再返回数值。

接下来我们分析 set_bit()函数：

```
.text:00541000 set_bit          proc near
.text:00541000
.text:00541000 arg_0            = dword ptr  4
.text:00541000 arg_4            = dword ptr  8
.text:00541000 arg_8            = dword ptr  0Ch
.text:00541000 arg_C            = byte ptr  10h
.text:00541000
.text:00541000                  mov     al, [esp+arg_C]
.text:00541004                  mov     ecx, [esp+arg_8]
.text:00541008                  push    esi
.text:00541009                  mov     esi, [esp+4+arg_0]
.text:0054100D                  test    al, al
.text:0054100F                  mov     eax, [esp+4+arg_4]
.text:00541013                  mov     dl, 1
.text:00541015                  jz      short loc_54102B
```

DL 寄存器的值目前为 1。函数将其左移 arg_8 位。例如，如果 arg_8 是 4，那么 DL 寄存器的值将变为 0x10，即二进制的 1000b。

```
.text:00541017                  shl     dl, cl
.text:00541019                  mov     cl, cube64[eax+esi*8]
```

从数组中提取 1 个位，然后进行设置：

```
.text:00541020                  or      cl, dl
```

存储运算结果：

```
.text:00541022                  mov     cube64[eax+esi*8], cl
.text:00541029                  pop     esi
.text:0054102A                  retn
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B                  shl     dl, cl
```

如果 arg_C 的值不是 0：

```
.text:0054102D                  mov     cl, cube64[eax+esi*8]
```

对 DL 的值求“非”。例如，如果前一步位移使 DL 为 0x10(即二进制的 1000b)，在进行非运算之后，它的值将变为 0xEF（即二进制的 11101111b）。

```
.text:00541034                  not     dl
```

下述指令将过滤相关位。如果 DL 寄存器的某个位是 1，那么它会保留 CL 寄存器的对应位；否则将把 CL 寄存器的相关位设置为 0。如果刚才 DL 的值不变，还是 11101111b，那么 CL 的第 5 位（从数权最低位开始数）将变为 0，其余位保持不变。

```
.text:00541036                  and     cl, dl
```

存储运算结果：

```
.text:00541038                  mov     cube64[eax+esi*8], cl
.text:0054103F                  pop     esi
.text:00541040                  retn
.text:00541040 set_bit          endp
```

get_bit()函数的工作模式也差不多。在 arg_C 为 0 的情况下，函数将清除数组中的特定比特位，否则设置相关比特位为 1。

我们已经知道数组占用 64 字节。无论是 set_bit()函数还是 get_bit()函数，它们的前两个参数都相当于 2D 坐标。然后数组变为 8×8 矩阵。

通过上面的分析，我们可整理出 set_bit()和 get_bit()的源代码如下：

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<shift);
        else
                REMOVE_BIT (cube[x][y], 1<<shift);
};

bool get_bit (int x, int y, int shift)
{
        if ((cube[x][y]>>shift)&1==1)
                return 1;
        return 0;
};
```

回顾 rotate1/2/3 函数，可看到：

```
.text:00541070 rotate1          proc near
.text:00541070
```

函数使用局部数据栈给数组分配了 64 个字节：

```
.text:00541070 internal_array_64 = byte ptr -40h
.text:00541070 arg_0             = dword ptr  4
.text:00541070
.text:00541070                  sub     esp, 40h
.text:00541073                  push    ebx
.text:00541074                  push    ebp
.text:00541075                  mov     ebp, [esp+48h+arg_0]
.text:00541079                  push    esi
.text:0054107A                  push    edi
```

```
.text:0054107B                  xor     edi, edi     ;EDI is loop1 counter
```

EBX 寄存器是指向内部数组的指针：

```
.text:0054107D                  lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

函数使用了 2 层循环：

```
.text:00541081 first_loop1_begin:
.text:00541081    xor     esi, esi          ; ESI is loop 2 counter
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083    push    ebp               ; arg_0
.text:00541084    push    esi               ; loop 1 counter
.text:00541085    push    edi               ; loop 2 counter
.text:00541086    call    get_bit
.text:0054108B    add     esp, 0Ch
.text:0054108E    mov     [ebx+esi], al     ; store to internal array
.text:00541091    inc     esi               ; increment loop 1 counter
.text:00541092    cmp     esi, 8
.text:00541095    jl      short first_loop2_begin
.text:00541097    inc     edi               ; increment loop 2 counter

; increment internal array pointer by 8 at each loop 1 iteration
.text:00541098    add     ebx, 8
.text:0054109B    cmp     edi, 8
.text:0054109E    jl      short first_loop1_begin
```

这两个循环的控制变量，不仅取值范围都在 0～7 之间，而且它们分别充当了 get_bit()函数的第一个、第二个参数。而 get_bit()函数使用的第三个参数是 rotate1()函数的唯一一个参数。而后，get_bit()函数的返回值存储在内部数组里。

函数再次制备了指向内部数组的指针：

```
.text:005410A0    lea     ebx, [esp+50h+internal_array_64]
.text:005410A4    mov     edi, 7            ; EDI is loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:
.text:005410A9    xor     esi, esi          ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:
.text:005410AB    mov     al, [ebx+esi]     ; value from internal array
.text:005410AE    push    eax
.text:005410AF    push    ebp               ; arg_0
.text:005410B0    push    edi               ; loop1 counter
.text:005410B1    push    esi               ; loop2 counter
.text:005410B2    call    set_bit
.text:005410B7    add     esp, 10h
.text:005410BA    inc     esi               ; increment loop2 counter
.text:005410BB    cmp     esi, 8
.text:005410BE    jl      short second_loop2_begin
.text:005410C0    dec     edi               ; decrement loop2 counter
.text:005410C1    add     ebx, 8            ; increment pointer in internal array
.text:005410C4    cmp     edi, 0FFFFFFFFh
.text:005410C7    jg      short second_loop1_begin
.text:005410C9    pop     edi
.text:005410CA    pop     esi
.text:005410CB    pop     ebp
.text:005410CC    pop     ebx
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0 rotate1        endp
```

虽然上述代码通过 set_bit()函数把内部数组里的数据复制到了全局数组里，但是数组的排列顺序完全

不一样。第一层循环的循环控制变量从 7 逐渐变为 0，呈递减的变化趋势。

通过上面的分析，我们可整理出 rotate1()的源代码如下：

```
void rotate1 (int v)
{
        bool tmp[8][8];//internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i,j, v);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (j, 7-i, v, tmp[x][y]);
};
```

既然已经搞清了 rotate1()函数，那么我们继续研究 rotate2()函数：

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr  4
.text:005410E0
.text:005410E0     sub     esp, 40h
.text:005410E3     push    ebx
.text:005410E4     push    ebp
.text:005410E5     mov     ebp, [esp+48h+arg_0]
.text:005410E9     push    esi
.text:005410EA     push    edi
.text:005410EB     xor     edi, edi        ; loop 1 counter
.text:005410ED     lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor     esi, esi        ; loop 2 counter
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3     push    esi             ; loop 2 counter
.text:005410F4     push    edi             ; loop 1 counter
.text:005410F5     push    ebp             ; arg_0
.text:005410F6     call    get_bit
.text:005410FB     add     esp, 0Ch
.text:005410FE     mov     [ebx+esi], al   ; store to internal array
.text:00541101     inc     esi             ; increment loop 1 counter
.text:00541102     cmp     esi, 8
.text:00541105     jl      short loc_5410F3
.text:00541107     inc     edi             ; increment loop 2 counter
.text:00541108     add     ebx, 8
.text:0054110B     cmp     edi, 8
.text:0054110E     jl      short loc_5410F1
.text:00541110     lea     ebx, [esp+50h+internal_array_64]
.text:00541114     mov     edi, 7          ; loop 1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor     esi, esi        ; loop 2 counter
.text:0054111B
.text:0054111B loc_54111B:
.text:0054111B     mov     al, [ebx+esi]   ; get byte from internal array
.text:0054111E     push    eax
.text:0054111F     push    edi             ; loop 1 counter
.text:00541120     push    esi             ; loop 2 counter
.text:00541121     push    ebp             ; arg_0
.text:00541122     call    set_bit
.text:00541127     add     esp, 10h
```

```
.text:0054112A     inc     esi             ; increment loop 2 counter
.text:0054112B     cmp     esi, 8
.text:0054112E     jl      short loc_54111B
.text:00541130     dec     edi             ; decrement loop 2 counter
.text:00541131     add     ebx, 8
.text:00541134     cmp     edi, 0FFFFFFFFh
.text:00541137     jg      short loc_541119
.text:00541139     pop     edi
.text:0054113A     pop     esi
.text:0054113B     pop     ebp
.text:0054113C     pop     ebx
.text:0054113D     add     esp, 40h
.text:00541140     retn
.text:00541140 rotate2 endp
```

在调用 get_bit()和 set_bit()时，rotate1/2 以不同的顺序传递参数。除此之外，rotate1/2 基本相同。可得 rotate1()的源代码如下：

```
void rotate2 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (v, i, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (v, j, 7-i, tmp[i][j]);
};
```

下面我们分析 rotate3()函数：

```
void rotate3 (int v)
{
        bool tmp[8][8];
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, v, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (7-j, v, i, tmp[i][j]);
};
```

这个函数更容易分析。如果把 cube64 当作 8×8×8 的 3D 立方体、每个元素都是一个比特位，那么 get_bit()函数和 set_bit()函数的作用就是从相应坐标读取 1 个比特位。

如此来看，rotate1/2/3 的作用就是以特定隔层旋转所有比特位。这三个参数分别旋转魔方的不同平面，而参数 *v* 代表的是设置隔层的位置（0～7）。

或许，这个算法的作者就是把数据当作 8×8×8 的魔方来处理的。

再次整理 decrypt()函数的分析结果后，可推导出下列源代码：

```
void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
```

```
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};
```

该函数通过标准 C 函数 strrev()[①]对密码字符串进行逆序排列，而且传递给 rotate_all()函数的参数是 3。除此之外，它和 crypt()函数基本一致。

因此，解密过程调用了 3 次 rotate1/2/3。

这和魔方的玩法没什么区别！如果要撤销调整魔方的操作，那么就要用相反的次序和相反方向进行等幅操作。举例来说，您刚刚顺时针转动了魔方的某层方块、后来又要撤销那次操作，那么就把那层方块顺时针转回来，要不然就再顺时针旋转 3 次。

如果说 rotate1()函数旋转的是魔方的“正”面，那么 rotate2()函数旋转的则是“上”面，而 rotate3()函数则用于旋转魔方的侧面。

不妨再回顾一下 rotate_all()函数的代码：

```
q=c-'a';
if (q>24)
        q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
};
```

综合上述分析可知：密码中的每个字符都描述了旋转的“面”（0～2）和“层”（0～7）信息。3 个面×8 个层＝24（密码字节的取值范围）。为了把 26 个字母映射到 24 个数据元素，该程序把字母表的最后两个字母重新映射为 0 和 1。

但是这种算法非常脆弱：在使用短密码的情况下，加密文件的密文篇幅和明文原文的尺寸相等。

重新分析整个程序，可整理得源代码如下：

```
#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<z);
        else
                REMOVE_BIT (cube[x][y], 1<<z);
};
```

① 请参考 MSDN 的说明：https://msdn.microsoft.com/en-us/library/9hby7w40(VS.80).aspx。

```
bool get_bit (int x, int y, int z)
{
        if ((cube[x][y]>>z)&1==1)
                return true;
        return false;
};

void rotate_f (int row)
{
        bool tmp[8][8];
        int x, y;

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        tmp[x][y]=get_bit (x, y, row);

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
        bool tmp[8][8];
        int y, z;

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        tmp[y][z]=get_bit (row, y, z);

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
        bool tmp[8][8];
        int x, z;

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        tmp[x][z]=get_bit (x, row, z);

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;
```

```
                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate_f (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate_t (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate_l (quotient); break;
                        };
                };
                p++;
        };
};

void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        };
        while (i<sz);
};

void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;
```

```
        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);

};

void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not encrypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};

// run: input output 0/1 password
```

```
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
        if (argc!=5)
        {
                printf ("Incorrect parameters!\n");
                return 1;
        };

        if (strcmp (argv[3], "0")==0)
                crypt_file (argv[1], argv[2], argv[4]);
        else
                if (strcmp (argv[3], "1")==0)
                        decrypt_file (argv[1], argv[2], argv[4]);
                else
                        printf ("Wrong param %s\n", argv[3]);

        return 0;
};
```

第 80 章　SAP

80.1　关闭客户端的网络数据包压缩功能

根据公开资料的记载，默认情况下，SAP GUI（客户端）和 SAP 服务端之间的通信是压缩通信，而非加密通信。[①]

决定 SAP GUI（客户端）是否采用网络数据包压缩功能的环境变量是 TDW_NOCOMPRESS。若把这个变量设置为 1，那么就可以关闭客户端的网络数据压缩功能。不过，如果当真进行了这种设置，客户端程序就会弹出如图 80.1 所示的提示窗口，而且这个弹出窗口根本关不掉。

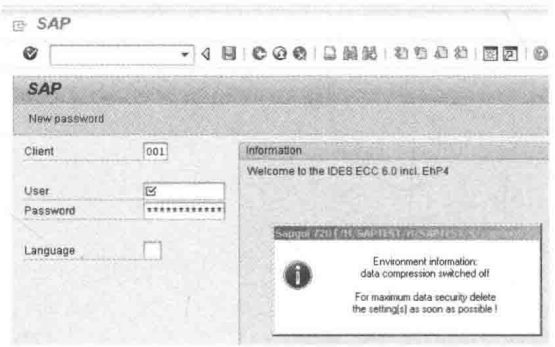


图 80.1　截屏

本节的任务就是去除这个提示窗口。

在此之前，我们可以确定的事实是：

① 在 SAP GUI 客户端程序里，环境变量 TDW_NOCOMPRESS 被设为 1。

② 程序的某个文件里应当存在字符串“data compression switched off”。

借助文件管理器 FAR 程序[②]，我们可在 SAPguilib.dll 里找到这个字符串。

然后，我们用 IDA 程序打开文件 SAPguilib.dll，在文件里搜索字符串“TDW_NOCOMPRESS”。所幸的是，我们可以找到这个字符串，而且整个文件只有一处指令调用了这个字符串。

该文件的相关指令为：[③]

```
.text:6440D51B                lea     eax, [ebp+2108h+var_211C]
.text:6440D51E                push    eax             ; int
.text:6440D51F                push    offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524                mov     byte ptr [edi+15h], 0
.text:6440D528                call    chk_env
.text:6440D52D                pop     ecx
.text:6440D52E                pop     ecx
.text:6440D52F                push    offset byte_64443AF8
.text:6440D534                lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537                call    ds:mfc90_1603
.text:6440D53D                test    eax, eax
.text:6440D53F                jz      short loc_6440D55A
.text:6440D541                lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544                call    ds:mfc90_910
.text:6440D54A                push    eax             ; Str
.text:6440D54B                call    ds:atoi
.text:6440D551                test    eax, eax
```

① 关于 SAP 采用的密码传输和指令传输技术，请参见笔者的博客：http://blog.yurichev.com/node/44http://blog.yurichev.com/node/47。

② http://www.farmanager.com/。

③ 本章演示的程序是 SAP GUI v720 for Win32，其版本号为 7200,1,0,9009。如果版本不同，那么相关偏移量应当会是不同值。

```
.text:6440D553                  setnz   al
.text:6440D556                  pop     ecx
.text:6440D557                  mov     [edi+15h], al
```

据此可知，chk_env()函数通过第二个参数获取环境变量字符串，然后 MFC 字符串处理函数会继续处理这个字符串，接下来标准 C 函数 atoi()（从字符串转换为数字）再从这个字符串里提取数值。最终，程序会把解析出来的环境变量的变量值存储在地址 edi+15h 里。

接下来，我们一起分析 chk_env()函数[①]：

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env          proc near
.text:64413F20
.text:64413F20 DstSize          = dword ptr -0Ch
.text:64413F20 var_8            = dword ptr -8
.text:64413F20 DstBuf           = dword ptr -4
.text:64413F20 VarName          = dword ptr  8
.text:64413F20 arg_4            = dword ptr  0Ch
.text:64413F20
.text:64413F20                  push    ebp
.text:64413F21                  mov     ebp, esp
.text:64413F23                  sub     esp, 0Ch
.text:64413F26                  mov     [ebp+DstSize], 0
.text:64413F2D                  mov     [ebp+DstBuf], 0
.text:64413F34                  push    offset unk_6444C88C
.text:64413F39                  mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C                  call    ds:mfc90_820
.text:64413F42                  mov     eax, [ebp+VarName]
.text:64413F45                  push    eax             ; VarName
.text:64413F46                  mov     ecx, [ebp+DstSize]
.text:64413F49                  push    ecx             ; DstSize
.text:64413F4A                  mov     edx, [ebp+DstBuf]
.text:64413F4D                  push    edx             ; DstBuf
.text:64413F4E                  lea     eax, [ebp+DstSize]
.text:64413F51                  push    eax             ; ReturnSize
.text:64413F52                  call    ds:getenv_s
.text:64413F58                  add     esp, 10h
.text:64413F5B                  mov     [ebp+var_8], eax
.text:64413F5E                  cmp     [ebp+var_8], 0
.text:64413F62                  jz      short loc_64413F68
.text:64413F64                  xor     eax, eax
.text:64413F66                  jmp     short loc_64413FBC
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68                  cmp     [ebp+DstSize], 0
.text:64413F6C                  jnz     short loc_64413F72
.text:64413F6E                  xor     eax, eax
.text:64413F70                  jmp     short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72                  jmp     short loc_64413FBC
.text:64413F75                  push    ecx
.text:64413F76                  mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79                  call    ds:mfc90_2691
.text:64413F7F                  mov     [ebp+DstBuf], eax
.text:64413F82                  mov     edx, [ebp+VarName]
.text:64413F85                  push    edx             ; VarName
.text:64413F86                  mov     eax, [ebp+DstSize]
.text:64413F89                  push    eax             ; DstSize
```

① 这个函数名称是笔者命名的。

```
.text:64413F8A                  mov     ecx, [ebp+DstBuf]
.text:64413F8D                  push    ecx             ; DstBuf
.text:64413F8E                  lea     edx, [ebp+DstSize]
.text:64413F91                  push    edx             ; ReturnSize
.text:64413F92                  call    ds:getenv_s
.text:64413F98                  add     esp, 10h
.text:64413F9B                  mov     [ebp+var_8], eax
.text:64413F9E                  push    0FFFFFFFFh
.text:64413FA0                  mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3                  call    ds:mfc90_5835
.text:64413FA9                  cmp     [ebp+var_8], 0
.text:64413FAD                  jz      short loc_64413FB3
.text:64413FAF                  xor     eax, eax
.text:64413FB1                  jmp     short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3                  mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64413FB6                  call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC                  mov     esp, ebp
.text:64413FBE                  mov     esp, ebp
.text:64413FBF                  retn
.text:64413FBF chk_env          endp
```

其中，getenv_s()函数[①]是微软退出的改进版 getenv()函数[②]，它提升了原有函数的安全特性。

这个函数多处调用了 MFC 字符串处理函数。

不仅如此，它还检测了其他的环境变量。如果打开 SAP GUI（客户端）程序的日志记录功能，就会在 trace log 里看到它检测的环境变量，如下表所示。

DPTRACE	“GUI-OPTION: Trace set to %d”
TDW_HEXDUMP	“GUI-OPTION: Hexdump enabled”
TDW_WORKDIR	“GUI-OPTION: working directory ‘%s’ ”
TDW_SPLASHSRCEENOFF	“GUI-OPTION: Splash Screen Off” / “GUI-OPTION: Splash Screen On”
TDW_REPLYTIMEOUT	“GUI-OPTION: reply timeout %d milliseconds”
TDW_PLAYBACKTIMEOUT	“GUI-OPTION: PlaybackTimeout set to %d milliseconds”
TDW_NOCOMPRESS	“GUI-OPTION: no compression read”
TDW_EXPERT	“GUI-OPTION: expert mode”
TDW_PLAYBACKPROGRESS	“GUI-OPTION: PlaybackProgress”
TDW_PLAYBACKNETTRAFFIC	“GUI-OPTION: PlaybackNetTraffic”
TDW_PLAYLOG	“GUI-OPTION: /PlayLog is YES, file %s”
TDW_PLAYTIME	“GUI-OPTION: /PlayTime set to %d milliseconds”
TDW_LOGFILE	“GUI-OPTION: TDW_LOGFILE ‘%s’ ”
TDW_WAN	“GUI-OPTION: WAN - low speed connection enabled”
TDW_FULLMENU	“GUI-OPTION: FullMenu enabled”
SAP_CP / SAP_CODEPAGE	“GUI-OPTION: SAP_CODEPAGE ‘%d’ ”
UPDOWNLOAD_CP	“GUI-OPTION: UPDOWNLOAD_CP ‘%d’ ”
SNC_PARTNERNAME	“GUI-OPTION: SNC name ‘%s’ ”

① https://msdn.microsoft.com/en-us/library/tb2sfw2z(VS.80).aspx。

② 返回环境变量的标准 C 函数。

续表

SNC_QOP	"GUI-OPTION: SNC_QOP '%s'"
SNC_LIB	"GUI-OPTION: SNC is set to: %s"
SAPGUI_INPLACE	"GUI-OPTION: environment variable SAPGUI_INPLACE is on"

函数把这些变量都存储在数组里，而且把 EDI 寄存器当作这个数组的指针。在调用 chk_evn()函数之前，程序首先设置了 EDI 的值：

```
.text:6440EE00                lea     edi, [ebp+2884h+var_2884] ; options here like +0x15...
.text:6440EE03                lea     ecx, [esi+24h]
.text:6440EE06                call    load_command_line
.text:6440EE0B                mov     edi, eax
.text:6440EE0D                xor     ebx, ebx
.text:6440EE0F                cmp     edi, ebx
.text:6440EE11                jz      short loc_6440EE42
.text:6440EE13                push    edi
.text:6440EE14                push    offset aSapguiStoppedA ; "Sapgui stopped after ↙
   ↳ commandline interp"...
.text:6440EE19                push    dword_644F93E8
.text:6440EE1F                call    FEWTraceError
```

那么，我们关注的“data record mode switched on”字符串在这个文件里吗？整个文件里，只有 CDwsGui::PrepareInfoWindow()构造函数调用了这个字符串。我们可通过日志文件的调试调用（debugging calls）信息了解各个 class/method 的名字。

例如，下述调试调用信息就透露了构造函数的函数名称：

```
.text:64405160                push    dword ptr [esi+2854h]
.text:64405166                push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: ↙
   ↳ sapgui env"...
.text:6440516B                push    dword ptr [esi+2848h]
.text:64405171                call    dbg
.text:64405176                add     esp, 0Ch
```

以及：

```
.text:6440237A                push    eax
.text:6440237B                push    offset aCclientStart_6 ; "CClient::Start: set shortcut ↙
   ↳ user to '\%"...
.text:64402380                push    dword ptr [edi+4]
.text:64402383                call    dbg
.text:64402388                add     esp, 0Ch
```

这些信息的作用很大。

接下来，我们直奔那个令人恼火的弹出窗口：

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam        = byte ptr -3Ch
.text:64404F4F var_38         = dword ptr -38h
.text:64404F4F var_34         = dword ptr -34h
.text:64404F4F rc             = tagRECT ptr -2Ch
.text:64404F4F cy             = dword ptr -1Ch
.text:64404F4F h              = dword ptr -18h
.text:64404F4F var_14         = dword ptr -14h
.text:64404F4F var_10         = dword ptr -10h
.text:64404F4F var_4          = dword ptr -4
.text:64404F4F
.text:64404F4F                push    30h
.text:64404F51                mov     eax, offset loc_64438E00
.text:64404F56                call    __EH_prolog3
.text:64404F5B                mov     esi, ecx        ; ECX is pointer to object
.text:64404F5D                xor     ebx, ebx
```

```
.text:64404F5F                  lea     ecx, [ebp+var_14]
.text:64404F62                  mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65                  call    ds:mfc90_316
.text:64404F6B                  mov     [ebp+var_4], ebx
.text:64404F6E                  lea     edi, [esi+2854h]
.text:64404F74                  push    offset aEnvironmentInf ; "Environment information:\n"
.text:64404F79                  mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B                  call    ds:mfc90_820
.text:64404F81                  cmp     [esi+38h], ebx
.text:64404F84                  mov     ebx, ds:mfc90_2539
.text:64404F8A                  jbe     short loc_64404FA9
.text:64404F8C                  push    dword ptr [esi+34h]
.text:64404F8F                  lea     eax, [ebp+var_14]
.text:64404F92                  push    offset aWorkingDirecto ; "working directory: '\%s'\n"
.text:64404F97                  push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98                  call    ebx ; mfc90_2539
.text:64404F9A                  add     esp, 0Ch
.text:64404F9D                  lea     eax, [ebp+var_14]
.text:64404FA0                  push    eax
.text:64404FA1                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3                  call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9                  mov     eax, [esi+38h]
.text:64404FAC                  test    eax, eax
.text:64404FAE                  jbe     short loc_64404FD3
.text:64404FB0                  push    eax
.text:64404FB1                  lea     eax, [ebp+var_14]
.text:64404FB4                  push    offset aTraceLevelDAct ; "trace level \%d activated\n"
.text:64404FB9                  push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA                  call    ebx ; mfc90_2539
.text:64404FBC                  add     esp, 0Ch
.text:64404FBF                  lea     eax, [ebp+var_14]
.text:64404FC2                  push    eax
.text:64404FC3                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FC5                  call    ds:mfc90_941
.text:64404FCB                  xor     ebx, ebx
.text:64404FCD                  inc     ebx
.text:64404FCE                  mov     [ebp+var_10], ebx
.text:64404FD1                  jmp     short loc_64404FD6
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3                  xor     ebx, ebx
.text:64404FD5                  inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6                  cmp     [esi+38h], ebx
.text:64404FD9                  jbe     short loc_64404FF1
.text:64404FDB                  cmp     dword ptr [esi+2978h], 0
.text:64404FE2                  jz      short loc_64404FF1
.text:64404FE4                  push    offset aHexdumpInTrace ; "hexdump in trace activated\n"
.text:64404FE9                  mov     ecx, edi
```

```
; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB                  call    ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1                  cmp     byte ptr [esi+78h], 0
.text:64404FF5                  jz      short loc_64405007
.text:64404FF7                  push    offset aLoggingActivat ; "logging activated\n"
.text:64404FFC                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE                  call    ds:mfc90_945
.text:64405004                  mov     [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007                  cmp     byte ptr [esi+3Dh], 0
.text:6440500B                  jz      short bypass
.text:6440500D                  push    offset aDataCompressio ;"data compression switched off\n"
.text:64405012                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                  call    ds:mfc90_945
.text:6440501A                  mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D                  mov     eax, [esi+20h]
.text:64405020                  test    eax, eax
.text:64405022                  jz      short loc_6440503A
.text:64405024                  cmp     dword ptr [eax+28h], 0
.text:64405028                  jz      short loc_6440503A
.text:6440502A                  push    offset aDataRecordMode ; "data record mode switched on\n"
.text:6440502F                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031                  call    ds:mfc90_945
.text:64405037                  mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A                  mov     ecx, edi
.text:6440503C                  cmp     [ebp+var_10], ebx
.text:6440503F                  jnz     loc_64405142
.text:64405045                  push    offset aForMaximumData ; "\nFor maximum data security ↙
   ↳ delete\nthe s"

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A                  call    ds:mfc90_945
.text:64405050                  xor     edi, edi
.text:64405052                  push    edi             ; fWinIni
.text:64405053                  lea     eax, [ebp+pvParam]
.text:64405056                  push    eax             ; PvParam
.text:64405057                  push    edi             ; uiParam
.text:64405058                  push    30h             ; uiAction
.text:6440505A                  call    ds:SystemParametersInfoA
.text:64405060                  mov     eax, [ebp+var_34]
.text:64405063                  cmp     eax, 1600
.text:64405068                  jle     short loc_64405072
.text:6440506A                  cdq
.text:6440506B                  sub     eax, edx
.text:6440506D                  sar     eax, 1
.text:6440506F                  mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072                  push    edi             ; hWnd
.text:64405073                  mov     [ebp+cy], 0A0h
```

```
.text:6440507A                  call    ds:GetDC
.text:64405080                  mov     [ebp+var_10], eax
.text:64405083                  mov     ebx, 12Ch
.text:64405088                  cmp     eax, edi
.text:6440508A                  jz      loc_64405113
.text:64405090                  push    11h             ; i
.text:64405092                  call    ds:GetStockObject
.text:64405098                  mov     edi, ds:SelectObject
.text:6440509E                  push    eax             ;h
.text:6440509F                  push    [ebp+var_10]    ; hdc
.text:644050A2                  call    edi ; SelectObject
.text:644050A4                  and     [ebp+rc.left], 0
.text:644050A8                  and     [ebp+rc.top], 0
.text:644050AC                  mov     [ebp+h], eax
.text:644050AF                  push    401h            ; format
.text:644050B4                  lea     eax, [ebp+rc]
.text:644050B7                  push    eax             ; lprc
.text:644050B8                  lea     ecx, [esi+2854h]
.text:644050BE                  mov     [ebp+rc.right], ebx
.text:644050C1                  mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8                  call    ds:mfc90_3178
.text:644050CE                  push    eax             ; cchText
.text:644050CF                  lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5                  call    ds:mfc90_910
.text:644050DB                  push    eax             ; lpchText
.text:644050DC                  push    [ebp+var_10]    ; hdc
.text:644050DF                  call    ds:DrawTextA
.text:644050E5                  push    4               ; nIndex
.text:644050E7                  call    ds:GetSystemMetrics
.text:644050ED                  mov     ecx, [ebp+rc.bottom]
.text:644050F0                  sub     ecx, [ebp+rc.top]
.text:644050F3                  cmp     [ebp+h], 0
.text:644050F7                  lea     eax, [eax+ecx+28h]
.text:644050FB                  mov     [ebp+cy], eax
.text:644050FE                  jz      short loc_64405108
.text:64405100                  push    [ebp+h]         ; h
.text:64405103                  push    [ebp+var_10]    ; hdc
.text:64405106                  call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108                  push    [ebp+var_10]    ; hDC
.text:6440510B                  push    0               ; hWnd
.text:6440510D                  call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113                  mov     eax, [ebp+var_38]
.text:64405116                  push    80h             ; uFlags
.text:6440511B                  push    [ebp+cy]        ; cy
.text:6440511E                  inc     eax
.text:6440511F                  push    ebx             ; cx
.text:64405120                  push    eax             ; Y
.text:64405121                  mov     eax, [ebp+var_34]
.text:64405124                  add     eax, 0FFFFFED4h
.text:64405129                  cdq
.text:6440512A                  sub     eax, edx
.text:6440512C                  sar     eax, 1
.text:6440512E                  push    eax             ; X
.text:6440512F                  push    0               ; hWndInsertAfter
.text:64405131                  push    dword ptr [esi+285Ch] ; hWnd
.text:64405137                  call    ds:SetWindowPos
.text:6440513D                  xor     ebx, ebx
.text:6440513F                  inc     ebx
```

```
.text:64405140                  jmp     short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
.text:64405142                  push    offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147                  call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D                  cmp     dword_6450B970, ebx
.text:64405153                  jl      short loc_64405188
.text:64405155                  call    sub_6441C910
.text:6440515A                  mov     dword_644F858C, ebx
.text:64405160                  push    dword ptr [esi+2854h]
.text:64405166                  push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: ↙
    ↳ sapgui env"...
.text:6440516B                  push    dword ptr [esi+2848h]
.text:64405171                  call    dbg
.text:64405176                  add     esp, 0Ch
.text:64405179                  mov     dword_644F858C, 2
.text:64405183                  call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188                  or      [ebp+var_4], 0FFFFFFFFh
.text:6440518C                  lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F                  call    ds:mfc90_601
.text:64405195                  call    __EH_epilog3
.text:6440519A                  retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

在执行上述函数的最初几个指令时，数据对象（thiscall）的指针存储于 ECX 寄存器。[①]本例中，对象明显使用了 CDwsGui 类。它所记录的选项开关，直接决定着函数窗口的提示信息。

如果地址 this+0x3D 的值不是 0，那么整个程序就会关闭网络数据的压缩功能：

```
.text:64405007 loc_64405007:
.text:64405007                  cmp     byte ptr [esi+3Dh], 0
.text:6440500B                  jz      short bypass
.text:6440500D                  push    offset aDataCompressio ; "data compression switched off\n"
.text:64405012                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                  call    ds:mfc90_945
.text:6440501A                  mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

更有意思的是，决定程序是否显示提示窗口的关键因素是变量 var_10：

```
.text:6440503C                  cmp     [ebp+var_10], ebx
.text:6440503F                  jnz     exit ; bypass drawing

; add strings "For maximum data security delete" / "the setting(s) as soon as possible !":

.text:64405045                  push    offset aForMaximumData ; "\nFor maximum data security ↙
    ↳ delete\nthe s"...
.text:6440504A                  call    ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)
.text:64405050                  xor     edi, edi
.text:64405052                  push    edi             ; fWinIni
.text:64405053                  lea     eax, [ebp+pvParam]
.text:64405056                  push    eax             ; pvParam
```

① 这属于 thiscall 类型的函数。有关 thiscall 类型的函数，可参见本书 51.1.1 节。

```
.text:64405057                  push   edi              ; uiParam
.text:64405058                  push   30h              ; uiAction
.text:6440505A                  call   ds:SystemParametersInfoA
.text:64405060                  mov    eax, [ebp+var_34]
.text:64405063                  cmp    eax, 1600
.text:64405068                  jle    short loc_64405072
.text:6440506A                  cdq
.text:6440506B                  sub    eax, edx
.text:6440506D                  sar    eax, 1
.text:6440506F                  mov    [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:

start drawing:

.text:64405072                  push   edi              ; hWnd
.text:64405073                  mov    [ebp+cy], 0A0h
.text:6440507A                  call   ds:GetDC
```

那么，我们通过实践来验证刚才这些推测吧。

首先找到这个 JNZ 指令：

```
.text:6440503F                  jnz    exit ; bypass drawing
```

把它改为 JMP 之后，SAPGUI 程序就再也不会显示恼人的提示窗口了！

下一步，我们找到 load_command_line()函数（函数名称是笔者命名的名字）里偏移量为 0x15 的数据，以及 CDwsGui::PrepareInfoWindow 里的变量 this+0x3D。这两个值是相等的值么？

为了验证这一猜测，笔者在程序里搜索与偏移量 0x15 有关的全部指令。在 SAPGUI 这样的小型程序里，某个规定变量一般只会被同一个文件调用；换而言之，我们不必检索其他文件。

在当前文件里，第一处赋值的指令如下：

```
.text:64404C19 sub_64404C19     proc near
.text:64404C19
.text:64404C19 arg_0            = dword ptr  4
.text:64404C19
.text:64404C19                  push    ebx
.text:64404C1A                  push    ebp
.text:64404C1B                  push    esi
.text:64404C1C                  push    edi
.text:64404C1D                  mov     edi, [esp+10h+arg_0]
.text:64404C21                  mov     eax, [edi]
.text:64404C23                  mov     esi, ecx ; ESI/ECX are pointers to some unknown object.
.text:64404C25                  mov     [esi], eax
.text:64404C27                  mov     eax, [edi+4]
.text:64404C2A                  mov     [esi+4], eax
.text:64404C2D                  mov     eax, [edi+8]
.text:64404C30                  mov     [esi+8], eax
.text:64404C33                  lea     eax, [edi+0Ch]
.text:64404C36                  push    eax
.text:64404C37                  lea     ecx, [esi+0Ch]

; demangled name:  ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A                  call    ds:mfc90_817
.text:64404C40                  mov     eax, [edi+10h]
.text:64404C43                  mov     [esi+10h], eax
.text:64404C46                  mov     al, [edi+14h]
.text:64404C49                  mov     [esi+14h], al
.text:64404C4C                  mov     al, [edi+15h] ; copy byte from 0x15 offset
.text:64404C4F                  mov     [esi+15h], al ; to 0x15 offset in CDwsGui object
```

上述函数的调用方函数是 CDwsGui::CopyOptions。这些名字都是参照调试信息命名的。

但是在整理程序流程之后，我们会发现在“时间上”第一次调用上述函数的调用方函数是 CDWsGui::Init()：

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                  mov     eax, [ebp+arg_0]
.text:6440B0C2                  push    [ebp+arg_4]
.text:6440B0C5                  mov     [esi+2844h], eax
.text:6440B0CB                  lea     eax, [esi+28h] ; ESI is pointer to CDwsGui object
.text:6440B0CE                  push    eax
.text:6440B0CF                  call    CDwsGui__CopyOptions
```

综合上述分析可知：由 load_command_line()函数填充的数组，是 CDwsGui class（类）的一部分。这个数组的地址是 this+0x28。0x15+0x28＝0x3D。这应该就是数据被传递的终点站。

接下来，我们在程序里查找“使用偏移量 0x3D 的指令”。我们发现 CDwsGui::SapguiRun 函数（根据调试信息进行命名）就使用了这个偏移量：

```
.text:64409D58                  cmp     [esi+3Dh], bl     ; ESI is pointer to CDwsGui object
.text:64409D5B                  lea     ecx, [esi+2B8h]
.text:64409D61                  setz    al
.text:64409D64                  push    eax               ; arg_10 of CConnectionContext:: ↙
    ↳ CreateNetwork
.text:64409D65                  push    dword ptr [esi+64h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D68                  call    ds:mfc90_910
.text:64409D68                                            ; no arguments
.text:64409D6E                  push    eax
.text:64409D6F                  lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D75                  call        ds:mfc90_910
.text:64409D75                                            ; no arguments
.text:64409D7B                  push    eax
.text:64409D7C                  push    esi
.text:64409D7D                  lea     ecx, [esi+8]
.text:64409D80                  call    CConnectionContext__CreateNetwork
```

而后，验证我们的推测：把“setz al”换为“xor eax, eax / nop”指令，清除环境变量 TDW_NOCOMPRESS，然后再次运行 SAPGUI。此后，令人不快的提示窗口果然不见了，而且 Wireshark 显示网络包不再压缩了！显然，这种修改可以对 CConnectionContext 对象的压缩标识进行直接操作。

可见，压缩标识传递到了 CConnectionContext::CreateNetwork 的第五个参数。不过这个构造函数还调用了其他函数：

```
...
.text:64403476                  push    [ebp+compression]
.text:64403479                  push    [ebp+arg_C]
.text:6440347C                  push    [ebp+arg_8]
.text:6440347F                  push    [ebp+arg_4]
.text:64403482                  push    [ebp+arg_0]
.text:64403485                  call    CNetwork__CNetwork
```

压缩标识接着被传递到构造函数 CNetwork::CNetwork 的第五个参数。根据这个参数，构造函数 CNetwork 在对象体 CNetwork 设置标识、并设置另一个与压缩传输可能有关的变量。这个构造函数的有关操作如下：

```
.text:64411DF1                  cmp     [ebp+compression], esi
.text:64411DF7                  jz      short set_EAX_to_0
.text:64411DF9                  mov     al, [ebx+78h]    ; another value may affect compression?
.text:64411DFC                  cmp     al, '3'
.text:64411DFE                  jz      short set_EAX_to_1
.text:64411E00                  cmp     al, '4'
.text:64411E02                  jnz     short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
```

```
.text:64411E04                  xor     eax, eax
.text:64411E06                  inc     eax             ; EAX -> 1
.text:64411E07                  jmp     short loc_64411E0B
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09
.text:64411E09                  xor     eax, eax        ; EAX -> 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B                  mov     [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

综上，我们可以确定 CNetwork class 存储压缩标识的相对地址是 this+0x3A4。

然后我们以 0x3A4 为着手点，继续分析 SAPguilib.dll。这个偏移量再次出现在 CDwsGui::OnClientMessageWrite 里。当然，笔者还是通过调试信息才能确定构造函数的准确名称：

```
.text:64406F76 loc_64406F76:
.text:64406F76                  mov     ecx, [ebp+7728h+var_7794]
.text:64406F79                  cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                  jnz     compression_flag_is_zero
.text:64406F86                  mov     byte ptr [ebx+7], 1
.text:64406F8A                  mov     eax, [esi+18h]
.text:64406F8D                  mov     ecx, eax
.text:64406F8F                  test    eax, eax
.text:64406F91                  ja      short loc_64406FFF
.text:64406F93                  mov     ecx, [esi+14h]
.text:64406F96                  mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99                  push    dword ptr [edi+2868h] ; int
.text:64406F9F                  lea     edx, [ebp+7728h+var_77A4]
.text:64406FA2                  push    edx             ; int
.text:64406FA3                  push    30000           ; int
.text:64406FA8                  lea     edx, [ebp+7728h+Dst]
.text:64406FAB                  push    edx             ; Dst
.text:64406FAC                  push    ecx             ; int
.text:64406FAD                  push    eax             ; Src
.text:64406FAE                  push    dword ptr [edi+28C0h] ; int
.text:64406FB4                  call    sub_644055C5        ; actual compression routine
.text:64406FB9                  add     esp, 1Ch
.text:64406FBC                  cmp     eax, 0FFFFFFF6h
.text:64406FBF                  jz      short loc_64407004
.text:64406FC1                  cmp     eax, 1
.text:64406FC4                  jz      loc_6440708C
.text:64406FCA                  cmp     eax, 2
.text:64406FCD                  jz      short loc_64407004
.text:64406FCF                  push    eax
.text:64406FD0                  push    offset aCompressionErr ; "compression error [rc = \%d]- program wi" ...
.text:64406FD5                  push    offset aGui_err_compre ; "GUI_ERR_COMPRESS"
.text:64406FDA                  push    dword ptr [edi+28D0h]
.text:64406FE0                  call    SapPcTxtRead
```

由此可见，压缩网络数据的关键函数是 sub_644055C5。这个函数分别调用了 memcpy()函数，以及函数 sub_64417440（IDA 显示的函数名）。而 sub_64417440 的指令是：

```
.text:6441747C                  push    offset aErrorCsrcompre ; "\nERROR: CsRCompress: invalid handle"
.text:64417481                  call    eax ; dword_644F94C8
.text:64417483                  add     esp, 4
```

到此为止，我们完整地分析了压缩网络数据包的函数。参照笔者先前的分析[①]可知，这个网络包压缩函数是 SAP 和开源项目 MaxDB 的公用函数（上述两个产品都是 SAP 开发的）。因此，实际上我们可以找到它的源代码。

① http://conus.info/utils/SAP_pkt_decompr.txt。

最后要分析的是：

```
.text:64406F79                  cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                  jnz     compression_flag_is_zero
```

把此处的 JNZ 替换为无条件转移指令 JMP，删除环境变量 TDW_NOCOMPRESS。瞧！再用 WireShark 分析网络数据包时，我们就会发现 SAPGUI 不再压缩网络数据了。

综上，在找到环境变量与数据压缩功能的切合点之后，我们可以强行启用这个功能，也可以强制程序关闭这项功能。

80.2　SAP 6.0 的密码验证函数

某天，在 VMware 平台上打开 SAP 6.0 IDES 时，笔者发现自己忘记 SAP 账户名了。几经周折找到了账户名之后，我尝试着用常用密码进行登录。结果可想而知，笔者最终遇到了提示信息“Password logon no longer possible-too many failed attempts”，再也无法登录。

好消息是 SAP 官方提供了完整的 disp+work.pdb 文件。这个 PDB 文件涵盖的信息还十分全面：函数名、结构体、类型、局部变量及参数名等等，简直是应有尽有。

为了便于挖掘信息，笔者使用 TYPEINFODUMP 程序[①]，把 PDB 文件转换为了人类可读的文本文件。转换后的文本文件含有函数名称、函数参数、局部变量等信息：

```
FUNCTION ThVmcSysEvent
  Address:          10143190 Size:       675 bytes   Index:     60483  TypeIndex:     60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288 Size:         4 bytes   Index:    60488  TypeIndex:   60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296 Size:         1 bytes   Index:    60490  TypeIndex:   60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304 Size:         8 bytes   Index:    60492  TypeIndex:   60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:        12274af0 Size:         8 bytes  Index:     60495  TypeIndex:   60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304 Size:         8 bytes  Index:    60498  TypeIndex:     60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64 Size:      204 bytes  Index:    60501  TypeIndex:     60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296 Size:       4 bytes  Index:    60508  TypeIndex:     60509
  Type: int
Flags: 90
```

不仅如此，它还解释了结构体的有关信息：

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
```

① http://www.debuginfo.com/tools/typeinfodump.html。

```
MEMBER moduletype
  Type:  DBSL_MODULETYPE
  Offset:        0   Index:       3  TypeIndex:    38653
MEMBER module
  Type:  wchar_t module[40]
  Offset:        4   Index:       3  TypeIndex:      831
MEMBER stmtnum
  Type: long
  Offset:       84   Index:       3  TypeIndex:      440
MEMBER timestamp
  Type:  wchar_t timestamp[15]
  Offset:       88   Index:       3  TypeIndex:     6612
```

此外，调试呼叫（debugging calls）也可提供大量信息。

不久，笔者就注意到设置日志详细程度的全局变量 ct_level。SAP 官方对这个变量有详细的解释：http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e10000000a155369/content.htm。

disp+work.exe 文件保留了大量的调试信息：

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh        ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl     ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

如果 ctl_level 的值大于或等于程序预设的某个阈值，那么程序将会按照相应的详细程度记录 dev_wo、dev_disp 等 dev-日志文件。

在使用 TYPEINFODUMP 程序把 PDB 文件转换为文本文件之后，我们使用 grep 指令搜索与密码有关的函数名称：

```
cat    "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

上述指令的返回结果是：

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION 'rcui::AgiPassword::AgiPassword'::'1'::dtor$2
FUNCTION 'rcui::AgiPassword::AgiPassword'::'1'::dtor$0
FUNCTION 'rcui::AgiPassword::AgiPassword'::'1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION 'rcui::AgiPassword::`scalar deleting destructor''::'1'::dtor$1
```

```
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::'1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

根据提示信息，接下来我们在调试信息里搜索关键词“password”和“locked”。略加分析之后，笔者发现 password_attempt_limit_exceeded()函数会调用关键字符串“user was locked by subsequently failed password logon attempts”。

这个函数还会在日志文件里记录“password logon attempt will be rejected immediately (preventing dictionary attacks)”“failed-logon lock: expired (but not removed due to ‘read-only’ operation)”以及“failed-logon lock: expired => removed”。

进一步的研究表明，这个函数就是登录保护函数。它会被密码验证函数——chckpass()函数调用。

首先要验证上述推测是否正确。使用笔者开发的 tracer 程序进行分析：

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1 ", 0x41) (called from ↙
    ↳ 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

调用逻辑是 syssigni()→DyISigni()→dychkusr()→usrexist()→chckpass()。

数字 0x35 是 chckpass()函数返回的错误信息编号：

```
.text:00000001402ED567 loc_1402ED567:                          ; CODE XREF: chckpass+B4
.text:00000001402ED567                 mov     rcx, rbx        ; usr02
.text:00000001402ED56A                 call    password_idle_check
.text:00000001402ED56F                 cmp     eax, 33h
.text:00000001402ED572                 jz      loc_1402EDB4E
.text:00000001402ED578                 cmp     eax, 36h
.text:00000001402ED57B                 jz      loc_1402EDB3D
.text:00000001402ED581                 xor     edx, edx        ; usr02_readonly
.text:00000001402ED583                 mov     rcx, rbx        ; usr02
.text:00000001402ED586                 call    password_attempt_limit_exceeded
.text:00000001402ED58B                 test    al, al
.text:00000001402ED58D                 jz      short loc_1402ED5A0
.text:00000001402ED58F                 mov     eax, 35h
.text:00000001402ED594                 add     rsp, 60h
.text:00000001402ED598                 pop     r14
.text:00000001402ED59A                 pop     r12
.text:00000001402ED59C                 pop     rdi
.text:00000001402ED59D                 pop     rsi
.text:00000001402ED59E                 pop     rbx
.text:00000001402ED59F                 retn
```

然后进行试验：

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode, rt:0

PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) ↙
    ↳ (called from 0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
```

```
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (called ↙
    ↳ from 0x1402e9794 (disp+work.exe!chngpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

此后我们就可以进行登录了。

顺便提一下，如果忘记密码的话，可以把 chckpass()函数的返回值强制改为 0，那样它就不会进行密码验证了：

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0

PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus ", 0x41) (called from 0x1402f1060 ↙
    ↳ (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

在分析 password_attemp_limit_exceeded()函数时，我们可以看到函数的前几行指令是：

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E19D4
cmp     ax, 'n'
jz      short loc_1402E19D4
cmp     ax, '0'
jnz     short loc_1402E19DE
```

很显然，sapgparam()函数的作用是获取配置参数。整个程序有 1768 处指令调用这个函数。据此推测，只要追踪这个函数的调用关系，就可以分析特定参数对整个程序的影响。

不得不说，SAP 要比 Oracle RDBMS 亲切得多。前者提供的函数名等信息远比后者清晰。不过 disp+work 程序具有 C++程序的特征，莫非官方最近重新编写了它的源程序？

第 81 章　Oracle RDBMS

81.1　V$VERSION 表

Oracle RDBMS 11.2 是个规模庞大的数据库系统。其主程序 oracle.exe 包含近 124000 个函数。相比之下，Windows 7 x86 的内核 ntoskrnl.exe 只有近 11000 函数；Linux 3.9.8 的内核（默认编译/带有默认驱动程序）包含的函数也不过 31000 个左右。

本章首先演示一个最简单的 Oracle 查询指令。我们可通过下述指令查询 Oracle RDBMS 数据库的版本信息：

```
SQL> select * from V$VERSION;
```

上述指令的返回结果如下：

```
BANNER
--------------------------------------------------------------------------------

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

第一个问题就来了：字符串“V$VERSION”存储在 Oracle RDBMS 的什么地方？

在 Win32 版本的 oracle.exe 程序里不难发现这个字符串。但是在 Linux 平台的文件里，函数名称和全局变量名都会走样。因此，即使在 Linux 版的 Oracle RDBMS 里找到了正确的对象（.o）文件，挖掘相应的处理函数也会花费更多的时间。

在 Linux 版程序的文件里，包含字符串“V$VERSION”的文件是 kqf.o。这个文件在 Oracle 的库文件目录 lib/libserver11.a 之中。

kqf.o 文件在定义数据表 kqfviw 的时候，调用了字符串“V$VERSION”。

指令清单 81.1　kqf.o

```
.rodata:0800C4A0 kqfviw          dd  0Bh                 ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                                         ; kqfgbn+34
.rodata:0800C4A4                 dd  offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0800C4A8                 dd  4
.rodata:0800C4AC                 dd  offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0                 dd  3
.rodata:0800C4B4                 dd  0
.rodata:0800C4B8                 dd  195h
.rodata:0800C4BC                 dd  4
.rodata:0800C4C0                 dd  0
.rodata:0800C4C4                 dd  0FFFFC1CBh
.rodata:0800C4C8                 dd  3
.rodata:0800C4CC                 dd  0
.rodata:0800C4D0                 dd  0Ah
.rodata:0800C4D4                 dd  offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodata:0800C4D8                 dd  4
.rodata:0800C4DC                 dd  offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4E0                 dd  3
.rodata:0800C4E4                 dd  0
.rodata:0800C4E8                 dd  4Eh
.rodata:0800C4EC                 dd  3
```

```
.rodata:0800C4F0                dd 0
.rodata:0800C4F4                dd 0FFFFC003h
.rodata:0800C4F8                dd 4
.rodata:0800C4FC                dd 0
.rodata:0800C500                dd 5
.rodata:0800C504                dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508                dd 4
.rodata:0800C50C                dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510                dd 3
.rodata:0800C514                dd 0
.rodata:0800C518                dd 269h
.rodata:0800C51C                dd 15h
.rodata:0800C520                dd 0
.rodata:0800C524                dd 0FFFFC1EDh
.rodata:0800C528                dd 8
.rodata:0800C52C                dd 0
.rodata:0800C530                dd 4
.rodata:0800C534                dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538                dd 4
.rodata:0800C53C                dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540                dd 3
.rodata:0800C544                dd 0
.rodata:0800C548                dd 0F5h
.rodata:0800C54C                dd 14h
.rodata:0800C550                dd 0
.rodata:0800C554                dd 0FFFFC1EEh
.rodata:0800C558                dd 5
.rodata:0800C55C                dd 0
```

在分析 Oracle RDBMS 的内部文件时，很多人都会奇怪“为什么函数名称和全局变量名称都那么诡异？”这大概是因为 Oracle 是 20 世纪 80 年代的古典作品吧。那个时代 C 语言编译器都遵循的 ANSI 标准：函数名称和变量名称不得超出 6 个字符（linker 的局限），即“外部标识符以前 6 个字符为准”的规则。[①]

名字以 V$-开头的数据视图，多数（很有可能是全部）都由这个文件的 kqfviw 表定义。这些 V$视图都是内容固定视图（fixed Views）。从表面看来，这些数据具有显著的循环周期。因此，我们可以初步判断，kqfviw 表的每个元素都由 12 个 32 位字段构成。借助 IDA 程序，我们可以轻易地再现出这种 12 字段的数据结构，套用到整个数据表。在 Oracle RDBMS v11.2 里，总共有 1023 个固定视图。即，这个文件可能描述了 1023 个预定义的视图。本章稍后讨论这个数字。

关于视图中的各字段、及各字段对应的数据，并没有多少资料可寻。虽然我们发现第一个数字就是数据库图的名称（没有最末的那个零字节）、而且这个规律适用于全部的数据元素，但是这种信息的作用不大。

我们还查到了一个叫作“V$FIXED_VIEW_DEFINITION”的固定视图[②]，它能够检索所有固定视图的信息。顺便提一下，这个表有 1023 个元素，正好对应预定义视图的总数。

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

可见，对于 GV$VERSION 而言，V$VERSION 是 thunk view（形实转换视图）：

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';
```

① 1988 年的 ANSI 标准请可参见笔者的摘录：http://yurichev.com/ref/Draft%20ANSI%20C%20Standard%20(ANSI%20X3J11-88-090)%20(May%2013,%201988).txt。作为对比，微软的标识符标准可参阅 https://msdn.microsoft.com/en-us/library/e7f8y25b.aspx。

② 笔者通过挖掘 kqfviw 和 kqfvip 表里的数据，最终发现了这个视图的信息。

```
VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$VERSION
select inst_id, banner from x$version
```

另外，在 Oracle 数据库里，那些官方文档没有介绍的、以 X$开头的数据表同样是记载系统信息的服务表。因为这些以 X$开头的表由 Oracle 程序控制并动态更新的数据表，所以数据库用户不能修改它们。

如果我们在文件 kqf.o 里搜索文本“select BANNER from GV$VERSION where inst_id = USERENV('Instance')”，那么就会发现它在 kqfvip 表里。

指令清单 81.2　kqf.o

```
.rodata:080185A0 kqfvip          dd  offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0                                               ; kqfgvt+F
.rodata:080185A0                                               ; "select inst_id, decode(indx,1,'data ↙
    ↳ bloc" ...
.rodata:080185A4                 dd  offset kqfv459_c_0
.rodata:080185A8                 dd  0
.rodata:080185AC                 dd  0

...

.rodata:08019570                 dd  offset _2__STRING_11378_0 ; "select BANNER from GV$VERSION ↙
    ↳  where in "...
.rodata:08019574                 dd  offset kqfv133_c_0
.rodata:08019578                 dd  0
.rodata:0801957C                 dd  0
.rodata:08019580                 dd  offset _2__STRING_11379_0 ; "select inst_id,decode(bitand(↙
    ↳ cflg,1),0"...
.rodata:08019584                 dd  offset kqfv403_c_0
.rodata:08019588                 dd  0
.rodata:0801958C                 dd  0
.rodata:08019590                 dd  offset _2__STRING_11380_0 ; "select STATUS , NAME, ↙
    ↳ IS_RECOVERY_DEST"...
.rodata:08019594                 dd  offset kqfv199_c_0
```

这个表的每个元素由 4 个字段构成。而且它同样包含了 1023 个元素。第二个字段指向了另一个表——也就是与表名称相对应的固定视图。V$VERSION 的表格只有 2 个元素，第一个是 6（后面字符串的长度），第二个是 BANNER 字符串。此后是终止符——零字节和 C 语言字符 null。

指令清单 81.3　kqf.o

```
.rodata:080BBAC4 kqfv133_c_0     dd  6                       ; DATA XREF: .rodata:08019574
.rodata:080BBAC8                 dd  offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC                 dd  0
.rodata:080BBAD0                 dd  offset _2__STRING_0_0
```

因此可见，综合 kqfviw 和 kqfvip 表的各项信息，我们可以获悉某个固定视图都含有哪些可被查询的字段。

基于上述分析结果，笔者编写了一个专门导出 Linux Oracle 数据库系统表的小程序——oracle_tables[①]。用它导出 V$VERSION 时，可得到如下所示的各项信息。

指令清单 81.4　Result of oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where inst_id = USERENV('Instance')]
```

① http://yurichev.com/oracle_tables.html。

```
kqfvip_element.params:
[BANNER]
```

指令清单 81.5 Result of oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

固定视图 GV$VERSION 比 V$VERSION 多出了一个“instance”字段，除此以外两者相同。因此，我们只要专心研究数据表 X$VERSION 就可举一反三地理解另一个表。与其他名字以 X$-开头的数据表一样，这个表也没有资料可查。但是，我们可以直接对其进行检索：

```
SQL> select * from x$version;

ADDR           INDX    INST_ID
-------- ---------- ----------
BANNER
--------------------------------------------------------------------------------

0DBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production

...
```

这个表的字段名里有 ADDR 和 INDX。

继续使用 IDA 分析 kqf.o 的时候，我们会发现在 kqftab 表里有一个指向 X$VERSION 字符串的指针。

指令清单 81.6 kqf.o

```
.rodata:0803CAC0                dd 9                    ; element number 0x1f6
.rodata:0803CAC4                dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8                dd 4
.rodata:0803CACC                dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0                dd 4
.rodata:0803CAD4                dd 4
.rodata:0803CAD8                dd 0
.rodata:0803CADC                dd 4
.rodata:0803CAE0                dd 0Ch
.rodata:0803CAE4                dd 0FFFFC075h
.rodata:0803CAE8                dd 3
.rodata:0803CAEC                dd 0
.rodata:0803CAF0                dd 7
.rodata:0803CAF4                dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8                dd 5
.rodata:0803CAFC                dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00                dd 1
.rodata:0803CB04                dd 38h
.rodata:0803CB08                dd 0
.rodata:0803CB0C                dd 7
.rodata:0803CB10                dd 0
.rodata:0803CB14                dd 0FFFFC09Dh
.rodata:0803CB18                dd 2
.rodata:0803CB1C                dd 0
```

上述指令中有很多处数据都引用了以 X$-开头的数据表名称。很显然，这些名字都是 Oracle 数据库的数据表名称。鉴于公开资料没有这些信息，笔者还不能理解字符串“kqvt”的实际含义。“kq-”前缀的指令，多数是与 Kernel（内核）和 query（查询）有关的指令。不过，至于“v 是否是 version 的缩写”、“t 是否是 type 的缩写”，这些猜测都无法证明。

另外，kqf.o 文件里还记录了类似的数据表名称。

指令清单 81.7　kqf.o

```
.rodata:0808C360 kqvt_c_0        kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, ↙
    ↳ 0>
.rodata:0808C360                                        ; DATA XREF: .rodata:08042680
.rodata:0808C360                                        ; "ADDR"
.rodata:0808C384                 kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0, ↙
    ↳ 0>;"INDX"
.rodata:0808C3A8                 kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0, ↙
    ↳ 0>;"INST_ID"
.rodata:0808C3CC                 kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 50h,↙
    ↳ 0, 0> ; "BANNER"
.rodata:0808C3F0                 kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

这些信息可以解释 X$VERSION 表中的所有字段。在 kqftap 表中，唯一一个引用这个表的指令如下所示。

指令清单 81.8　kqf.o

```
.rodata:08042680          kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0> ; ↙
    ↳ element 0x1f6
```

值得关注的是，这个元素是表中第 502 个（0x1f6）元素。它就像 kqftab 表中指向 X$VERSION 字符串的指针一般。数据表 kqftap 和 kqftab 之间的关系，很可能像 kqfvip 和 kqfviw 之间的关系那样是某种互补关系。我们还在其中找到了指向 kqvrow() 函数的函数指针。我们最终挖掘到了有价值的信息！

笔者把上述各表的有关信息也添加到了自制的 oracle 系统表查询工具——oracle_tables 里[①]。用它检索 X$VERSION 后，可得如下所示的各项信息。

指令清单 81.9　Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

借助笔者自创的 tracer 程序，我们不难发现：在查询 X$VERSION 表时，这个函数被连续调用了 6 次（由 qerfxFetch() 函数）。

为了查看具体执行了哪些指令，我们以 cc 模式运行 tracer 程序：

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc


_kqvrow_  proc near

var_7C    = byte ptr -7Ch
var_18    = dword ptr -18h
var_14    = dword ptr -14h
Dest      = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4
arg_8     = dword ptr  10h
arg_C     = dword ptr  14h
arg_14    = dword ptr  1Ch
arg_18    = dword ptr  20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES
```

① http://yurichev.com/oracle_tables.html。

```
        push    ebp
        mov     ebp, esp
        sub     esp, 7Ch
        mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
        mov     ecx, TlsIndex   ; [69AEB08h]=0
        mov     edx, large fs:2Ch
        mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
        cmp     eax, 2          ; EAX=1
        mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
        jz      loc_2CE1288
        mov     ecx, [eax]      ; [EAX]=0..5
        mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6:  ; CODE XREF: _kqvrow_+10A
            ; _kqvrow_+1A9
        cmp     ecx, 5          ; ECX=0..5
        ja      loc_56C11C7
        mov     edi, [ebp+arg_18] ; [EBP+20h]=0
        mov     [ebp+var_14], edx ; EDX=0xc98c938
        mov     [ebp+var_8], ebx ; EBX=0
        mov     ebx, eax        ; EAX=0xcdfe554
        mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
        mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]= 0x2ce1116, 0x2ce11ac, 0x2ce11db↙
    ↳ , 0x2ce11f6, 0x2ce1236, 0x2ce127a
        jmp     edx             ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, ↙
    ↳ 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
        push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
        mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
        xor     edx, edx
        mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
        push    edx             ; EDX=0
        push    edx             ; EDX=0
        push    50h
        push    ecx             ; ECX=0x8a172b4
        push    dword ptr [esi+10494h] ;[ESI+10494h]=0xc98cd58
        call    _kghalf         ; tracing nested maximum level (1) reached, skipping this ↙
   ↳ CALL
        mov     esi, ds:__imp__vsnnum ; [59771A8h]=0x61bc49e0
        mov     [ebp+Dest], eax ; EAX=0xce2ffb0
        mov     [ebx+8], eax    ; EAX=0xce2ffb0
        mov     [ebx+4], eax    ; EAX=0xce2ffb0
        mov     edi, [esi]      ; [ESI]=0xb200100
        mov     esi, ds:__imp__vsnstr ; [597D6D4h]=0x65852148, "- Production"
        push    esi             ; ESI=0x65852148, "- Production"
        mov     ebx, edi        ; EDI=0xb200100
        shr     ebx, 18h        ; EBX=0xb200100
        mov     ecx, edi        ; EDI=0xb200100
        shr     ecx, 14h        ; ECX=0xb200100
        and     ecx, 0Fh        ; ECX=0xb2
        mov     edx, edi        ; EDI=0xb200100
        shr     edx, 0Ch        ; EDX=0xb200100
        movzx   edx, dl         ; DL=0
        mov     eax, edi        ; EDI=0xb200100
        shr     eax, 8          ; EAX=0xb200100
        and     eax, 0Fh        ; EAX=0xb2001
        and     edi, 0FFh       ; EDI=0xb200100
        push    edi             ; EDI=0
        mov     edi, [ebp+arg_18] ; [EBP+20h]=0
        push    eax             ; EAX=1
        mov     eax, ds:__imp__vsnban ; [597D6D8h]=0x65852100, "Oracle Database 11g ↙
```

```
    ↳ Enterprise Edition Release %d.%d.%d.%d.%d %s"
        push    edx             ; EDX=0
        push    ecx             ; ECX=2
        push    ebx             ; EBX=0xb
        mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
        push    eax             ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release %d.%d.%d.%d.%d %s"
        mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
        push    eax             ; EAX=0xce2ffb0
        call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested maximum level (1) ↙
    ↳ reached, skipping this CALL
        add     esp, 38h
        mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
           ; _kqvrow_+128 ...
        test    edi, edi        ; EDI=0
        jnz     __VInfreq__kqvrow
        mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
        mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
        mov     eax, ebx        ; EBX=0xcdfe554
        mov     ebx, [ebp+var_8] ; [EBP-8]=0
        lea     eax, [eax+4]    ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production↙
    ↳ ", "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production", "PL/SQL ↙
    ↳ Release 11.2.0.1.0 - Production", "TNS for 32-bit Windows: Version 11.2.0.1.0 - ↙
    ↳ Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
        mov     esp, ebp
        pop     ebp
        retn                    ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0- Production"
        mov     dword ptr [ebx], 2
        mov     [ebx+4], edx    ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0 - Production"
        push    edx             ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0 - Production"
        call    _kkxvsn         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        pop     ecx
        mov     edx, [ebx+4]    ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        movzx   ecx, byte ptr [edx] ; [EDX]=0x50
        test    ecx, ecx        ; ECX=0x50
        jnz     short loc_2CE1192
        mov     edx, [ebp+var_14]
        mov     esi, [ebp+var_C]
        mov     eax, ebx
        mov     ebx, [ebp+var_8]
        mov     ecx, [eax]
        jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
        push    0
        push    50h
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        mov     [ebx+4], edx    ; EDX=0xce2ffb0,"PL/SQL Release 11.2.0.1.0 - Production"
        push    edx             ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        call    _lmxver         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        add     esp, 0Ch
        mov     dword ptr [ebx], 3
        jmp     short loc_2CE1192
```

```
loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
        mov     [ebp+var_18], 50h
        mov     [ebx+4], edx    ; EDX=0xce2ffb0
        push    0
        call    _npinli         ; tracing nested maximum level (1)reached, skipping this ↙
    ↳ CALL
        pop     ecx
        test    eax, eax        ; EAX=0
        jnz     loc_56C11DA
        mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
        lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
        push    edx             ; EDX=0xd76c93c
        push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
        push    dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
        call    _nrtnsvrs       ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
        mov     dword ptr [ebx], 4
        jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
        lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
        push    edx             ; EDX=0xd76c8d8
        push    0
        mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version ↙
    ↳ 11.2.0.1.0 - Production"
        mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↙
    ↳ - Production"
        mov     ecx, 50h
        mov     [ebp+var_18], ecx ; ECX=0x50
        push    ecx             ; ECX=0x50
        push    esi             ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↙
    ↳ - Production"
        call    _lxvers         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        add     esp, 10h
        mov     edx, [ebp+var_18 ; [EBP-18h]=0x50
        mov     dword ptr [ebx], 5
        test    edx, edx        ; EDX=0x50
        jnz     loc_2CE1192
        mov     edx, [ebp+var_14]
        mov     esi, [ebp+var_C]
        mov     eax, ebx
        mov     ebx, [ebp+var_8]
        mov     ecx, 5
        jmp     loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
        mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
        mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
        mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
        mov     eax, ebx        ; EBX=0xcdfe554
        mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
        mov     eax, [eax+8]    ; [EAX+8]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
        test    eax, eax        ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
        jz      short loc_2CE12A7
        push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
        push    eax             ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
        mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
```

```
        push    eax             ; EAX=0x8a172b4
        push    dword ptr [edx+10494h]  ; [EDX+10494h]=0xc98cd58
        call    _kghfrf         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        add     esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        retn                    ; EAX=0
_kqvrow_  endp
```

不难看出，该函数从外部获取行号信息，然后按照下述顺序组装、返回字符串。

String 1	Using vsnstr, vsnnum, vsnban global variables. Calling sprintf().
String 2	Calling kkxvsn().
String 3	Calling lmxver().
String 4	Calling npinli(), nrtnsvrs().
String 5	Calling lxvers().

Oracle 按照上述次序依次调用相应函数，从而获取各个模块的版本信息。

81.2 X$KSMLRU 表

官方文件《Diagnosing and Resolving Error ORA-04031》[①]特别提到了这个数据表：

> Oracle 能够记录内存池内发生的、强制释放其他对象的内存占用情况。负责记录这种情况的数据表是固定表 x$ksmlru。它可用来诊断内存异常消耗的具体原因。
>
> 如果内存池里发生了大量对象周期性释放的情况，那么这种问题会增加数据库的响应时间。而且当这些对象再次被加载到内存池时，这一现象还会增加库缓存（library cache）互锁的概率。
>
> 固定表 x$ksmlru 具有一个特性：只要出现了检索表的人为操作，那么这个表内的数据就会被立刻清空。此外，该数据表只会存储内存占用最大的前几项记录。“查询后立刻清空”的设定，是为了凸显那些先前并不那么耗费资源的内存分配情况。也就是说，每次检索所对应的时间段都是不同的。正因如此，数据库用户应当妥善保管该表的查询结果。

换句话说，查询这个表不是问题，问题是查询后它会被立即清空。那么，清空表的具体原因是什么？既然 kqftab 表和 kqftap 表含有 X$-表的全部信息，我们可以继续使用前文介绍的 oracle_tables 进行分析。在 oracle_tables 的返回结果里，我们看到：在制备 X$KSMLRU 表的元素时，oracle 调用了 ksmlrs() 函数。

指令清单 81.10 Result of oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
```

① http://www.oralab.net/METANOTES/DIAGNOSING%20AND%20RESOLVING%20ORA-04031%20ERROR.htm。

```
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

tracer 程序可以印证这个结果：每次查询 X$KSMLRU 表时，Oracle 都会调用这个函数。

另外，我们还看到 ksmsplu_sp() 函数和 ksmsplu_jp() 函数都引用了 ksmsplu() 函数。即，无论是执行 ksmsplu_sp() 函数、还是执行 ksmsplu_jp() 函数，最后都会调用 ksmsplu() 函数。在 ksmsplu() 结束之前，它调用了 memset() 函数。

指令清单 81.11 ksm.o

```
...
.text:00434C50 loc_434C50:                             ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50                 mov     edx, [ebp-4]
.text:00434C53                 mov     [eax], esi
.text:00434C55                 mov     esi, [edi]
.text:00434C57                 mov     [eax+4], esi
.text:00434C5A                 mov     [edi], eax
.text:00434C5C                 add     edx, 1
.text:00434C5F                 mov     [ebp-4], edx
.text:00434C62                 jnz     loc_434B7D
.text:00434C68                 mov     ecx, [ebp+14h]
.text:00434C6B                 mov     ebx, [ebp-10h]
.text:00434C6E                 mov     esi, [ebp-0Ch]
.text:00434C71                 mov     edi, [ebp-8]
.text:00434C74                 lea     eax, [ecx+8Ch]
.text:00434C7A                 push    370h            ; Size
.text:00434C7F                 push    0               ; Val
.text:00434C81                 push    eax             ; Dst
.text:00434C82                 call    __intel_fast_memset
.text:00434C87                 add     esp, 0Ch
.text:00434C8A                 mov     esp, ebp
.text:00434C8C                 pop     ebp
.text:00434C8D                 retn
.text:00434C8D _ksmsplu        endp
```

含有 memset（block，0，size）的构造函数通常用于清空内存区域。如果我们阻止它调用这个 memset() 函数，那么将发生什么情况？

为此，我们在程序向 memset() 函数传递参数的 0x434C7A 处设置断点、令调试程序 tracer 在此刻将程序计数器（PC，即 EIP）调整为 0x434C8A，从而使程序“跳过”清除内存的 memset() 函数。可以说，这种“调试”相当于令程序在 0x434C7A 处无条件转移到 0x434C8A。相关的 tracer 指令如下：

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

请注意：上述地址仅对 Win32 版本的 Oracle RDBMS 11.2 有效。

经上述调试指令启动 Oracle 以后，无论查询 X$ KSMLRU 表多少次，这个表都不会被清空了。当然，不要在投入实用的业务服务器上进行这种测试。

或许这种调试的用处不大，或许这种修改有悖实用性原则。不过，当我们要查找特定的指令时，我们可以采用这样的调试步骤！

81.3 V$TIMER 表

固定视图 V$TIMER 算得上是更新最频繁的视图之一了。

V$TIME 以百分之一秒为单位、记录实际运行时间。这个值以计时原点开始测算，因此具体数值与操作系统相关。它会在 4 字节溢出时（大约历经 497 天后）循环，重新变为 0。

上述内容摘自官方文档。[①]

比较有趣的是：Win32 版本的 Oracle 程序和 Linux 版本的程序，返回的时间戳竟然是不同的。我们能否找到生成返回值的函数呢？

下述操作表明，时间信息最终取自 X$KSUTM 表：

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

不过 kqftab/kqftap 表没有引用生成这项数值的函数。

指令清单 81.12　Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

当我们搜索字符串 KSUTMTIM 时，我们看到了下述函数：

```
kqfd_DRN_ksutm_c proc near              ; DATA XREF: .rodata:0805B4E8

arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    [ebp+arg_C]
                push    offset ksugtm
                push    offset _2__STRING_1263_0 ; "KSUTMTIM"
                push    [ebp+arg_8]
                push    [ebp+arg_0]
                call    kqfd_cfui_drain
                add     esp, 14h
                mov     esp, ebp
                pop     ebp
                retn
```

① http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm。

```
kqfd_DRN_ksutm_c endp
```

而数据表 kqfd_tab_registry_0 引用了 kqfd_DRN_ksutm_c() 函数：

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

打开 Linux x86 版本的这个文件，可看到如下所示的代码。

指令清单 81.13 ksu.o

```
ksugtm          proc near

var_1C          = byte ptr -1Ch
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 1Ch
                lea     eax, [ebp+var_1C]
                push    eax
                call    slgcs
                pop     ecx
                mov     edx, [ebp+arg_4]
                mov     [edx], eax
                mov     eax, 4
                mov     esp, ebp
                pop     ebp
                retn
ksugtm          endp
```

在 Win32 版本的程序里，相应文件的有关指令几乎相同。

这是我们寻找的函数吗？我们通过下述指令验证一下：

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

然后在 SQL*Plus 里执行以下指令：

```
SQL> select * from V$TIMER;

     HSECS
----------
  27294929

SQL> select * from V$TIMER;

     HSECS
----------
  27295006

SQL> select * from V$TIMER;

     HSECS
----------
  27295167
```

指令清单 81.14 tracer output

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                          "8.             "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
```

```
Argument 2/2 difference
00000000: D1 7C A0 01                                        ".|..            "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                              "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                              "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                        ".}..            "
```

上述数据和我们在 SQL*Plus 看到的数据完全一样。它是函数的第二个参数。

然后我们再来分析 Linux x86 程序里的 slgcs() 函数：

```
slgcs           proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    esi
                mov     [ebp+var_4], ebx
                mov     eax, [ebp+arg_0]
                call    $+5
                pop     ebx
                nop                     ; PIC mode
                mov     ebx, offset _GLOBAL_OFFSET_TABLE_
                mov     dword ptr [eax], 0
                call    sltrgatime64    ; PIC mode
                push    0
                push    0Ah
                push    edx
                push    eax
                call    __udivdi3       ; PIC mode
                mov     ebx, [ebp+var_4]
                add     esp, 10h
                mov     esp, ebp
                pop     ebp
                retn
slgcs           endp
```

这个函数调用了 sltrgatime64()，然后把返回值除以 10。[①]

在 Win32 版本的程序里，这个函数则是：

```
_slgcs          proc near               ; CODE XREF: _dbgefgHtElResetCount+15
                                        ; _dbgerRunActions+1528

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+8]
                mov     dword ptr [eax], 0
                call    ds:__imp__GetTickCount@0 ; GetTickCount()
                mov     edx, eax
                mov     eax, 0CCCCCCCDh
```

① 有关除法运算的有关细节，请参见本书第 41 章。

```
                mul     edx
                shr     edx, 3
                mov     eax, edx
                mov     esp, ebp
                pop     ebp
                retn
_slgcs          endp
```

Win32 的结果就是 GetTickCount() 函数返回值的十分之一。[①]

这就是 Oracle 在 Win32 下和 Linux x86 下返回不同结果的根本原因——它调用了完全不同的操作系统函数。

“call kqfd_cfui_drain”里有个“drain”。这个关键字有“表中的某个列取自特定函数的返回值”的含义。

前面介绍过的 oracle_tables 工具能够处理 kqfd_tab_registry_0。因此，我们可以用它分析“列”的值与特定函数之间的关联关系：

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

上述信息中的 OPN 代表“Open”和“DRN”。DRN 当然还是“drain”的意思。

① 有关 GetTickCount() 函数，请参见 MSDN：https://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx。

第 82 章　汇编指令与屏显字符

82.1　EICAR

多数反病毒软件都用 EICAR 进行自检。EICAR 是一个可以在 MS-DOS 平台上运行的应用程序。它仅在屏幕上显示“EICAR-STANDARD-ANTIVIRUS-TEST-FILE!”这样一个字符串。[①]

EICAR 最重要的特点是：它的每个字节都是可以在屏幕上显示出来的 ASCII 字符串。我们在文本编译器里粘贴下列字符串，即可生成 EICAR 文件：

```
X5O!P%@AP[4\PZX54(P^)7CC]7]$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

EICAR 文件的汇编指令如下：

```
; initial conditions: SP=0FFFEh, SS:[SP]=0
0100 58            pop     ax
; AX=0, SP=0
0101 35 4F 21      xor     ax, 214Fh
; AX = 214Fh and SP = 0
0104 50            push    ax
; AX = 214Fh, SP = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41      and     ax, 4140h
; AX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
0108 50            push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFE] = 214Fh
0109 5B            pop     bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C         xor     al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010C 50            push    ax
010D 5A            pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFFE] = 214Fh
010E 58            pop     ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28      xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50            push    ax
0113 5E            pop     si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37         sub     [bx], si
0116 43            inc     bx
0117 43            inc     bx
0118 29 37         sub     [bx], si
011A 7D 24         jge     short near ptr word_10140
011C 45 49 43 ...  db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B word_10140    dw 2B48h  ; CD 21 (INT 21) will be here
0142 48 2A               dw 2A48h  ; CD 20 (INT 20) will be here
0144 0D                  db  0Dh
0145 0A                  db  0Ah
```

笔者在上述代码里追加了各种注释，以介绍执行指令后各寄存器和栈的状态。

本质上说，这个程序的关键指令（下文简称“核心指令”）只有：

```
B4 09    MOV AH, 9
BA 1C 01 MOV DX, 11Ch
```

① 请参见 https://en.wikipedia.org/wiki/EICAR_test_file。

```
CD 21    INT 21h
CD 20    INT 20h
```

INT 21 的第 9 号功能（AH 寄存器）的作用是显示字符串。系统中断从 DS:DX 获取字符串的指针，然后把它输出在屏幕上。另外，这种字符串必须以“$”字符结尾。当今的操作系统显然继续兼容了这种指令。不过这种指令属于 CP/M（Control Program for Microcomputers），来自比 MS-DOS 还要古老的磁盘操作系统。

由此可见，EICAR 的主要功能是：

- 向寄存器（AH 和 DX）传递预定值。
- 在内存中准备 INT 21 和 INT 20 的 opcode。
- 执行 INT 21 和 INT 20。

严格地讲，核心指令的 opcode 基本都不是屏显字符。EICAR 采用“拼凑”指令的方法，把核心指令凑成了可存储在字符串里的屏显字符组合。这项“拼凑”技术还普遍应用于 shellcode。

有关“可用屏显字符表示的 opcode”，请参见本书附录 A.6.5。

第 83 章　实 例 演 示

编写 Demo 程序不仅要求编程人员具备熟练的数学技巧、卓越的计算机绘图水平，而且还非常考验他们手写 x86 代码的基本功。

83.1　10PRINT CHR$(205.5+RND(1));:GOTO 10

本节介绍的程序都是 MS-DOS 环境下的.COM 程序。

在参考资料［a12］里，我们找到了一个非常简单的随机图案生成程序。虽然它的功能只是不停地在屏幕上打印斜杠和反斜杠，但是这两种字符最终可构成一种几何图案，如图 83.1 所示。

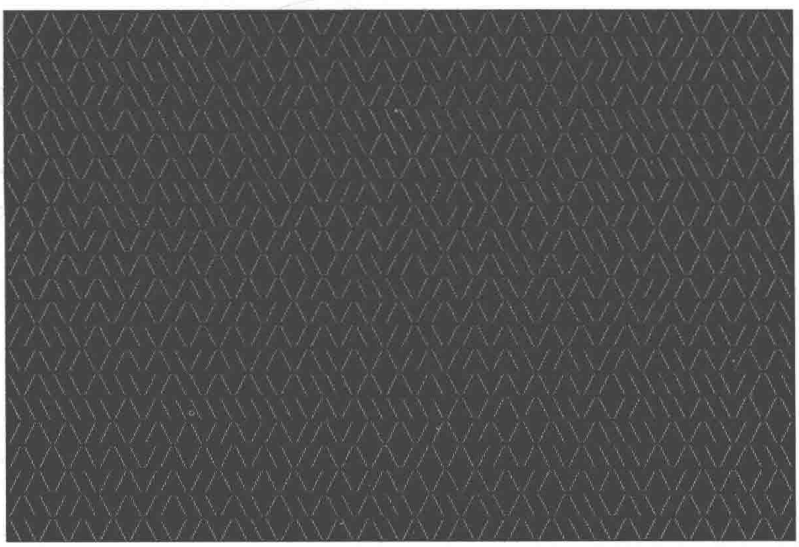


图 83.1　一种随机几何图案

在 16 位的 x86 平台上，这种算法非常多。

83.1.1　Trixter 的 42 字节程序

Trixter 在他的网站上[①]公开了具备 42 字节大小的程序。笔者把它摘录出来，并标注上了自己的注释：

```
00000000: B001          mov         al,1        ; set 40x25 video mode
00000002: CD10          int         010
00000004: 30FF          xor         bh,bh       ; set video page for int 10h call
00000006: B9D007        mov         cx,007D0    ; 2000 characters to output
00000009: 31C0          xor         ax,ax
0000000B: 9C            pushf                   ; push flags
; get random value from timer chip
0000000C: FA            cli                     ; disable interrupts
0000000D: E643          out         043,al      ; write 0 to port 43h
; read 16-bit value from port 40h
0000000F: E440          in          al,040
00000011: 88C4          mov         ah,al
```

① http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/。

```
00000013: E440          in          al,040
00000015: 9D            popf                    ; enable interrupts by restoring IF flag
00000016: 86C4          xchg        ah,al
; here we have 16-bit pseudorandom value
00000018: D1E8          shr         ax,1
0000001A: D1E8          shr         ax,1
; CF currently have second bit from the value
0000001C: B05C          mov         al,05C ;'\'
; if CF=1, skip the next instruction
0000001E: 7202          jc          000000022
; if CF=0, reload AL register with another character
00000020: B02F          mov         al,02F ;'/'
; output character
00000022: B40E          mov         ah,00E
00000024: CD10          int         010
00000026: E2E1          loop        000000009 ; loop 2000 times
00000028: CD20          int         020       ; exit to DOS
```

实际上，上述程序里的伪随机数是 Intel 8253 计时器芯片（硬件）回传过来的时间信息。它选用了零号计时器，而时钟决定这个计时器每秒递增 18.2 次。

向 0x43 端口发送零字节，相当于发送了“选定#0 号（通用）计数器”“计数器的输出持续可读”和“采用二进制计数（返回值是二进制数字，而非 BCD 码）”这三条指令。[①]

当程序执行 POPF 指令时，CPU 恢复 IF 标识的同时会恢复终端功能。

在使用 IN 指令读取数据时，返回值必须写到 AL 寄存器里，所以后面出现了数据交换指令 xchg。

83.1.2 笔者对 Trixter 算法的改进：27 字节

这个程序并没有使用计时器查询确切时间，而是用它来生成伪随机数。因此，我们没有必要屏蔽系统中断。此外，我们只需要返回值的低 8 位数据，所以读这低 8 位数据即可。

笔者对 Trixter 的程序稍作精简，把它改进为 27 字节的程序：

```
00000000: B9D007    mov     cx,007D0 ; limit output to 2000 characters
00000003: 31C0      xor     ax,ax    ; command to timer chip
00000005: E643      out     043,al
00000007: E440      in      al,040   ; read 8-bit of timer
00000009: D1E8      shr     ax,1     ; get second bit to CF flag
0000000B: D1E8      shr     ax,1
0000000D: B05C      mov     al,05C   ; prepare '\'
0000000F: 7202      jc      000000013
00000011: B02F      mov     al,02F   ; prepare '/'
; output character to screen
00000013: B40E      mov     ah,00E
00000015: CD10      int     010
00000017: E2EA      loop    000000003
; exit to DOS
00000019: CD20      int     020
```

83.1.3 从随机地址读取随机数

MS-DOS 系统完全没有内存保护技术的概念。换句话说，应用程序可以任意访问内存地址。不仅如此，在使用 LODSB 指令从 DS:SI 读取单字节数据的时候，即使程序没有预先给寄存器赋值也没有问题——它会从任意地址读取一个字节！

Trixter 甚至在其网页里[②]推荐不初始化相关寄存器就直接使用 LODSB 指令。

原文同时建议使用 SCASB 指令替代 LODSB 指令，因为前者可以在读取数据的同时，根据数据直接设置标识位。

---

① 实际上，8 位控制字相当于四条指令。因为另一个指令没有在本程序发挥作用，所以本文没有进行介绍。

② http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/。

此外，使用 DOS 系统的 syscall INT 29h 还能对程序进行进一步精简。这个中断可以在屏幕上输出 AL 寄存器里的字符。

Peter Ferrie 和 Andrey “hermlt” Baranovich 分别写出了 11 字节和 10 字节的程序。[①]

指令清单 83.1　Andrey “herm1t” Baranovich: 11 bytes

```
00000000: B05C       mov         al,05C    ;'\'
; read AL byte from random place of memory
00000002: AE         scasb
; PF = parity(AL - random_memory_byte) = parity(5Ch - random_memory_byte)
00000003: 7A02       jp          000000007
00000005: B02F       mov         al,02F    ;'/'
00000007: CD29       int         029       ; output AL to screen
00000009: EBF5       jmp         000000000 ; loop endlessly
```

SCASB 指令计算 “AL-[随机地址]”，并设置相应标识位。JP 指令比较少见，触发 JP 转移的条件是奇偶标识位 PF 为 1（偶数）。在这个程序里，输出的字符不再由随机字节的某个比特位决定，而是由这个字节的各个比特位共同决定。因此，这个程序的散列程度有望更好一些。

如果使用 x86 未公开的 SALC 指令（即 SETALC）、单指令完成 “SET AL CF”，那么整个程序还可以更短。这个指令最初出现在 NEC v20 CPU 上。用自然语言解释的话，它的功能就是 “有 CF 标识位填充 AL 寄存器”若 CF 为 1，则 AL 的值将会是 0xFF；否则 AL 的值就是零。受到 SALC 适用性的影响，任 8086/8088 平台上这个程序应该跑不起来。

指令清单 83.2　Peter Ferrie: 10 bytes

```
; AL is random at this point
00000000: AE         scasb
; CF is set according subtracting random memory byte from AL.
; so it is somewhat random at this point
00000001: D6         setalc
; AL is set to 0xFF if CF=1 or to 0 if otherwise
00000002: 242D       and         al,02D ;'-'
; AL here is 0x2D or 0
00000004: 042F       add         al,02F ;'/'
; AL here is 0x5C or 0x2F
00000006: CD29       int         029       ; output AL to screen
00000008: EBF6       jmps        000000000 ; loop endlessly
```

因此，完全有可能彻底抛弃条件转移指令。反斜杠（“\”）和斜杠（“/”）的 ASCII 值分别是 0x5C 和 0x2F。余下的问题就是：可否根据 CF 的（伪随机）状态把 AL 寄存器的值设置为 0x5C 和 0x2F。

实际上解决方法十分简单：无论 AL 的值是 0 还是 0xFF，我们把它和 0x2D 进行 “与” 运算，即可得到 0 和 0x2D。再把这个值与 0x2F 相加，就得到了 0x2F 和 0x5C。然后把 AL 寄存器里的值打印到屏幕上即可。

83.1.4　其他

应当注意的是：在 DOSBox、Windows NT 主机、甚至是 MS-DOS 主机上运行同一个程序，看到的图案都可能是不同的。影响程序结果的因素有：模拟器对 Intel 8253 计时器的不同模拟方式、寄存器的不同初始值，以及其他因素。

83.2　曼德博集合

多少年来，编程人员不懈地钻研曼德博集合[②]的各种算法。本文将要介绍的，是 Sir_Lagsalot 在 2009 年发

① 请参照 http://pferrie.host22.com/misc/10print.htm。
② 还被译作曼德布洛特集合，英文原文是 Mandelbrot set。

表的曼德博集合的 Demo 程序[①]。这个程序由 30 个 16 位 x86 指令构成，文件大小仅为 64 字节。

它绘制的图案如图 83.2 所示。

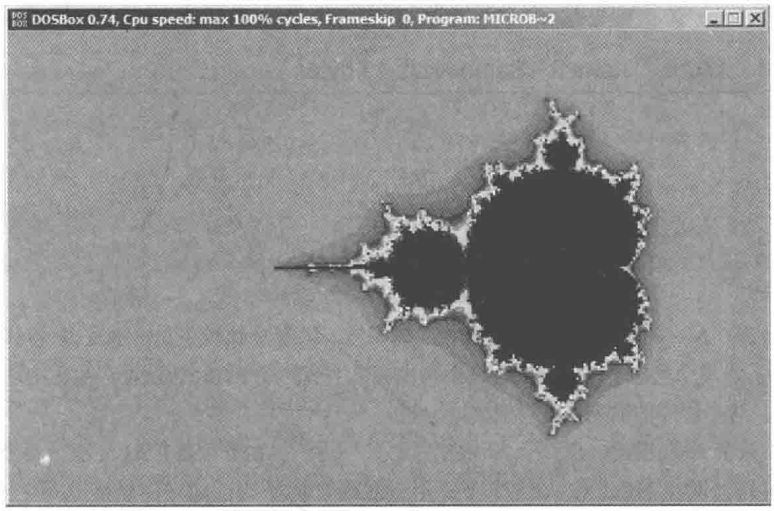


图 83.2 曼德博集合 Demo 程序绘制的图案

本节将介绍这个程序的工作原理。

83.2.1 理论

复数

复数是二元有序实数对，由实部（Re()）和虚部（Im()）两部分构成。在复数概念的二维空间里，任意复数的实部和虚部都可表示为二维坐标，从而把该复数表示为平面的一个点。

本节用到的复数运算的基本法则有：

- 加法：$(a+bi)+(c+di)=(a+c)+(b+d)i$。

即：

$\mathrm{Re}(sum) = \mathrm{Re}(a) + \mathrm{Re}(b)$

$\mathrm{Im}(sum) = \mathrm{Im}(a) + \mathrm{Im}(b)$

- 乘法：$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$。

即：

$\mathrm{Re}(product) = \mathrm{Re}(a) \bullet \mathrm{Re}(c) - \mathrm{Re}(b) \bullet \mathrm{Re}(d)$

$\mathrm{Im}(product) = \mathrm{Im}(b) \bullet \mathrm{Im}(c) + \mathrm{Im}(a) \bullet \mathrm{Im}(d)$

- 平方：$(a+bi)^2 = (a+bi)(a+bi)=(a^2 - b^2)+(2ab)i$。

即：

$\mathrm{Re}(square) = \mathrm{Re}(a)^2 - \mathrm{Im}(a)^2$

$\mathrm{Im}(square) = 2 \bullet \mathrm{Re}(a) \bullet \mathrm{Im}(a)$

曼德博集合的绘制方法

曼德博集合可以由复二次多项式来定义：对于由复数 Z() 构成的递归序列 $z_{n+1} = z_n^2 + c$ 来说，不同的参数 c 可能使序列的绝对值逐渐发散到无限大，也可能收敛在有限的区域内。曼德博集合就是使序列不延伸至无限大的所有复数 c 的集合。

① http://www.pouet.net/prod.php?which=53287。

简单地说，普通程序的做法大致如下：[①]

- 把屏幕划分为像限/取值区域。
- 将每个坐标点视为一个 c 值，并验证它是否属于曼德布洛特集合。
- 验证各坐标的方法如下：
  - 将每个坐标点视为一个复数参数 c。
  - 描述该点的复数值。
  - 计算复数值的平方。
  - 计算上述值与初始值的矢量和[②]。
  - 判断上述结果是否逃逸。如果逃逸则立刻终断。
  - 在一定的迭代次数（n）内，进行复数二项式 $z_{n+1}=z_n^2+c$ 的迭代。
- 如果这个点所代表的 c 值不会使递归序列的值逃逸到无限大，那么就在屏幕上用某种颜色标注这个点。
- 如果这个点所代表的 c 值会使递归序列的值逃逸，则：
  - （黑白构图）不给这个点着色。
  - （彩色构图）把逃逸时的迭代次数转换成某种颜色，并用这种颜色给这个点着色。因此，彩色曼德博集合图上的颜色，描述的是该点的逃逸速度。

笔者编写了两个程序，分别以复数（宏观参数）和代数二项式（表达形式）两种角度实现上述算法。

指令清单 83.3　For complex numbers

```
def check_if_is_in_set(P):
    P_start=P
    iterations=0

    while True:
        if (P>bounds):
            break
        P=P^2+P_start
       if iterations > max_iterations:
           break
       iterations++

    return iterations

# black-white
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        draw point

# colored
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    map iterations to color
    draw color point
```

当参数为复数时，就要使用代数二项式“代入”上述参数，并应用前文介绍过的复数计算法则。

指令清单 83.4　For integer numbers

```
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0
```

① 微软较为全面地介绍了数学方面和编程方面的细节，建议读者阅读：https://msdn.microsoft.com/zh-cn/library/jj635753%28v=vs.85%29.aspx。
② 实际算法都令 z_0=0，因此前几步运算在计算 z_1。请参考前面推荐的微软文档。

```
    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            draw point at X, Y

# colored
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        map iterations to color
        draw color point at X,Y
```

维基百科的网站介绍了一种以 C#语言实现的曼德博集合算法[①]。不过，那个源程序只能用符号显示迭代次数的信息。笔者把它改得更为直观了一些，让它能够显示出序列的迭代次数[②]：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Mnoj
{
    class Program
    {
        static void Main(string[] args)
        {
            double realCoord, imagCoord;
            double realTemp, imagTemp, realTemp2, arg;
            int iterations;
            for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
            {
                for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
                {
                    iterations = 0;
                    realTemp = realCoord;
                    imagTemp = imagCoord;
                    arg = (realCoord * realCoord) + (imagCoord * imagCoord);
                    while ((arg < 2*2) && (iterations < 40))
                    {
                        realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
                        imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                        realTemp = realTemp2;
                        arg = (realTemp * realTemp) + (imagTemp * imagTemp);
                        iterations += 1;
                    }
                    Console.Write("{0,2:D} ", iterations);
                }
                Console.Write("\n");
            }
            Console.ReadKey();
        }
    }
}
```

① https://en.wikipedia.org/wiki/Mandelbrot_set。

② 修改版程序的下载地址是：http://beginners.re/examples/mandelbrot/dump_iterations.exe。

运行上述程序，可得到文件：http://beginners.re/examples/mandelbrot/result.txt。

这个程序限定迭代次数的上限为 40。输出结果中的“40”表示在 40 次迭代之内序列仍然收敛；而 40 以内的数字（比方说是 n），则表示在该点的 c 值在迭代 n 次之后令序列逃逸。

除此之外，http://demonstrations.wolfram.com/MandelbrotSetDoodle/公开了一个精彩的 demo 程序。它能够用彩色线条显示指定 c 值产生的递归序列 z_n。如果彩线落在灰色圆圈（模为 2）之外，那么该点代表的 c 值就不属于曼德博集合。

首先，笔者在黄色区域之内选取一个 c 值，观察它产生的递归序列，如图 83.3 所示。

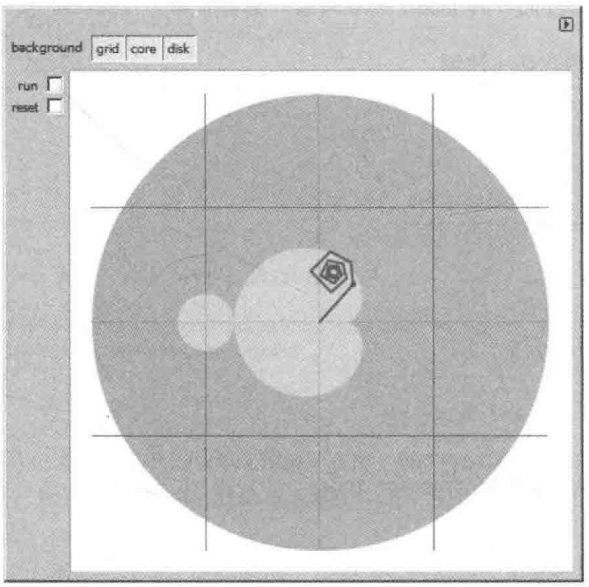


图 83.3　在黄色区域之内选取 c 值

上述线条是收敛的。这表明笔者选取的 c 值属于曼德博集合。

然后，笔者在黄色区域之外选取 c 值。线条立刻混乱、甚至超出边界。如图 83.4 所示。

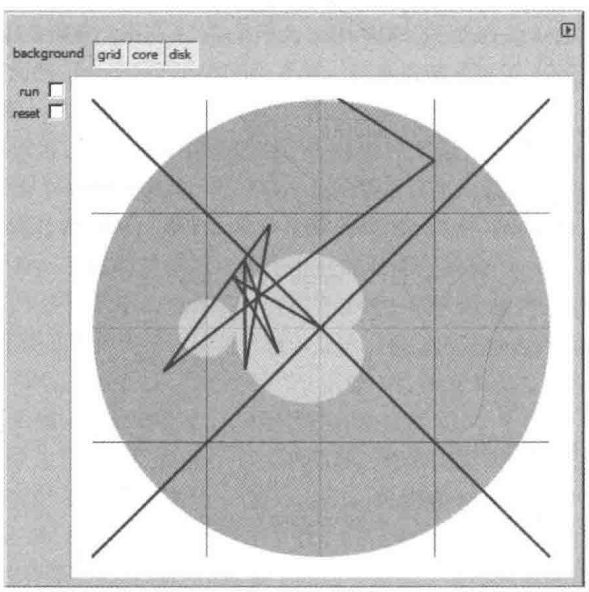


图 83.4　在黄色区域之外选取 c 值

这种图案是发散的。这表示刚才选取的 c 值不属于曼德博集合。

这个网站还开发了另一款曼德博集合的研究程序，有兴趣的读者可访问：http://demonstrations.wolfram.com/IteratesForTheMandelbrotSet/。

83.2.2 demo 程序

本节将要介绍的这个曼德博集合的 demo 程序，仅由 30 条指令、64 个字节构成。虽然它的算法与前文的算法一致，但是它用到了很多编程技巧。

整个程序的源代码是公开的。为了便于读者阅读，笔者添加了一些注释。

指令清单 83.5 程序源代码 + 注释

```
; X is column on screen
; Y is row on screen


; X=0, Y=0              X=319, Y=0
; +------------------------------->
; |
; |
; |
; |
; |
; |
; v
; X=0, Y=199            X=319, Y=199


; switch to VGA 320*200*256 graphics mode
mov al,13h
int 10h
; initial BX is 0
; initial DI is 0xFFFE
; DS:BX (or DS:0) is pointing to Program Segment Prefix at this moment
; ... first 4 bytes of which are CD 20 FF 9F
les ax,[bx]
; ES:AX=9FFF:20CD

FillLoop:
; set DX to 0. CWD works as: DX:AX = sign_extend(AX).
; AX here 0x20CD (at startup) or less then 320 (when getting back after loop),
; so DX will always be 0.
cwd
mov ax,di
; AX is current pointer within VGA buffer
; divide current pointer by 320
mov cx,320
div cx
; DX (start_X) - remainder (column: 0..319); AX - result (row: 0..199)
sub ax,100
; AX=AX-100, so AX (start_Y) now is in range -100..99
; DX is in range 0..319 or 0x0000..0x013F
dec dh
; DX now is in range 0xFF00..0x003F (-256..63)

xor bx,bx
xor si,si
; BX (temp_X)=0; SI (temp_Y)=0

; get maximal number of iterations
; CX is still 320 here, so this is also maximal number of iteration
MandelLoop:
mov bp,si       ; BP = temp_Y
imul si,bx      ; SI = temp_X*temp_Y
add si,si       ; SI = SI*2= (temp_X*temp_Y)*2
```

```
54  imul bx,bx      ; BX = BX^2= temp_X^2
55  jo MandelBreak ; overflow?
56  imul bp,bp      ; BP = BP^2= temp_Y^2
57  jo MandelBreak ; overflow?
58  add bx,bp       ; BX = BX+BP = temp_X^2 + temp_Y^2
59  jo MandelBreak ; overflow?
60  sub bx,bp       ; BX = BX-BP = temp_X^2+temp_Y^2 - temp_Y^2 = temp_X^2
61  sub bx,bp       ; BX = BX-BP = temp_X^2 - temp_Y^2
62
63  ; correct scale:
64  sar bx,6        ; BX=BX/64
65  add bx,dx       ; BX=BX+start_X
66  ; now temp_X = temp_X^2 - temp_Y^2 + start_X
67  sar si,6        ; SI=SI/64
68  add si,ax       ; SI=SI+start_Y
69  ; now temp_Y = (temp_X*temp_Y)*2 + start_Y
70
71  loop MandelLoop
72
73  MandelBreak:
74  ; CX=iterations
75  xchg ax,cx
76  ; AX=iterations. store AL to VGA buffer at ES:[DI]
77  stosb
78  ; stosb also increments DI, so DI now points to the next point in VGA buffer
79  ; jump always, so this is eternal loop here
80  jmp FillLoop
```

这个程序的算法是：

- 切换到分辨率为 320 × 200/256 色的 VGA 模式。320 × 200 = 64000（0xFA00）。256 色的每个像素都是单字节的数据，所以缓冲区大小就是 0xFA00 字节。应用程序使用 ES:DI 寄存器对即可对像素进行寻址。

VGA 图形缓冲区的段地址要存储在 ES 寄存器里，所以 ES 寄存器的值必须是 0xA000。但是向 ES 寄存器传递 0xA000 的指令至少要占用 4 个字节（PUSH 0A000h/POP ES）。有关 6 位 MS-DOS 系统的内存模型，可参见第 94 章的详细介绍。

假设 BX 寄存器的值为零、程序段前缀（Program Segment Prefix）位于第 0 号地址处，那么 2 字节的 LES AX，[BX] 指令就会在 AX 寄存器里存储 0x20CD、在 ES 寄存器里存储 0x9FFF。也就是说，这个程序会在图形缓冲区之前输出 16 个像素（字节）。但是因为该程序的运行平台是 MS-DOS，而 MS-DOS 没有实现内存保护技术，所以这种问题不会引发程序崩溃。不过，屏幕右侧多出了一个 16 像素宽的红色条带，整个图像向左移动了 16 个像素。这就是在程序里节省 2 字节空间的代价。

- 单个循环处理全部像素。在处理图像问题时，一般的程序都会使用两个循环：一个循环遍历 x 坐标、另一个循环遍历 y 坐标。不过，在 VGA 图形缓存区里对某个像素进行定位时，双循环的程序必须通过乘法运算才能寻址。为此，这个程序的作者决定使用单循环的数据结构，通过除法运算获取当前点的坐标。经过转换以后，程序坐标的取值范围是：x～ [−256，63]，y～ [−100，99]。正因如此，整个图像的中心点偏右。虽然直接把 x 的值减去 160 就可以使 x 的取值范围变成 [−160，159]，从而校准图像中心，但是“SUB DX，160”的指令占用 4 个字节，而作者的“DEC DH”（相当于 SUB DX，0x100）只用两个字节。图像中心偏右算是节省文件空间的另一个代价吧。
  - 检测当前点是否属于曼德博集合。检测算法和前文相同。
  - 以 CX 寄存器作为循环计数器进行 LOOP 循环。作者没有明确地给循环计数器赋值，而是让 CX 寄存器继续沿用第 35 行的数值（320）。或许，数值越大越精确吧。这同样可以节省文件空间。
  - 使用 IMUL 指令。因为操作符是有符号数，所以乘法指令不是 MUL。同理，为了把原点坐标 0，0 调整到屏幕中心区域，在转换坐标时（除法）使用到的是 SAR 指令（带符号右移），而不是 SHR 指令。

— 简化逃逸检测的算法。常规算法检测的是坐标，即一对坐标值。而作者分三次检测了溢出问题：两个求平方操作和一次加法运算的溢出。事实正是如此，我们使用的是16位寄存器，寄存器内的数值不会超过［-32768，+32767］。在计算机进行有符号数的乘法运算时，只要坐标的值大于32767，那么该点的复数就不会属于曼德博集合。

— 后面再次出现的SAR指令（相当于除以64）设置了64级素灰度。调色版的灰度值越大，对比度就越高、图像就越清晰；反之，灰度越集中，对比度就越低、图像就越模糊。

- 程序执行到MandelBreak标签的情况分为两种：一种情况是循环结束时CX＝0这就说明，该点属于曼德博集合；另一种情况是程序发生溢出、此时CX存储着非零值。这个程序把CX的低8位（即CL）写到图形缓冲区里。在默认调色板中，0代表黑色，所以属于曼德博集合的坐标点都会显示纯黑色像素。当然，我们确实能够在绘图之前把调色板设置得更个性化一些，不过那种程序就不会只有64字节了！
- 这个程序采用的是无限循环，不能正常退出。如果还要判断循环终止或者进行互动响应，那么程序文件还要更大。

此外，这个程序的优化技巧同样值得一提：

- 使用单字节的CWD指令清空DX寄存器。相比之下，“XOR DX，DX”占用2个字节，而“MOV DX，0”则占用3个字节。
- 使用单字节的“XCHG AX，CX”完成双字节“MOV AX，CX”的操作。毕竟后面不再使用AX寄存器，所以交换数据完全不会造成问题。
- 因为程序没有对DI寄存器（图形缓冲区的位置）进行初始化赋值，所以它在启动时的初始值为0xFFFE（寄存器初始值由操作系统决定）。不过这无伤大雅，这个程序只要求DI的值保持在［0，0xFFFF］之间，软件用户也看不到屏幕区域之外的点（在320×200×256的图像缓冲区里，最后一个像素的地址是0xF9FF）。也就是说，因为这个程序前后衔接得严丝合缝，所以不管DI寄存器也没问题。否则，编程人员还要添加把DI寄存器置零、以及检测图形缓冲区结束边界的指令。

83.2.3 笔者的改进版

指令清单83.6 My “fixed” version

```
 1 org 100h
 2 mov al, 13h
 3 int 10h
 4
 5 ; set palette
 6 mov dx, 3c8h
 7 mov al, 0
 8 out dx, al
 9 mov cx, 100h
10 inc dx
11 l00:
12 mov al, cl
13 shl ax, 2
14 out dx, al ; red
15 out dx, al ; green
16 out dx, al ; blue
17 loop l00
18
19 push 0a000h
20 pop es
21
22 xor di, di
23
24 FillLoop:
25 cwd
26 mov ax, di
27 mov cx, 320
```

```
28 div cx
29 sub ax, 100
30 sub dx, 160
31
32 xor bx, bx
33 xor si, si
34
35 MandelLoop:
36 mov bp, si
37 imul si, bx
38 add si, si
39 imul bx, bx
40 jo MandelBreak
41 imul   bp, bp
42 jo MandelBreak
43 add bx, bp
44 jo MandelBreak
45 sub bx, bp
46 sub bx, bp
47
48 sar bx, 6
49 add bx, dx
50 sar si, 6
51 add si, ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax, cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; wait for keypress
62 xor ax, ax
63 int 16h
64 ; set text video mode
65 mov ax, 3
66 int 10h
67 ; exit
68 int 20h
```

笔者修正了上一个程序的缺陷：使用了平滑过渡的灰度调色板；把整个图形全部输出到了图形缓冲区里（第 19、20 行）；使图像中心与屏幕中心重合（第 30 行）；绘图结束后等待键盘敲击再退出程序（第 58～68 行）。不过，整个程序大了近一倍：它由 54 条指令构成，文件大小也增长到了 105 字节。该程序的绘图如图 83.5 所示。

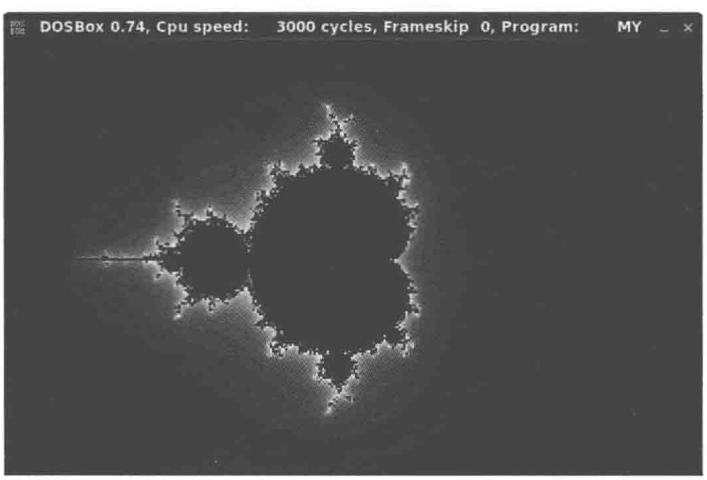


图 83.5　笔者的改进版程序的绘图

第九部分

文件分析



第 84 章　基于 XOR 的文件加密

84.1　Norton Guide：单字节 XOR 加密实例

在 MS-DOS 时代，Norton Guide（http://en.wikipedia.org/wiki/Norton_Guides）风靡一时。它是一款驻留内存的 TSR 程序，可与编程语言的编辑程序整合，提供超文本形式的参考信息。

Norton Guide 的数据库文件是.ng 文件。一看便知，这种文件经过加密处理。如图 84.1 所示。

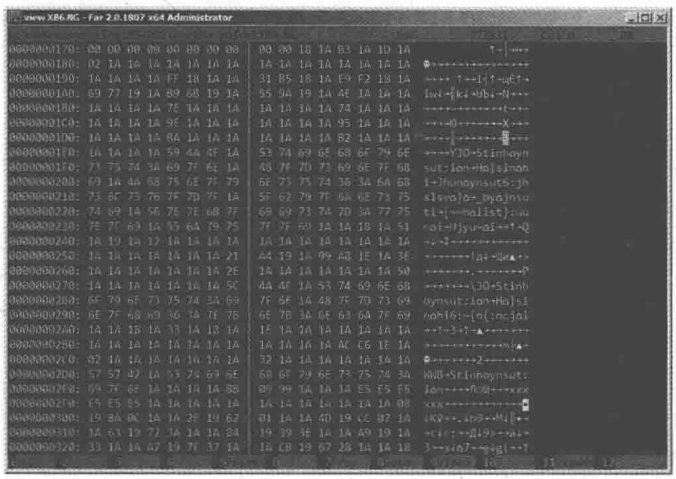


图 84.1　Very typical look

为什么说它是加密文件而非压缩文件呢？我们可以看到数值为 0x1A 的字节（右箭头字符）多次出现，而压缩文件则不会发生这种情况。我们还看到了大量有拉丁字符的片段，只是这些字符串不可自然解释。

因为文件多次出现了 0x1A，所以我们按照加密文件进行处理，并且假设该文件是经异或/XOR 加密的密文。我们可在 Hiew 里清楚地看到，当使用 0x1A 对每个字节进行异或运算时，文件里出现了亲切的英语文字。如图 84.2 所示。

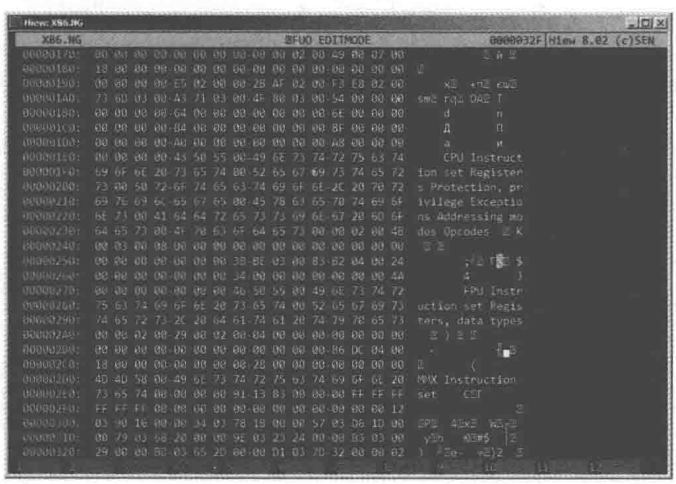


图 84.2　Hiew XORing with 0x1A

对每个字节进行 XOR 运算是一种最初级的加密方法。不要因为其简单就瞧不起这种分析方法，实际的分析工作中就是会频繁遇到这种技术。

现在，我们大体理解了 0x1A 出现次数很多的原因了。在进行异或运算之后，数值为 0 的明文字节会演变为数值为 0x1A 的密文。

应当注意：加密的常量会因文件而异。在解密单字节 XOR 加密的密文时，我们应当尝试 0～255 之间的每个数，再分析一下密文文件被解密成什么样子。有关 Norton Guide 的文件格式，请参见：http://www.davep.org/norton-guides/file-format/。

信息熵

在加密领域，信息熵（entropy）属于重要的信息指标。它有一个重要特性：加密前后的明文和密文，其信息熵不变。本节将介绍使用 Wolfram Mathematica 10 来计算信息熵的具体方法。

指令清单 84.1 Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];

In[2]:= Entropy[2, input] // N
Out[2]= 5.62724

In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];

In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];

In[5]:= Entropy[2, decrypted] // N
Out[5]= 5.62724

In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N
Out[6]= 4.42366
```

上述各指令分别用于加载文件、计算信息熵、解密、保存和计算明文的信息熵（熵不变）。Mathematica 还提供了知名的英文片段以供人们进行分析。我选取了莎士比亚的十四行韵律诗进行分析，其信息熵与前一个例子基本相同。我们分析的英文语句，其信息熵与莎士比亚的语言相似。对英文原文进行单字节的 XOR 加密之后，其信息熵与原文相同。

但是，如果加密单元大于一个字节，那么信息熵就是另外一种情况了。

本节分析的英文原文，可在下述地址下载：http://beginners.re/examples/norton_guide/X86.NG。

其他

Wolfram Mathematica 计算的熵以自然指数 e 为基数，而 UNIX 的 ent 工具[①]则以 2 为基数。所以上例明确指定“以 2 为基数”，以使得 Mathematica 的计算结果与 ent 工具的计算结果相同。

84.2 4 字节 XOR 加密实例

即使 XOR 算法采用多字节密钥，例如说 4 字节密钥，分析方法也没有什么两样。本节以 32 位 Windows Server 2008 的 Kernel32.dll 为例进行说明。源文件如图 84.3 所示。

以 4 字节密钥进行加密，可得到图 84.4 所示的结果。

通过观察文件，就可以看到一组循环出现的 4 字节字符串。实际上这并不困难，因为 PE 文件的文件头中含有大量的零字节，所以我们可以直接看到密钥。

在 16 进制编辑器里，PE 文件头大体如图 84.5 所示。

① 官方网站为 http://www.fourmilab.ch/random/。

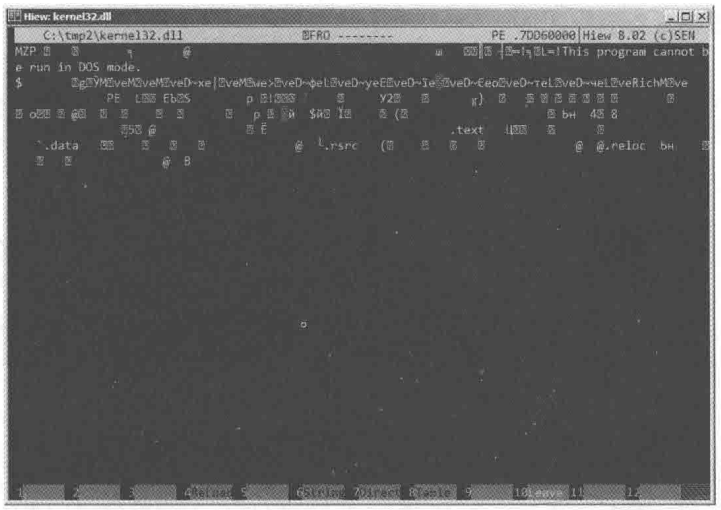


图 84.3　源文件

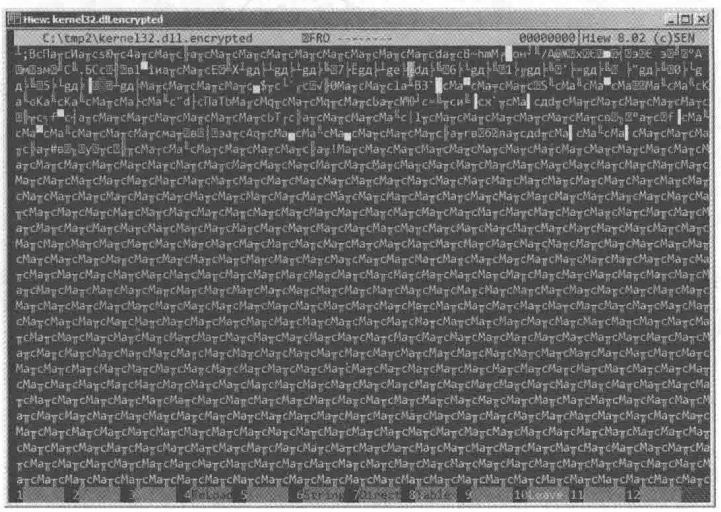


图 84.4　文件密文

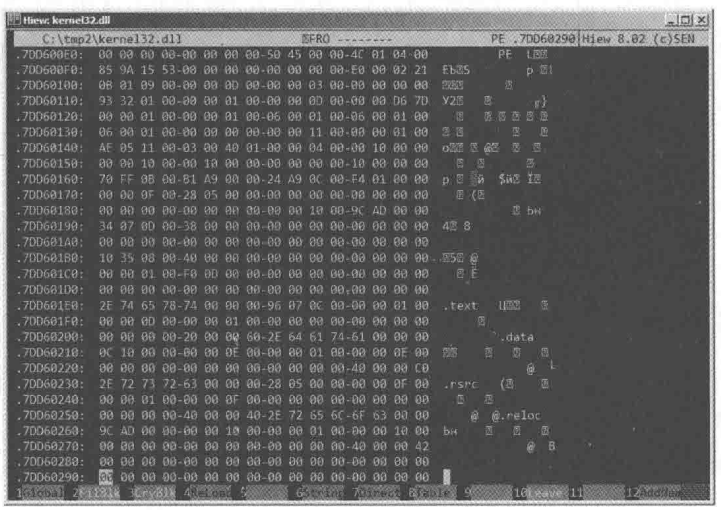


图 84.5　PE 文件头

加密之后，如图 84.6 所示。

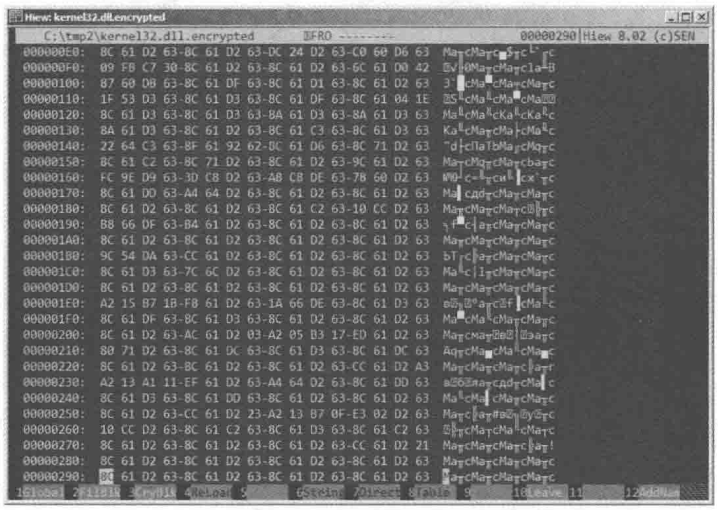


图 84.6 加密后的 PE 文件头

观察可得 4 字节密钥：8C 61 D2 63。使用这个消息块即可对文件解密。

此处不得不提 PE 文件的几个特点：

① PE 文件头里含有大量的零字节。

② 所有的 PE 字段都向分页边界——4096 字节对齐，用零字节填补空缺；所以每个字段之后肯定存在大量的零字节。

用零来实现边界对齐的文件格式并不罕见。很多科学计算软件及工程类软件都采用了这种文件格式。

有兴趣的读者可研究一下本例的文件。它们的下载地址是：http://beginners.re/examples/XOR_4byte/。

84.3 练习题

请尝试解密下列链接中的密文。

http://go.yurichev.com/17353

第 85 章　Millenium 游戏的存档文件

“Millenium Return to Earth”是一款古老的 DOS 游戏（1991 年问世）。玩家可在游戏中挖矿、修建战舰、在其他星球上作战，等等。有兴趣的读者可以体验一下它：http://thehouseofgames.org/index.php?t=10&id=110。

和其他的游戏程序一样，这个游戏也有游戏存档的功能。现在我们来分析一下它的存档文件。

游戏中有“矿”的概念。挖矿的速度因星球而异，在某些星球上快些，而在另一些星球上慢些。另外，在游戏的设定中，矿产的种类也有差异化的设定。在图 85.1 中，您可看到游戏存盘时的挖矿进度。



图 85.1　Mine：状态 1

我保存了这一时刻的游戏状态。存盘文件总计 9538 字节。

然后，我又在游戏里挖了几“天”的矿。挖矿进度如图 85.2 所示。



图 85.2　Mine：状态 2

此时我再次保存了游戏状态。

现在，我们使用 DOS/Windows 的 FC 程序来比较两个存档文件的差别：

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68
...
```

上述内容是比对结果的部分内容。两个文件的不同之处还有很多，但是其余内容不如这些信息那样富有代表性。

在第一次存盘时，我持有 14 个单位的 hydrogen 和 102 个单位的 oxygen。在第二次存盘时，相应的持有量变为 22 和 155 个单位。如果程序把这两个值存储在存档文件中，那么我们应当可以在存档里找到它。实际情况正是如此。在存档文件的 0xBDA 处，第一个存档文件的值为 0x0E（14），在第二个存档文件的值为 0x16（22）。地址 0xBDA 存储的应当是 hygrogen 的量。然后，在文件的 0xBDC 处，两个值分别为 0x66（102）和 0x9B（155）。地址 0xBDC 存储的应当是 oxygen 的值。

您可以自己把玩一下这个游戏，分析存档文件的具体格式。您还可以下载我用的游戏存档：http://beginners.re/examples/millenium_DOS_game/。

使用 Hiew 打开第二次存档的存档文件，可以看到有关矿石的持有量。如图 85.3 所示。

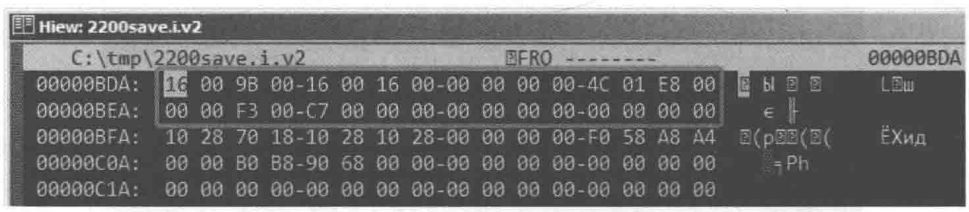


图 85.3 Hiew：状态 1

这个值无疑是 16 位数值：在 DOS 时代的 16 位软件程序里，int 型数据就是 16 位数据，这并不意外。验证一下我们的推测是否正确。把这个地址的值（hydrogen）改为 1234（0x4D2），如图 85.4 所示。

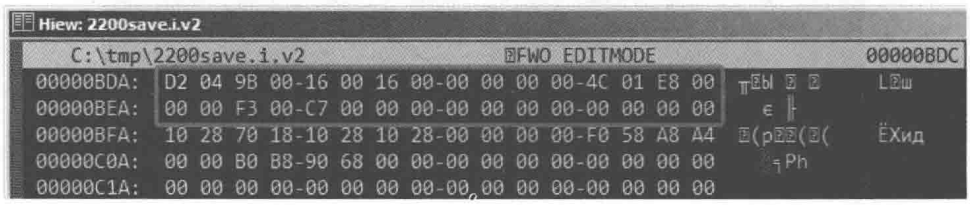


图 85.4 Hiew：把数值修改为 1234（0x04D2）

然后打开游戏、加载存档中的进度，看看矿产持有量。如图 85.5 所示。



图 85.5　Let's check for hydrogen value

以上信息表明，我们的推测是正确的。

为了快速通关，我们把所有矿产的持有量都改成最大值，如图 85.6 所示。

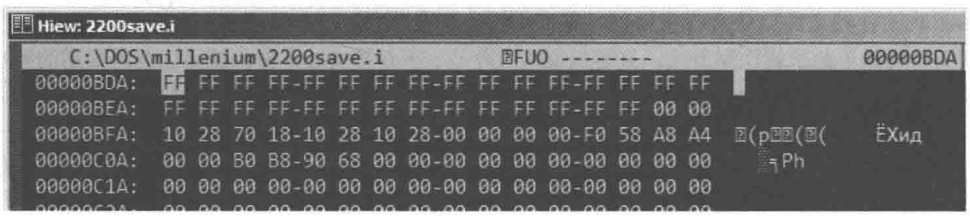


图 85.6　Hiew：把各项都修改为最大值

0xFFFF 是 65535。改动之后，我们就是资源大亨了。如图 85.7 所示。



图 85.7　所有资源都变为 65535

在进行了几个游戏日的奋斗之后——哎？部分资源变少了。如图 85.8 所示。



图 85.8 Resource variables overflow

这就发生了数值溢出。游戏开发人员可能没有想到玩家会持有这么多的矿产，所以未做溢出检测。但是挖矿就会增加矿产，超过数据最大值之后、数据溢出了。这样看来，要是我当初没那么贪心就好了，或许吧。

这款游戏的存档文件里还有很多数值，本文不再一一分析。

这属于一种简单的游戏作弊方法。只要玩家略微改动一下存档文件，他们就可以获得很高的游戏分值。

本书 63.4 节详细介绍了各种文件及内存快照的比较方法。

第 86 章　Oracle 的.SYM 文件

在程序崩溃的时候，Oracle RDBMS 会把大量信息写到日志文件（log）里。日志文件会记录数据栈的使用情况，如下所示。

```
----- Call Stack Trace -----
calling              call     entry              argument values in hex
location             type     point              (? means dubious value)
-------------------- -------- ------------------ ------------------
_kqvrow()                     00000000
_opifch2()+2729      CALLptr  00000000            23D4B914 E47F264 1F19AE2
                                                  EB1C8A8 1
_kpoal8()+2832       CALLrel  _opifch2()          89 5 EB1CC74
_opiodr()+1248       CALLreg  00000000            5E 1C EB1F0A0
_ttcpip()+1051       CALLreg  00000000            5E 1C EB1F0A0 0
_opitsk()+1404       CALL???  00000000            C96C040 5E EB1F0A0 0 EB1ED30
                                                  EB1F1CC 53E52E 0 EB1F1F8
_opiino()+980        CALLrel  _opitsk()           00
_opiodr()+1248       CALLreg  00000000            3C 4 EB1FBF4
_opidrv()+1201       CALLrel  _opiodr()           3C 4 EB1FBF4 0
_sou2o()+55          CALLrel  _opidrv()           3C 4 EB1FBF4
_opimai_real()+124   CALLrel  _sou2o()            EB1FC04 3C 4 EB1FBF4
_opimai()+125        CALLrel  _opimai_real()      2 EB1FC2C
_OracleThreadStart@  CALLrel  _opimai()           2 EB1FF6C 7C88A7F4 EB1FC34 0
4()+830                                           EB1FD04
77E6481C             CALLreg  00000000            E41FF9C 0 0 E41FF9C 0 EB1FFC4
00000000             CALL???  00000000
```

既然是编译器生成的程序，那么 Oracle 的可执行程序里必定会有调试信息、带有符号（symbol）信息的映射文件、或者是相似的信息。

在 Windows NT 版的 Oracle RDBMS 中，其可执行文件里存在着与.SYM 文件有关的符号信息。可惜，官方不会公开.SYM 文件的文件格式。固然 Oracle 可以使用纯文本文件，但是如此一来还要对其进行多次转换，性能必然大打折扣。

我们从最短的文件 orawtc8.sym 入手，试着分析这种格式的文件。Oracle 8.1.7 的动态库文件 orawtc8.dll 之中，存在着这个.SYM 文件的符号信息。对于 oracle 数据库的程序来说，版本越旧、功能模块的文件就越小。

使用 Hiew 打开上述文件，可以看到如图 86.1 所示的界面。

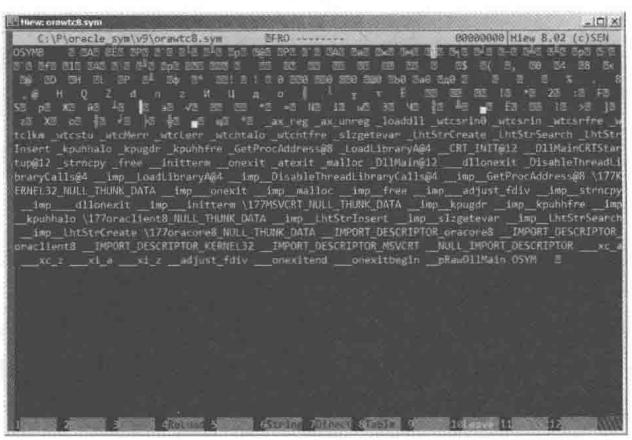


图 86.1　使用 Hiew 打开整个文件

参照其他.SYM 文件，可知这种格式的文件头（及文件尾部）都有 OSYM 字样。据此判断，这个字符串可能是某种文件签名。

大体来说，这种文件的格式是“OSYM + 某些二进制数据 + 以 0 做结束符的字符串 + OSYM”。很明显，这些文件中的字符串应当是函数名和全局变量名。

如图 86.2 所示，字符串 OSYM 的位置较为固定。

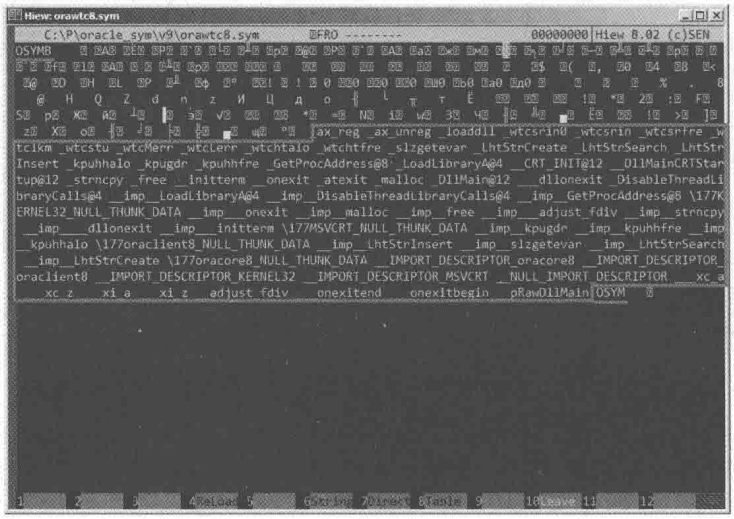


图 86.2 OSYM signature and text strings

接下来，我把这个文件中的整个字符串部分（不包含尾部的 OSYM 签名字符串）复制了出来，并单独存储为一个文件 strings_block。然后使用 UNIX 的 strings 和 wc 工具统计它字符串的数量：

```
strings strings_block | wc -l
66
```

可见它包含 66 个文本字符串。我们先把这个数字记下来。

通常来说，无论它是字符串、还是其他什么类型的数据，数据的总数往往会出现在二进制文件的其他部分。这次的分析过程再次印证了这个规律，我们可以在文件的开始部分、OSYM 之后看到 66（0x42）：

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00  00 10 00 10 80 10 00 10  |OSYMB...........|
00000010  f0 10 00 10 50 11 00 10  60 11 00 10 c0 11 00 10  |....P...`.......|
00000020  d0 11 00 10 70 13 00 10  40 15 00 10 50 15 00 10  |....p...@...P...|
00000030  60 15 00 10 80 15 00 10  a0 15 00 10 a6 15 00 10  |`...............|
....
```

当然，0x42 不是一个 byte 型数据，很可能是个以小端字节序存储的 32 位数据。正因如此，0x42 之后排列着 3 个以上的零字节。

判断它是 32 位数据的依据是什么？Oracle RDBMS 的符号文件可能非常大。以版本号为 10.2.0.4 的 Oracle 主程序为例，它的 oracle.sym 包含有 0x3A38E（即 238478）个符号。16 位的数据类型不足以表达这个数字。

分析过其他.SYM 文件之后，我更加确定了上述猜测：在 32 位的 OSYM 签名之后的数据，就是反映文本字符串数量的数据。

这也是多数二进制文件的常规格式：文件头通常包含程序签名和文件中的某种信息。

接下来，我们分析一下文件中的二进制部分。我把文件中第 8 字节（字符串计数器之后）到字符串之间的内容存储为另外一个文件（binary_block）。然后再使用 Hiew 打开这个新文件，如图 86.3 所示。

这个文件的模式逐渐清晰了起来。为了便于理解，我在图中添加了几条分割线，如图 86.4 所示。



图 86.3　Binary block

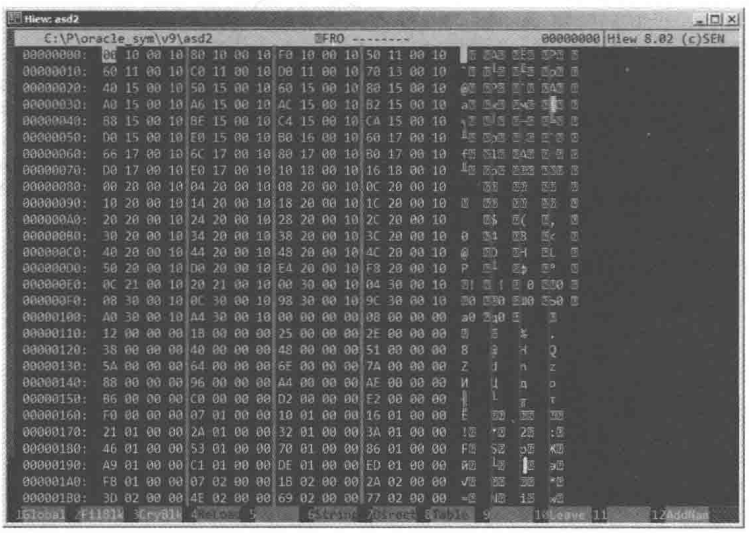


图 86.4　Binary block

多数的 hex 编辑器每行都显示 16 个字节，Hiew 也不例外。所以，在 Hiew 的窗口里，每行信息对应着 4 个 32 位数据。

文件中的数据凸显了它的这种特征：在地址 0x104 之前的数据都是 0x1000xxx 形式的数据（请注意小端字节序），数据都以 0x10、0x00 字节开头；以 0x108 开始的数据，都是 0x0000xxxx 型的数据，都以两个零字节开头。

我们把这些数据整理为 32 位的数组，代码如下所示。

指令清单 86.1　第一列是地址

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
```

```
0000200 10002000 10002004 10002008 1000200c
0000220 10002010 10002014 10002018 1000201c
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051
0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020
```

这里有 132 个值，是 66×2 的阵列。字符串的总量正好是 66。那么，到底是每个字符串符号对应了 2 个 32 位数据，还是说这 2 个 32 位数据完全就是两个互不相干数组呢？我们继续分析。

以 0x1000 开头的值可能是某种地址。毕竟.SYM 文件是为.DLL 文件服务的，而且 Win32 DLL 文件的默认基址是 0x10000000，代码的起始地址通常是 0x10001000。

使用 IDA 工具打开 orawtc8.dll 文件，可以看到它的基址不是默认地址。尽管如此，我们可以看到它的第一个函数的对应代码为：

```
.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0           = dword ptr  8
.text:60351000 arg_4           = dword ptr  0Ch
.text:60351000 arg_8           = dword ptr  10h
.text:60351000
.text:60351000                 push    ebp
.text:60351001                 mov     ebp, esp
.text:60351003                 mov     eax, dword_60353014
.text:60351008                 cmp     eax, 0FFFFFFFFh
.text:6035100B                 jnz     short loc_6035104F
.text:6035100D                 mov     ecx, hModule
.text:60351013                 xor     eax, eax
.text:60351015                 cmp     ecx, 0FFFFFFFFh
.text:60351018                 mov     dword_60353014, eax
.text:6035101D                 jnz     short loc_60351031
.text:6035101F                 call    sub_603510F0
.text:60351024                 mov     ecx, eax
.text:60351026                 mov     eax, dword_60353014
.text:6035102B                 mov     hModule, ecx
.text:60351031
.text:60351031 loc_60351031:                           ; CODE XREF: sub_60351000+1D
.text:60351031                 test    ecx, ecx
.text:60351033                 jbe     short loc_6035104F
.text:60351035                 push    offset ProcName ; "ax_reg"
.text:6035103A                 push    ecx             ; hModule
.text:6035103B                 call    ds:GetProcAddress
...
```

喔，我们好像见过字符串“ax_reg”！它不就是在.SYM 文件的字符串区里的第一个字符串嘛！可见，

这个函数的名字应该就是“ax_reg”。

上述 DLL 文件的第二个函数是：

```
.text:60351080 sub_60351080    proc near
.text:60351080
.text:60351080 arg_0           = dword ptr  8
.text:60351080 arg_4           = dword ptr  0Ch
.text:60351080
.text:60351080                 push    ebp
.text:60351081                 mov     ebp, esp
.text:60351083                 mov     eax, dword_60353018
.text:60351088                 cmp     eax, 0FFFFFFFFh
.text:6035108B                 jnz     short loc_603510CF
.text:6035108D                 mov     ecx, hModule
.text:60351093                 xor     eax, eax
.text:60351095                 cmp     ecx, 0FFFFFFFFh
.text:60351098                 mov     dword_60353018, eax
.text:6035109D                 jnz     short loc_603510B1
.text:6035109F                 call    sub_603510F0
.text:603510A4                 mov     ecx, eax
.text:603510A6                 mov     eax, dword_60353018
.text:603510AB                 mov     hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1:                           ; CODE XREF: sub_60351080+1D
.text:603510B1                 test    ecx, ecx
.text:603510B3                 jbe     short loc_603510CF
.text:603510B5                 push    offset aAx_unreg ; "ax_unreg"
.text:603510BA                 push    ecx             ; hModule
.text:603510BB                 call    ds:GetProcAddress
...
```

“ax_unreg”是字符串区域里的第二个字符串。第二个函数的起始地址是 0x60351080，而在 SYM 文件里二进制区域的第二个数值正是 10001080。据此推测，文件里的这个值应该就是相对地址，只不过，这个相对地址的基地址不是默认的 DLL 基址罢了。

简短截说，在.SYM 文件中那个 66×2 的数据里，前半部分 66 个数值是 DLL 文件里的函数地址。它们也可能是函数里某个标签的相对地址。那么，由 0x0000 开头的、余下的 66 个值表达的是什么信息呢？这些数据的取值区间是［0，0x3f8］。它不像是位域的值，只是某种递增序列。关键问题是：每个值的最后一个数之间没有什么明确关系，它也不像是某种地址信息——地址的值应该是 4、8 或 0x10 的整数倍。

不妨直接问问您自己：如果您是研发人员，还要在这个文件里写什么数据？即便是瞎猜，也会猜得八九不离十：目前还缺少文本字符串（函数名）在文件里的地址信息。简单验证可知，的确如此，这些数值与字符串的第一个字母的地址存在对应关系。

大功告成。

此外，我还写了一段把.SYM 文件中的函数名加载到 IDA 脚本的程序，以便.idc 脚本文件自动解析函数的函数名：

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (intargc, char *argv[])
{
        uint32_t sig, cnt, offset;
        uint32_t *d1, *d2;
        int     h, i, remain, file_len;
        char    *d3;
```

```
        uint32_t array_size_in_bytes;

        assert (argv[1]); // file name
        assert (argv[2]); // additional offset (if needed)

        // additional offset
        assert (sscanf (argv[2], "%X", &offset)==1);

        // get file length
        assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
        assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
        assert (lseek (h, 0, SEEK_SET)!=-1);

        // read signature
        assert (read (h, &sig, 4)==4);
        // read count
        assert (read (h, &cnt, 4)==4);

        assert (sig==0x4D59534F); // OSYM

        // skip timedatestamp (for 11g)
        //_lseek (h, 4, 1);

        array_size_in_bytes=cnt*sizeof(uint32_t);

        // load symbol addresses array
        d1=(uint32_t*)malloc (array_size_in_bytes);
        assert (d1);
        assert (read (h, d1, array_size_in_bytes) == array_size_in_bytes);

        // load string offsets array
        d2=(uint32_t*)malloc (array_size_in_bytes);
        assert (d2);
        assert (read (h, d2, array_size_in_bytes) ==array_size_in_bytes);

        // calculate strings block size
        remain=file_len-(8+4)-(cnt*8);

        // load strings block
        assert (d3=(char*)malloc (remain));
        assert (read (h, d3, remain)==remain);

        printf ("#include <idc.idc>\n\n");
        printf ("static main() {\n");

        for (i=0; i<cnt; i++)
                printf ("\tMakeName(0x%08X, \"%s\");\n", offset + d1[i], &d3[d2[i]]);

        printf ("}\n");

        close (h);
        free (d1); free (d2); free (d3);
};
```

使用这个脚本以后，我们可以看到：

```
#include <idc.idc>

static main() {
        MakeName(0x60351000, "_ax_reg");
        MakeName(0x60351080, "_ax_unreg");
        MakeName(0x603510F0, "_loaddll");
        MakeName(0x60351150, "_wtcsrin0");
        MakeName(0x60351160, "_wtcsrin");
```

```
        MakeName(0x603511C0, "_wtcsrfre");
        MakeName(0x603511D0, "_wtclkm");
        MakeName(0x60351370, "_wtcstu");
...
}
```

如需下载本章用到的 oracle 文件，请访问：http://beginners.re/examples/oracle/SYM/。

此外，我们来研究一下 Win64 下的 64 位 oracle RDBMS。64 位程序的指针肯定就是 64 位数据了吧！这种情况下，8 字节数据的数据特征就更为明显了。如图 86.5 所示。



图 86.5　RDBMS for Win64 的.SYM 文件（示例）

可见，数据表的所有元素都是 64 位数据，字符串偏移量也不例外。此外，大概是为了区别不同的操作系统，文件的签名改成了 OSYMAM64。

如需让 IDA 自动加载.SYM 文件中的函数名，可参考我的样本程序：https://github.com/dennis714/porg/blob/master/oracle_sym.c。

第 87 章　Oracle 的.MSDB 文件

在解决问题时，如果解是已知的，
那么你总会有章可循。

《墨菲定律－精确的法则》

Oracle 的.MSDB 文件是一种含有错误信息和相应错误编号的二进制文件。本章将与您共同研究它的文件格式，尝试解读其中的原始数据。

虽然 Oracle RDBMS 提供专门的、文本格式的错误信息文件，但是并非每个.MSB 文件里都有相应的、文本格式的错误信息文件，所以有时需要把二进制文件和信息文本进行关联分析。

过滤掉 ORAUS.MSG 的注释以后，文件开头部分的内容如下所示：

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process;cannot switch session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory;cannot detach session"
00024, 00000, "logins from more than one process not allowed in single-process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
```

其中，第一个数字是错误编号，第二个数字可能是某种特殊的标识信息。

现在，我们打开 ORAUS.MSB 的二进制文件，然后找到这些字符串，如图 87.1 所示。

文本字符串之间掺杂着二进制的数据。简单分析之后，可知文件的主体部分可分为多个固定长度的信息块，每个信息块的大小是 0x200（512）字节。

首先查看第一个信息块的数据，如图 87.2 所示。

可以看到第一条错误信息的文本内容。此外，我们还注意到错误信息之间没有零字节；也就是说，这些字符串不是以零字节分割的 C 语言字符串。作为一种替代机制，文件中必须有某个数据记录字符串的长度。

然后我们来找找它的错误代码。参照 ORAUS.MSG 文件起始部分的错误编号，我们在.msb 文件中找到取值为错误编号的几个字节：0，1，17（0x11），18（0x12），19（0x13），20（0x14），21（0x15），22（0x16），23（0x17），24（0x18）……笔者在这个信息块里找到了这些数字，并且在图 87.2 里用红线标出它们。相邻两个错误代码之间的空间周期是 6 个字节。这意味着每条错误信息可能都占用 6 个字节。

第一个 16 位值（0xA 即 10）代表着每个信息块包含的错误信息的总数——其他信息块的调查结果印证了这一猜想。稍微想一下就知道错误信息（文本字符串）的长度不会是一致的。这种字符串有长有短，但是信息块的尺寸却是固定的。所以，程序无法事先知道每个信息块装载了多少个文本字符串。

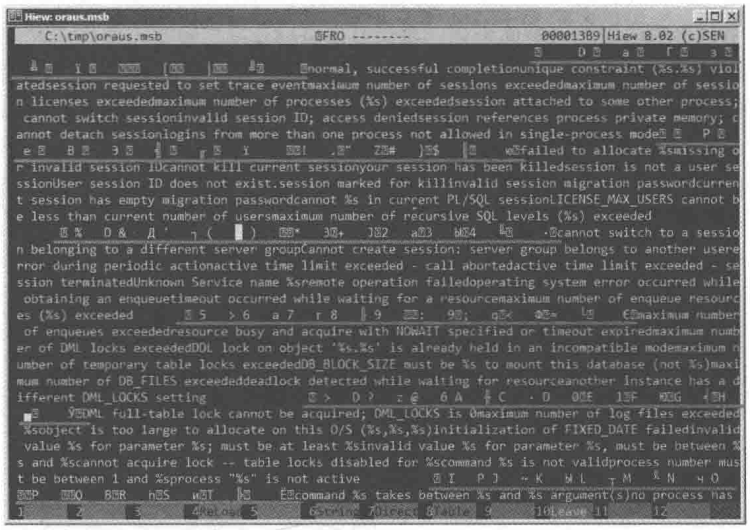


图 87.1　Hiew：第一个消息块

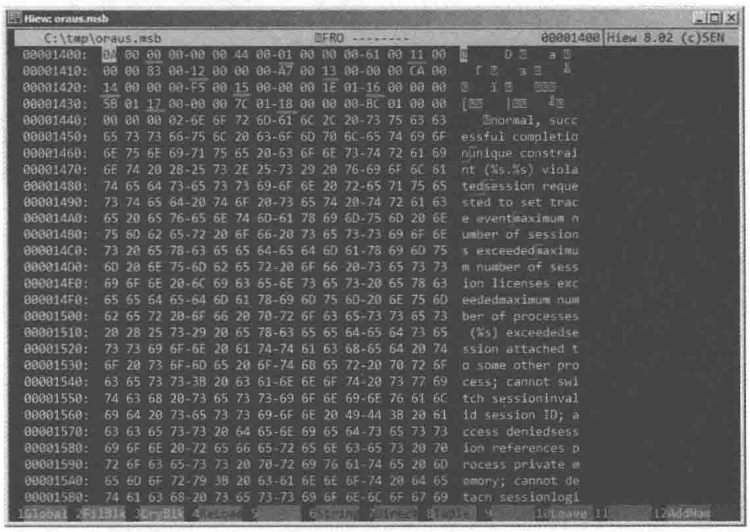


图 87.2　Hiew：第一个消息块

刚才讲过，这些字符串不是以零字节分割的 C 语言字符串，其字符串长度肯定位于文件中的其他什么地方。字符串“normal，successful completion”有 29（0x1D）字节，另一个字符串“unique constraint（%s.%s）violated”有 34 个（0x22）字节。但是在这个信息块里，我们找不到 0x1d 或者 0x22。

一般来说，Oracle RDBMS 应当需要确定每个字符串在信息块之中的相对位置。第一个字符串“normal，successful completion”在整个文件中的绝对位置是 0x1444，在信息块中的相对位置是 0x44。同理可得第二个字符串“unique constraint（%s.%s）violated”的位置 0x1461 和 0x61。这两个数（0x44 和 0x61）并不陌生！在信息块中的前几个字节里就有这些数字。

综上所述，我们分析出了各个信息块里 6 字节非文本信息的格式：

- 16 位错误代码。
- 16 位的零（可能含有其他的标识信息）。
- 16 位的字符串地址信息，用于标记字符串在当前信息块的相对位置。

这些猜想可以被事实验证。在信息块中的最后一个 6 字节的、错误信息为“dummy”的信息块，它的

错误编号为零、起始位置指向最后一个错误信息的最后一个字符之后的位置。或许这个凑数的字符串偏改量用于标记上一个字符串的结束符？至此为止，我们可以根据 6 字节数据的信息，索引指定的错误编号，从而获取文本字符串的起始位置。我们还知道源程序会根据下一个 6 字节数据块推算本字符串的文本长度。这样一来，我们可以确定字符串的界限。这种文件格式不必存储字符串的长度，因而十分节省空间。我们可能无法判断它最终能压缩多少文件空间，但是这无疑是一种不错的思路。

此后，我们返回来分析.MSB 文件的文件头信息。信息的总数如图 87.3 中红线部分所示。

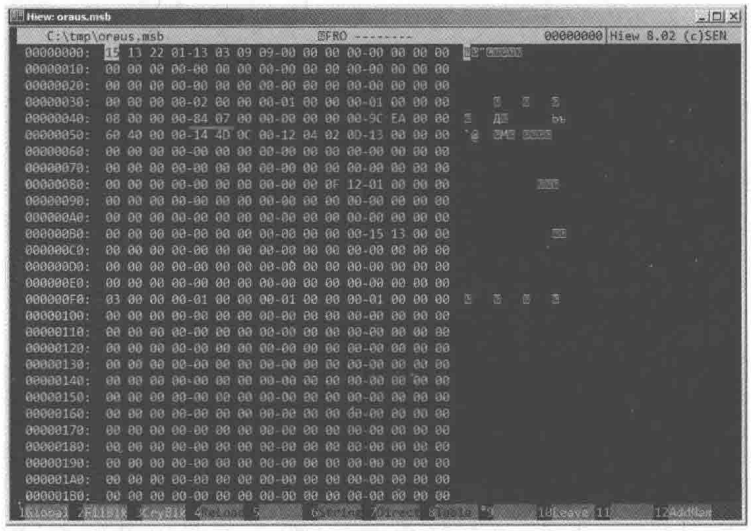


图 87.3 Hiew：文件头

对其他.MSB 文件进行了验证之后，我确信所有的推测都准确无误。虽然文件头中还富含其他信息，但是我们的目标（供调试工具调用）已经达成，故而本文不再分析那些数据。除非我们要编写一个.MSB 文件的封装程序，否则就不需要理解其他数据的用途。

如图 87.4 所示，在文件头之后还有一个 16 位数值构成的数据表。

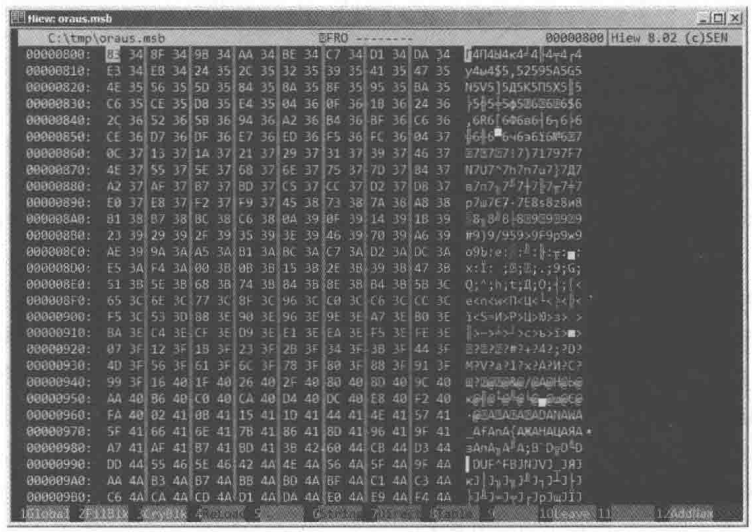


图 87.4 Hiew：last_errnos 表

如图 87.4 中的红线所示，这些数据的数据宽度可以直接观察出来。在导出这些数据时，我发现这些 16

位数据就是每个信息块里最后一个错误信息的错误编号。

可见，Oracle RDBMS 通过这部分数据进行快速检索的过程大体如下：

- 加载 last_errnos（随便起的名字）数据表。这个数据表包含每个信息块里的错误信息总数。
- 依据错误编号找到相应的信息块。此处假设各信息块中的信息以错误代码的增序排列。
- 加载相应的信息块。
- 逐一检索 6 字节的索引信息。
- 通过当前 6 字节数据找到字符串的第一个字符的位置。
- 通过下一个 6 字节数据找到最后一个字符的位置。
- 加载这个区间之内的全部字符。

我编写了一个展开.MSB 信息的 C 语言程序，有兴趣的读者可通过下述网址下载：http://beginners.re/examples/oracle/MSB/oracle_msb.c。

本例还用到了 Oracle RDBMS 11.1.06 的两个文件，如需下载请访问：

- go.yurichev.com/17214。
- go.yurichev.com/17215。

总结

对于现在的计算机系统来说，本章介绍的这种方法可能已经落伍了。恐怕只有那些在 20 世纪 80 年代中期做过大型工程、时刻讲究内存和磁盘的利用效率的老古董才会制定这样严谨的文件格式。无论怎样，这部分内容颇具代表性。我们可以在不分析 Oracle RDBMS 代码的前提下理解它的专用文件的文件格式。

第十部分

其他



第 88 章　npad

“npad”指令是一种汇编宏，用于把下一个指令标签的首地址向指定边界对齐。

被 npad 指令对齐的标签，通常都是需要被多次跳转到的地址标签。例如，在各种循环体起始地址处的标签之前，我们经常可以看到 npad 指令。它可通过对齐内存地址、内存总线或缓存线等手段，提高 CPU 加载数据（或指令代码）的访问效率。

下面这段代码摘自 MSVC 的文件 listing.inc。

顺便提一下，这都是 NOP 指令的变种。虽然这些指令没有实际的操作意义，但是它们可以占用不同的空间。

出于 CPU 性能的考虑，下述代码没有使用多条 NOP 指令，而是使用了单条指令。

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructivenops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
 if size eq 3
   ; lea ecx, [ecx+00]
   DB 8DH, 49H, 00H
 else
  if size eq 4
   ; lea esp, [esp+00]
   DB 8DH, 64H, 24H, 00H
  else
   if size eq 5
     add eax, DWORD PTR 0
   else
    if size eq 6
     ; lea ebx, [ebx+00000000]
     DB 8DH, 9BH, 00H, 00H, 00H, 00H
    else
     if size eq 7
       ; lea esp, [esp+00000000]
       DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
      else
       if size eq 8
        ; jmp .+8; .npad 6
        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
       else
       if size eq 9
        ; jmp .+9; .npad 7
        DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
       else
        if size eq 10
         ; jmp .+A; .npad 7; .npad 1
```

```
            DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
           else
            if size eq 11
             ; jmp .+B; .npad 7; .npad 2
             DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
            else
             if size eq 12
              ; jmp .+C; .npad 7; .npad 3
              DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
             else
              if size eq 13
               ; jmp .+D; .npad 7; .npad 4
               DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24H, 00H
              else
               if size eq 14
               ; jmp .+E; .npad 7; .npad 5
               DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, 00H, 00H, 00H
              else
               if size eq 15
                ; jmp .+F; .npad 7; .npad 6
                DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, 00H, 00H, 00H, 00H
               else
                %out error: unsupported npad size
                .err
               endif
              endif
             endif
            endif
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endm
```

第 89 章　修改可执行文件

89.1　文本字符串

除了那些经过加密存储的字符串以外，我们可以使用任何一款的 hex editor 直接编辑 C 字符串。即使那些不了解机器码和可执行文件具体格式的人，也可以使用这项技术直接编辑可执行程序中的字符串。修改后的字符串，其长度不得大于原来字符串的长度，否则可能覆盖其他的数据甚至是其他指令。在 MS-DOS 盛行的时代，人们普遍使用这种方式直接用译文替换软件中的外文文字。至少在 20 世纪 80 年代和 90 年代的前苏联，这种技术十分流行。所以，那个时代也出现了各种古灵精怪的超短缩写：预定长度的字符串存储空间可能容纳不下完整的译文，所以软件翻译人员不得不绞尽脑汁压缩译文的长度。

在修改 Delphi 程序的字符串时，有时还要调整字符串的长度。

89.2　x86 指令

修改可执行文件中汇编指令的方式有以下几种：

- 禁用某些指令。此时只要使用 0x90（NOP）替换相应的汇编指令即可。
- 禁用条件转移指令。在修改 74 xx（JZ）这样的条件转移指令时，我们可以直接把转移指令的 2 个字节替换为两个 0x90（NOP），也可以把第二个字节（jump offset）替换为 0，即把偏移量固定为 0。
- 强制程序进行跳转。有些时候，我们需要把条件转移指令替换为跳转指令，强制其进行跳转。此时，把 opcde 的第一个字节替换为 JMP 的 0xEB 即可。
- 禁用某函数。只要把函数的第一个指令替换为 RETN（0xC3），那么它就不会运行。只要程序的调用约定不是 stdcall（第 64 章第 2 节），那么这种修改方法都不会有问题。在修改遵循 stdcall 约定的函数时，修改人员首先要注意函数参数的数量（可查阅原函数的 RETN 指令），然后使用带有 16 位参数的 RETN（0xC2）指令替换函数的第一条指令。
- 某些情况下，被禁用的函数必须返回 0 或 1。此时可使用“MOV EAX，0”或“MOV EAX，1”进行处理。直接使用这两条指令的 opcode 进行替换时，您会发现其 opcode 较长。这种情况下就可以使用“XOR EAX，EAX”（0x31 0xC0 两个字节）或 XOR EAX，EAX /INC EAX （0x31 0xC0 0x40 三个字节）进行替换。

很多软件采用了防范修改的技术。这种功能通常都由“读取可执行文件代码”和“校验和（checksum）检验”两个步骤分步实现。即是说，要实现防修改机制，程序首先要读取（加载到内存里的）程序文件。我们可设置断点，解析其读取内存函数的具体地址。

tracer 工具可以满足这种调试需求。它具有 BPM（内存断点）功能。

在修改程序时，不得修改 PE 可执行文件的 relocs（参见本书的 68.2.6 节）。Windows 的加载程序会使用新的代码覆盖这部分代码。如果使用 Hiew 打开可执行程序，会发现这部分代码以灰色显示（请参考图 7.12）。万不得已的时候，您可使用跳转指令绕过 reclos，否则就要编辑 relocs 的数据表。

第 90 章　编译器内部函数

编译器内部函数（compiler intrinsic）是与特定编译器有关的函数，并非寻常的库函数。在编译库函数时，编译器会调用（call）这个函数；而在编译内部函数时，编译器会使用对应的机器码进行直译。内部函数通常是与特定 CPU 特定指令集有关的伪函数。

例如，C/C++语言里没有循环移位运算指令，而多数 CPU 硬件支持这种指令。为了便于编程人员使用这种指令，MSVC 推出了有关的伪函数_rotl() 和_rotr()。在编译这两个函数时，编译器会直接使用 x86 指令集中 ROL/ROR 指令的 opcode 进行替换。

此外，为了方便程序代码调用 SSE 指令，MSVC 还推出了一些内部函数。

如需查询所有的 MSVC 内部函数，请查阅 MSDN 网站：http://msdn.microsoft.com/en-us/library/26td21ds.aspx。

第 91 章　编译器的智能短板

Intel C++ 10.1（在 Linux x86 平台上编译 Oracle RDBMS 11.2 的编译器）有时候会生成两个连续的 JZ 指令。实际上第二条 JZ 指令不会被执行、没有实际意义。

指令清单 91.1　kdli.o from libserver11.a

```
.text:08114CF1                  loc_8114CF1: ; CODE XREF: __PGOSF539_kdlimemSer+89A
.text:08114CF1                               ; __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08             mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14          movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01             test    dl, 1
.text:08114CFB 0F 85 17 08 00 00    jnz     loc_8115518
.text:08114D01 85 C9                test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00    jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00    jz      loc_8115518
.text:08114D0F 8B 53 08             mov     edx, [ebx+8]
.text:08114D12 89 55 FC             mov     [ebp+var_4], edx
.text:08114D15 31 C0                xor     eax, eax
.text:08114D17 89 45 F4             mov     [ebp+var_C], eax
.text:08114D1A 50                   push    eax
.text:08114D1B 52                   push    edx
.text:08114D1C E8 03 54 00 00       call    esp, 8
.text:08114D21 83 C4 08             add     esp, 8
```

上述文件中，另有一处也存在这种问题。

指令清单 91.2　from the same code

```
.text:0811A2A5                  loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5                               ; kdliSerLengths+1C1
.text:0811A2A5 8B 7D 08             mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10             mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14          movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01             test    dl, 1
.text:0811A2B2 75 3E                jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01             and     eax, 1
.text:0811A2B7 74 1F                jz      short loc_811A2D8
.text:0811A2B9 74 37                jz      short loc_811A2F2
.text:0811A2BB 6A 00                push    0
.text:0811A2BD FF 71 08             push    dwordptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF       call    len2nbytes
```

这些问题可能属于编译器的 bug。但是它们生成的程序不会受到该 bug 的影响，所以可能被测试人员遗漏了下来。本书的 19.2.4 节、39.3 节、47.7 节、18.7 节、12.4.1 节、19.5.2 节中都演示了这种问题。

本文演示了这些编译器问题，以证明编译器确实可能出现匪夷所思的奇怪行为。如果遇到了这种现象，读者不必绞尽脑汁地去琢磨“编译器为什么生成这种诡异代码”。

第 92 章　OpenMP

OpenMP 是一种相对简单的、实现多线程并发功能的编程 API。

本章将以加密学意义上的非重复随机数 nonce 为例，演示 OpenMP 的应用方法。下面这段代码把 nonce 和不加密的明文进行串联（即添加），以进行非可逆加密，从而增加截获、破解密文的难度。此外，Bitcion 协议约定，在某个阶段中通过 nonce 使消息块的 hash 包含特定长度的、连续的零。这种机制又叫作“prove of system”（系统验证）（https://en.wikipedia.org/wiki/Proof-of-work_system），即参与通信的系统通过这种机制证明它已经采取了精密而耗时的计算。

虽然下面这段代码和 Bitcoin 没有直接关系，但是功能颇为类似。它会向字符串“hello，word!_”添加一个数字，使得“hello，word!_<数字>”的 SHA512 hash 包含三个或三个以上的 0 字节。

假设穷举的区间为［0，INT32 最大数−1］（即 0～0x7FFFFFFE/2147483646）。

整个算法并不复杂：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
        uint8_t buf[32];
        struct sha512_ctx ctx;
        uint8_t res[64];

        // update statistics
        int t=omp_get_thread_num();

        if (__min[t]==-1)
                __min[t]=nonce;
        if (__max[t]==-1)
                __max[t]=nonce;

        __min[t]=min(__min[t], nonce);
        __max[t]=max(__max[t], nonce);

        // idle if valid nonce found
        if (found)
                return;
```

```
        memset (buf, 0, sizeof(buf));
        sprintf (buf, "hello, world!_%d", nonce);

        sha512_init_ctx (&ctx);
        sha512_process_bytes (buf, strlen(buf), &ctx);
        sha512_finish_ctx (&ctx, &res);
        if (res[0]==0 && res[1]==0 && res[2]==0)
        {
                printf ("found (thread %d): [%s]. seconds spent=%d\n", t, buf, time(NULL)-start);
                found=1;
        };
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
};

int main()
{
        int32_t i;
        int threads=omp_get_max_threads();
        printf ("threads=%d\n", threads);

        __min=(int32_t*)malloc(threads*sizeof(int32_t));
        __max=(int32_t*)malloc(threads*sizeof(int32_t));
        for (i=0; i<threads; i++)
                __min[i]=__max[i]=-1;

        start=time(NULL);

        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);

        for (i=0; i<threads; i++)
                printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], i, __max[i]);

        free(__min); free(__max);
};
```

函数 check_nonce() 有 3 个作用：向字符串添加数字、使用 SHA512 算法计算新字符串的 hash、检查 hash 中是否有 3 个为 0 的字节。

这段代码中较为重要的部分是：

```
        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);
```

这个程序确实不复杂。如果没有#pragma，程序会从 0 依次穷举到 INT32 的最大值（0x7fffffff，即 2147483647），依次用 check_nonce() 函数验证。加上#pragma 之后，编译器会添加特定的代码把整个区间划分为若干子区间，充分利用 CPU 的多核进行并行运算。[①]

我们可以通过下述指令，使用 MSVC 2012 进行编译[②]：

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

GCC 对应的编译指令为：

① 本例仅为示范性说明。在实际情况下，使用 OpenMP 技术往往更为困难、复杂。

② sha512.（c|h）和 u64.h 的源文件可照搬 OpenSSL 的库文件：http://www.openssl.org/source/。

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

92.1 MSVC

MSVC 2012 生成的主循环的指令如下所示。

指令清单 92.1 MSVC 2012

```
        push    OFFSET _main$omp$1
        push    0
        push    1
        call    __vcomp_fork
        add     esp, 16                         ; 00000010H
```

所有以 vcomp 开头的函数都是与 OpenMP 有关的函数，通过 vcomp*.dll 进行加载。它将发起一组线程进行并行计算。

具体来说，_mainomp1 的汇编代码如下所示。

指令清单 92.2 MSVC 2012

```
$T1 = -8                                                ; size = 4
$T2 = -4                                                ; size = 4
_main$omp$1 PROC                                        ; COMDAT
        push    ebp
        mov     ebp, esp
        push    ecx
        push    ecx
        push    esi
        lea     eax, DWORD PTR $T2[ebp]
        push    eax
        lea     eax, DWORD PTR $T1[ebp]
        push    eax
        push    1
        push    1
        push    2147483646                              ; 7ffffffeH
        push    0
        call    __vcomp_for_static_simple_init
        mov     esi, DWORD PTR $T1[ebp]
        add     esp, 24                                 ; 00000018H
        jmp     SHORT $LN6@main$omp$1
$LL2@main$omp$1:
        push    esi
        call    _check_nonce
        pop     ecx
        inc     esi
$LN6@main$omp$1:
        cmp     esi, DWORD PTR $T2[ebp]
        jle     SHORT $LL2@main$omp$1
        call    __vcomp_for_static_end
        pop     esi
        leave
        ret     0
_main$omp$1 ENDP
```

这个函数会启动 *n* 个并发线程，其中 *n* 就是 CPU 核芯（cores）的总数。函数 vcomp_for_static_simple_init() 计算当前线程里 for() 结构体的区间，而区间的间隔则由当前线程的总数决定。循环计数器的起始值和结束值分别存储于局部变量$T1 和$T2。细心的读者可能注意到函数 vcomp_for_static_simple_init() 的一个参数为 0x7ffffffeh（即 2147483646）——它是整个循环体的迭代次数，最终会被 *n* 整除。

接下来程序发起了调用函数 check_nonce() 的循环，完成余下的工作。

我在源代码的 check_nonce() 函数中有意添加了统计代码，用来统计参数被调用的次数。

整个程序的运行结果如下：

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7ffffffe
```

计算结果的前 3 个字节确实是零：

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 *test
```

在笔者的 4 核 Intel Xeon E3-1220 3.10Ghz CPU 上运行这个程序，总耗时大约 2～3 秒。我们可以在任务管理器中看到 5 个线程：1 个主线程和 4 个子线程。虽然理论上它还有精简的空间，但是我没有对程序源代码进行深度优化。深度优化应当可以大幅度提升它的运行效率。另外，因为我的 CPU 有 4 个内核，所以它发起了 4 个子线程——这完全符合 OpenMP 规范。

通过程序输出的统计数据，我们可以清楚地观察到整个穷举空间被划分为大致相等的四个部分。严格地说，考虑到最后一个比特位，这四个区间确实并非完全相等。

OpenMP 还提供了保障模块 crtomic（原子性）的 Prugms 指令。顾名思义，被标记为原子性的代码不会被拆分为多个子线程运行。我们来看看这段代码：

```
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
```

经 MSVC 2012 编译，上述代码对应的汇编指令如下所示。

指令清单 92.3　MSVC 2012

```
        push    edi
        push    OFFSET _checked
        call    __vcomp_atomic_add_i4
; Line 55
        push    OFFSET _$vcomp$critsect$
        call    __vcomp_enter_critsect
        add     esp, 12                                 ; 0000000cH
; Line 56
        mov     ecx, DWORD PTR _checked
        mov     eax, ecx
        cdq
        mov     esi, 100000                             ; 000186a0H
        idiv    esi
        test    edx, edx
        jne     SHORT $LN1@check_nonc
; Line 57
        push    ecx
        push    OFFSET ??_C@_0M@NPNHLIOO@checked?$DN?$CFd?6?$AA@
        call    _printf
        pop     ecx
        pop     ecx
$LN1@check_nonc:
```

```
        push    DWORD PTR _$vcomp$critsect$
        call    __vcomp_leave_critsect
        pop     ecx
```

封装在 vcomp*.dll 里的函数 vcomp_atomic_add_i4() 只是一个使用了 LOCK XADD 指令的小函数。[①] 函数 vcomp_enter_critsect() 调用的是 Win32 API 函数 EnterCriticalSection()[②]。

92.2 GCC

经 GCC 4.8.1 生成的程序，其统计结果和上面的程序一样。所以 GCC 分割区间的方法与 MSVC 相同。

指令清单 92.4 GCC 4.8.1

```
        mov     edi, OFFSET FLAT:main._omp_fn.0
        call    GOMP_parallel_start
        mov     edi, 0
        call    main._omp_fn.0
        call    GOMP_parallel_end
```

由 GCC 编译生成的程序，会发起 3 个新的线程，原有线程扮演第 4 进程的角色。所以，总体上 GCC 的进程数是 4，MSVC 的进程数是 5。

其中，函数 main._omp_fn.0 的代码如下所示。

指令清单 92.5 GCC 4.8.1

```
main._omp_fn.0:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 40
        mov     QWORD PTR [rbp-40], rdi
        call    omp_get_num_threads
        mov     ebx, eax
        call    omp_get_thread_num
        mov     esi, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     ecx, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     eax, edx
        cmp     esi, eax
        jl      .L15
.L18:
        imul    esi, ecx
        mov     edx, esi
        add     eax, edx
        lea     ebx, [rax+rcx]
        cmp     eax, ebx
        jge     .L14
        mov     DWORD PTR [rbp-20], eax
.L17:
        mov     eax, DWORD PTR [rbp-20]
        mov     edi, eax
        call    check_nonce
        add     DWORD PTR [rbp-20], 1
        cmp     DWORD PTR [rbp-20], ebx
```

① 有关 LOCK 前缀的详细说明，请参见本书附录 A.6.1。
② 请参见本书 68.4 节。

```
        jl      .L17
        jmp     .L14
.L15:
        mov     eax, 0
        add     ecx, 1
        jmp     .L18
.L14:
        add     rsp, 40
        pop     rbx
        pop     rbp
        ret
```

上述指令清晰地显示出：程序通过调用函数 omp_get_num_threads() 和另一个函数 omp_get_thread_num() 获取当前线程的总数以及当前线程的编号，然后分割循环体。之后，它再运行 check_nonce()。

GCC 在代码中直接使用 LOCK ADD 指令，而 MSVC 则是调用另一个 DLL 文件中的独立函数。

指令清单 92.6 GCC 4.8.1

```
        lock add        DWORD PTR checked[rip], 1
        call    GOMP_critical_start
        mov     ecx, DWORD PTR checked[rip]
        mov     edx, 351843721
        mov     eax, ecx
        imul    edx
        sar     edx, 13
        mov     eax, ecx
        sar     eax, 31
        sub     edx, eax
        mov     eax, edx
        imul    eax, eax, 100000
        sub     ecx, eax
        mov     eax, ecx
        test    eax, eax
        jne     .L7
        mov     eax, DWORD PTR checked[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
        mov     eax, 0
        call    printf
.L7:
        call    GOMP_critical_end
```

以 GOMP 开头的函数来自于 GNU OpenMP Library。您可在 GitHub 下载到它的源文件（https://github.com/gcc-mirror/gcc/tree/master/libgomp）。不过，微软的 vcomp*.dll 文件没有源代码可查。

第93章 安腾指令

就市场来讲，安腾（Itanium）处理器几乎是失败产品。但是它的 Intel Itanium（IA64）架构非常值得研究。乱序执行（OOE）CPU 理念，侧重于让 CPU 重新划分指令的片段和顺序，再把重组后的指令组分派到并联计算单位进行并行计算。而英特尔（Intel）安腾架构推出的并行计算技术（Explicitly Parallel Instruction Code，EPIC），则主张让编译器在编译的早期阶段实现指令分组。

厂商推出了配合这种并行计算技术的编译器。不过，这些编译器因异常复杂而颇受争议。

本章从 Linux 内核（3.2.0.4）摘录了部分 IA64 指令。这段程序用于实现某加密机制。其源代码如下所示。

指令清单 93.1 Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
        structtea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];

        n = TEA_ROUNDS;

        while (n-- > 0) {
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

其编译结果如下所示。

指令清单 93.2 Linux Kernel 3.2.0.4 for Itanium 2（McKinley）

```
0090|                                   tea_encrypt:
0090|08 80 80 41 00 21                      adds r16 = 96, r32          // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00                      adds r8 = 88, r32           // ptr to ctx->KEY[0]
009C|00 00 04 00                            nop.i 0
00A0|09 18 70 41 00 21                      adds r3 = 92, r32           // ptr to ctx->KEY[1]
00A6|F0 20 88 20 28 00                      ld4 r15 = [r34], 4          // load z
00AC|44 06 01 84                            adds r32 = 100, r32;;       // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10                      ld4 r19 = [r16]             // r19=k2
00B6|00 01 00 00 42 40                      mov r16=r0                  // r0一直是0
00BC|00 08 CA 00                            mov.i r2 = ar.lc            // 保存lc
```

```
00C0|05 70 00 44 10 10 9E FF FF FF 7F 20   ld4 r14 = [r34]                // load y
00CC|92 F3 CE 6B                           movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00                     nop.m 0
00D6|50 01 20 20 20 00                     ld4 r21 = [r8]                // r21=k0
00DC|F0 09 2A 00                           mov.i ar.lc = 31              //TEA_ROUNDS=32
00E0|0A A0 00 06 10 10                     ld4 r20 = [r3];;              // r20=k1
00E6|20 01 80 20 20 00                     ld4 r18 = [r32]               // r18=k3
00EC|00 00 04 00                           nop.i 0
00F0|
00F0|                                  loc_F0:
00F0|09 80 40 22 00 20                     add r16 = r16, r17            //r16=sum, r17=TEA_DELTA
00F6|D0 71 54 26 40 80                     shladd r29 = r14, 4, r21      // r14=y, r21=k0
00FC|A3 70 68 52                           extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20                     add r30 = r16, r14
0106|B0 E1 50 00 40 40                     add r27 = r28, r20;;          // r20=k1
010C|D3 F1 3C 80                           xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20                     xor r25 = r27, r26;;
0116|F0 78 64 00 40 00                     add r15 = r15, r25            // r15=z
011C|00 00 04 00                           nop.i 0;;
0120|00 00 00 00 01 00                     nop.m 0
0126|80 51 3C 34 29 60                     extr.u r24 = r15, 5, 27
012C|F1 98 4C 80                           shladd r11 = r15, 4, r19      // r19=k2
0130|0B B8 3C 20 00 20                     add r23 = r15, r16;;
0136|A0 C0 48 00 40 00                     add r10 = r24, r18            // r18=k3
013C|00 00 04 00                           nop.i 0;;
0140|0B 48 28 16 0F 20                     xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00                     xor r22 = r23, r9
014C|00 00 04 00                           nop.i 0;;
0150|11 00 00 00 01 00                     nop.m 0
0156|E0 70 58 00 40 A0                     add r14 = r14, r22
015C|A0 FF FF 48                           br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15                     st4 [r33] = r15, 4            // store z
0166|00 00 00 02 00 00                     nop.m 0
016C|20 08 AA 00                           mov.i ar.lc = r2;;            // restore lc legister
0170|11 00 38 42 90 11                     st4 [r33] = r14               // store y
0176|00 00 00 02 00 80                     nop.i 0
017C|08 00 84 00                           br.ret.sptk.many b0;;
```

上述 IA64 指令很有特点。

首先，每 3 条指令构成一个指令字（instruction bundles）。每个指令字的长度都是 16 字节即 128 位，由 1 个 5 位的模版字段和 3 个 41 位微操作指令构成。IDA 把这些指令组分为（6+6+4）字节的结构体，以便于调试人员进行区分。

除了含有停止位（stop bit）的指令之外，这些由 3 条微操作指令构成的指令字，通常都由 CPU 并行处理。

据称，Intel 和 HP 的开发人员针对常见指令进行了模式划分，从而推出了指令字类型（即指令模版）的概念——声明指令字运算资源的模版字段。CPU 依此把指令字区分为 12 种基本类型（basic bundle types），基本类型又分为带停止位的版本和不带停止位的版本。举例来说，第 0 类指令字叫作 MII 类指令字，依次由内存读写微操作指令（M）和两条整数运算的微操作指令（II）构成；最后一类指令，即 0x1d 类指令字，又叫作 MFB 类指令字，依次由内存读写微操作指令（M）、浮点数运算微操作指令（F）、分支（转移）微操作指令（B）构成。

如果编译器在指令字的指令位（instruction slot）上编排不了相应的微操作指令，那么它可能在这些指令位上安插空操作指令 nop。您可能注意到了，本文中的 nop 指令分为“nop，i”和“nop，m”。i 代表该 nop 指令占用整数（integer）处理单元，属于整数运算型微操作指令；m 代表它占用内存处理单元，属于内存操作型微操作指令。在人工编写汇编语言时，编辑程序会自动插入相应的 nop 指令。

IA64 汇编指令的特性不止这些。该平台的指令字还可进行分组，构成指令组（instruction group）。指令组可由任意个连续运算的指令字和一个含有停止位的指令字构成，是一个可并行执行的指令集合。在实

际应用中，安腾 2 处理器可以并行执行 2 路指令字，即同时处理 6 个微操作指令。

这就要求指令字中的各微操作指令和每个指令组的各指令字之间互不干扰，即不存在数据竞争。如果存在数据竞争、形成脏数据，那么运算结果不可控（undefined）。

在 IDA 中，微操作指令之后的两个分号（;;）表示该指令有停止位。可见，[90-ac] 及 [b0-bc] 分别属于可并行执行的两个指令组，它们之间不存在互扰。下一组则是 [b0-cc]。

另外，在 10c 处的指令，以及下一条位于 110 处的指令都有停止位。这就意味着 CPU 会在与其他指令隔绝的情况下运行这两个指令，这种运行模式就和常规的 CISC/复杂指令集的执行方式完全相同了。这是由于后续指令，即 110 处的指令需要前一条指令的运行结果（R26）寄存器，所以不能并行处理这两条指令。很明显，此时编译器不能找到更好的并行处理手段，无法更有效地利用 CPU，所以在此添加了 2 个停止位和多个 NOP 指令。虽说编译器在智能方面很不成熟，但是人工的 IA64 汇编编程也丝毫不轻松：程序员要手动完成指令字分组的工作。

要图省事的话，程序员可以给每条指令添加停止位，不过这将大幅度地浪费安腾处理器的运算性能。

Linux 内核的源代码中，就有一些经典的、手写 IA64 汇编代码。有兴趣的读者可参考：http://lxr.free-electrons.com/source/arch/ia64/lib/。

有关 IA64 汇编语言人工编程的具体方法，可参见 Mike Burrell 撰写的专业论著《Writing Effcient Itanium 2 Assembly Code》（http://yurichev.com/mirrors/RE/itanium.pdf）。有关汇编语言指令字的详细说明，请参见 Phrack Itanium 的帖子 http://phrack.org/issues/57/5.html。

第 94 章　8086 的寻址方式

在 MS-DOS 及 Win16（参见本书 78.3 节和 53.5 节）平台的 16 位应用程序中，我们可以看到程序指针由两个 16 位值构成。这是什么情况？不得不说这是 MS-DOS 和 8086 的另一大怪异的特色。

虽然 8086/8088 属于 16 位 CPU，但是它却有 20 位的 RAM 地址空间（内存总线有 20 个引脚）；即，它可直接寻址的存储空间只有 1MB。这 1MB 的外部内存空间又被划分为 RAM（最大 640kB）、ROM、显卡内存、EMS 卡，等等。

16 位的 8086/8088 CPU 实际上由 8080 CPU 发展而来。8080 CPU 的地址空间只有 16 位，所以可直接控制的内存只有 64KB。大概是 8086 的设计者认为 64KB 空间不够用，而且 8086 还要兼容 8080 平台的程序，所以就把 20 位/1MB 的内存划分为若干个段使用。这就是早期玩具级的虚拟内存技术的思路。而 8086 的寄存器又只是 16 位寄存器，为了进行更大范围（20 位）的寻址，它就得借助新推出的段寄存器。从此 CPU 就有了 CS、DS、ES 和 SS 寄存器。20 位的内存指针由短寄存器和地址寄存器对（DS:BX）混合计算而来：

$$real_address = (segment_register \ll 4) + address_register$$

举例来说，过去 IBM PC 兼容的主机，其显卡（EGA、VGA）的显存都只有 64KB。要读写显存，就要在某个段寄存器里（例如 DS）写入 0xA000。如此一来，程序就可使用 DS:0~DS:0xFFFF 访问整个显存。虽然地址总线是 20 位的，超过了 16 位寄存器的表达范围，但是 CPU 可借助段寄存器毫无障碍地访问 0xA0000～0AFFFF。

程序还可能直接访问固定的内存地址，例如 0x1234，但是操作系统加载应用程序到起始地址却不是固定的。段寄存器的出现，解决了这种问题——它可由段寄存器进行相对寻址，应用程序不必关心自己到底被加载到了什么 RAM 地址上。

实际上，MS-DOS 系统下的指针由段地址和段内地址构成，可由两个 16 位的数值表示。20 位地址总线足以满足这种寻址方式的需要。不过，程序员就需要重新计算内存地址了：他们要不停地考虑空间和效率的平衡，仔细规划数据栈的分配情况。

另外，8086 的寻址方式决定了每个内存块不能大于 64KB。

80286 平台仍然继承了段寄存器（segment registers），只是用途不同而已。

在支持更大 RAM 的 80386 CPU 问世时，市面上流行的仍然是 MS-DOS。更有一大批叫作 DOS extenders 的 DOS 粉丝在 Windows 系统问世以后继续坚守 DOS 阵地。他们甚至开发出各种像模像样的 OS 系统。这种系统不仅实现了 CPU 保护模式的切换功能，而且大幅度地改进了内存 API，可继续支持 MS-DOS 的应用程序。著名的有：DOS/4GW（游戏巨作 DOOM 就是面向它编译的）、Phar Lap 和 PMODE。

在 Win32 之前，16 位的 Windows 3.x 仍然沿用了这种寻址方式。

第 95 章　基本块重排

95.1　PGO 的优化方式

PGO 是 Profile-guided optimization 的缩写，中文有“配置文件引导的优化”等译法。经 PGO 方式优化以后，程序中的某些基本块（basic block）（所谓基本块，指的是程序里顺序执行的语句序列。基本块由第一个语句构成入口，由最后一个语句构成出口。在执行程序时，不可从入口以外进入该基本块（被跳入），也不可从出口以外的地址跳出该基本块。）可能会被调整到可执行文件的任意位置。

很明显，函数中的程序代码存在执行频率的差异。例如，循环语句一类代码的执行频率必然很高，而错误报告、异常处理之类代码的执行频率较低。

在使用 PGO 时，编译器首先会生成一种可记录运行细节的特殊程序。而后，研发人员通过试运行的手段收集该程序的各项统计信息。最后，编译器根据这些统计信息对可执行文件进行调整和优化，把执行几率较小的基本块挪到其他地方。

在经 PGO 优化后的程序里，频繁执行的函数代码会被调整得更为紧凑。PGO 优化了条件跳转的性能，提高了 CPU 分支预测的准确率。这些特性均有助于提升程序性能。

Oracle 是由 Intel C++编译器生成的程序。本文收录了 Oracle 中 orageneric11.dll（Win32）的部分代码。

指令清单 95.1　orageneric11.dll（Win32）

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue:
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error               ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...

; address 0x60B953F0
```

```
__VInfreq__skgfsync:
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ; "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue
error:
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function called with invalid FIB/IOV structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

上述两个基本块的地址相距 9MB 左右。

在这个文件中，所有的不常用函数都位于 DLL 文件的尾部。这部分不常用函数都被 Intel C++编译器打上了 VInfreq 前缀。例如，我们看到函数尾部的部分代码用于记录 log 文件（大概用于错误、警告和异常处理）。因为 Oracle 开发人员在试运行期间收集统计信息时，它的执行概率较低（甚至没被执行过），所以它们被标注上了__VInfreq 前缀。最终，这个日志基本块把控制流返回给位于“热门地区”的函数代码。

程序里另外一处“不常用”的区间是返回错误代码 27050 的基本块。

在 Linux ELF 环境下，Intel C++编译器会在 ELF 文件里通过.hot/.unlikely 标记“热门”/“冷门”基本块。

以逆向工程的角度来看，这些信息可用来辨别函数的核心部分和异常处理部分。

第十一部分

推荐阅读



第96章 参考书籍

96.1 Windows

Mark E. Russinovich、David A. Solomon 与 Alex Ionescu 合著的《Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition》2009。

96.2 C/C++

《ISO/IEC 14882:2011 (C++ 11 standard)》。

此外，读者可参见 http://go.yurichev.com/17275（2013）。

96.3 x86/x86-64

Intel 出版的《Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes》，其中的 1，2A，2B，2C，3A，3B 和 3C 章。本书作者将其收录为 http://go.yurichev.com/17283（2013）。

AMD 出版的《AMD64 Architecture Programmer's Manual》。本书作者将其收录为 http://go.yurichev.com/17284（2013）。

96.4 ARM

请参见本书作者收集的 ARM 手册：http://go.yurichev.com/17024。

96.5 加密学

Bruce Schneier 撰写的《Applied Cryptography: Protocols, Algorithms, and Source Code in C》（1994 年）。

第 97 章　博　　客

97.1　Windows 平台

- 微软：Raymond Chen (http://blogs.msdn.com/b/oldnewthing/)。
- nynaeve.net (http://www.nynaeve.net/)。

第98章 其他内容

reddit.com有两个非常出色的逆向工程相关板块，请参见：

- Reverse Engineering（http://www.reddit.com/r/ReverseEngineering/）。
- REMath逆向工程与数学的综合板块（http://www.reddit.com/r/remath）。

Stack Exchange网站同样有一个著名的逆向工程板块：

- http://reverseengineering.stackexchange.com/。

FreeNode（IRC）的#re频道是专门讨论逆向工程的主题聊天室。

第十二部分

练习题

除非文中有单独的提问，否则本卷题目的默认问题都是：

- 请用一句话描述这个程序的功能。
- 请把这个函数还原为 C/C++语言的源程序。

在解答题目时，您可以通过 Google 等搜索引擎查找线索。但是，不借助搜索引擎的乐趣会更多一些。另外，您还可以在本书的附录里查找相关提示。



第 99 章　初等难度练习题

这种难度的题目通常可以直接心算。

99.1　练习题 1.4

下列程序使用了密码保护机制，请找到程序指定的密码。

喜欢举一反三的读者，还可以修改可执行程序来改变程序的密码。在修改密码的时候，建议您同时调整密码的长度，并摸索最短密码到底可以有多短。

此外，单独一个字符串就可以令程序崩溃。请创建这种字符串。

- Win32（go.yurichev.com/17166）。
- Linux x86（go.yurichev.com/17167）。
- Mac OS X（go.yurichev.com/17168）。
- MIPS（go.yurichev.com/17169）。

第 100 章　中等难度练习题

要解答这个难度的题目，您可能会用到文本编辑器或者纸笔。

100.1　练习题 2.1

100.1.1　Optimizing MSVC 2010 x86

```
__real@3fe0000000000000 DQ 03fe0000000000000r
__real@3f50624dd2f1a9fc DQ 03f50624dd2f1a9fcr

_g$  = 8
tv132 = 16
_x$ = 16
f1 PROC
        fld     QWORD PTR _x$[esp-4]
        fld     QWORD PTR __real@3f50624dd2f1a9fc
        fld     QWORD PTR __real@3fe0000000000000
        fld     QWORD PTR _g$[esp-4]
$LN2@f1:
        fld     ST(0)
        fmul    ST(0), ST(1)
        fsub    ST(0), ST(4)
        call    __ftol2_sse
        cdq
        xor     eax, edx
        sub     eax, edx
        mov     DWORD PTR tv132[esp-4], eax
        fild    DWORD PTR tv132[esp-4]
        fcomp   ST(3)
        fnstsw  ax
        test    ah, 5
        jnp     SHORT $LN19@f1
        fld     ST(3)
        fdiv    ST(0), ST(1)
        faddp   ST(1), ST(0)
        fmul    ST(0), ST(1)
        jmp     SHORT $LN2@f1
$LN19@f1:
        fstp    ST(3)
        fstp    ST(1)
        fstp    ST(0)
        ret     0
f1 ENDP

__real@3ff0000000000000 DQ 03ff0000000000000r

_x$ = 8
f2 PROC
        fld     QWORD PTR _x$[esp-4]
        sub     esp, 16
        fstp    QWORD PTR [esp+8]
        fld1
        fstp    QWORD PTR [esp]
        call    f1
```

```
        add     esp, 16
        ret     0
f2 ENDP
```

100.1.2 Optimizing MSVC 2012 x64

```
__real@3fe0000000000000 DQ 03fe0000000000000r
__real@3f50624dd2f1a9fc DQ 03f50624dd2f1a9fcr
__real@3ff0000000000000 DQ 03ff0000000000000r

x$ = 8
f       PROC
        movsdx  xmm2, QWORD PTR __real@3ff0000000000000
        movsdx  xmm5, QWORD PTR __real@3f50624dd2f1a9fc
        movsdx  xmm4, QWORD PTR __real@3fe0000000000000
        movapd  xmm3, xmm0
        npad    4
$LL4@f:
        movapd  xmm1, xmm2
        mulsd   xmm1, xmm2
        subsd   xmm1, xmm3
        cvttsd2si eax, xmm1
        cdq
        xor     eax, edx
        sub     eax, edx
        movd    xmm0, eax
        cvtdq2pd xmm0, xmm0
        comisd  xmm5, xmm0
        ja      SHORT $LN18@f
        movapd  xmm0, xmm3
        divsd   xmm0, xmm2
        addsd   xmm0, xmm2
        movapd  xmm2, xmm0
        mulsd   xmm2, xmm4
        jmp     SHORT $LL4@f
$LN18@f:
        movapd  xmm0, xmm2
        ret     0
f       ENDP
```

100.2 练习题 2.4

下面这道题目摘自 MSVC 2010，是标准的库函数。

100.2.1 Optimizing MSVC 2010

```
PUBLIC   _f
_TEXT   SEGMENT
_arg1$ = 8              ;size=4
_arg2$ = 12             ;size=4
_f   PROC
    push   esi
    mov    esi, DWORD PTR _arg1$[esp]
    push   edi
    mov    edi, DWORD PTR _arg2$[esp+4]
    cmp    BYTE PTR [edi], 0
    mov    eax, esi
    je     SHORT $LN7@f
    mov    dl, BYTE PTR [esi]
    push   ebx
    test   dl, dl
    je     SHORT $LN4@f
```

```
    sub     esi, edi
    npad    6 ; align next label
$LL5@f:
    mov     ecx, edi
    test    dl, dl
    je      SHORT $LN2@f
$LL3@f:
    mov     dl, BYTE PTR [ecx]
    test    dl, dl
    je      SHORT $LN14@f
    movsx   ebx, BYTE PTR [esi+ecx]
    movsx   edx, dl
    sub     ebx, edx
    jne     SHORT $LN2@f
    inc     ecx
    cmp     BYTE PTR [esi+ecx], bl
    jne     SHORT $LL3@f
$LN2@f:
    cmp     BYTE PTR [ecx], 0
    je      SHORT $LN14@f
    mov     dl, BYTE PTR [eax+1]
    inc     eax
    inc     esi
    test    dl, dl
    jne     SHORT $LL5@f
    xor     eax, eax
    pop     ebx
    pop     edi
    pop     esi
    ret     0
_f    ENDP
_TEXT    ENDS
END
```

100.2.2 GCC 4.4.1

```
                public f
f               proc near

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                mov     eax, [ebp+arg_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_8048443
                mov     eax, [ebp+arg_0]
                jmp     short locret_8048453

loc_80483F4:
                mov     eax, [ebp+var_4]
                mov     [ebp+var_8], eax
                mov     eax, [ebp+arg_4]
                mov     [ebp+var_C], eax
                jmp     short loc_804840A
```

```
loc_8048402:
                add     [ebp+var_8], 1
                add     [ebp+var_C], 1

loc_804840A:
                mov     eax, [ebp+var_8]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_8]
                movzx   edx, byte ptr [eax]
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                cmp     dl, al
                jz      short loc_8048402

loc_804842E:
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_804843D
                mov     eax, [ebp+var_4]
                jmp     short locret_8048453

loc_804843D:
                add     [ebp+var_4], 1
                jmp     short loc_8048444

loc_8048443:
                nop

loc_8048444:
                mov     eax, [ebp+var_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_80483F4
                mov     eax, 0

locret_8048453:
                leave
                retn
f               endp
```

100.2.3　Optimizing Keil (ARM mode)

```
        PUSH    {r4,lr}
        LDRB    r2,[r1,#0]
        CMP     r2,#0
        POPEQ   {r4,pc}
        B       |L0.80|
|L0.20|
        LDRB    r12,[r3,#0]
        CMP     r12,#0
        BEQ     |L0.64|
        LDRB    r4,[r2,#0]
        CMP     r4,#0
        POPEQ   {r4,pc}
        CMP     r12,r4
        ADDEQ   r3,r3,#1
        ADDEQ   r2,r2,#1
```

```
        BEQ     |L0.20|
        B       |L0.76|
|L0.64|
        LDRB    r2,[r2,#0]
        CMP     r2,#0
        POPEQ   {r4,pc}
|L0.76|
        ADD     r0,r0,#1
|L0.80|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        MOVNE   r3,r0
        MOVNE   r2,r1
        MOVEQ   r0,#0
        BNE     |L0.20|
        POP     {r4,pc}
```

100.2.4 Optimizing Keil (Thumb mode)

```
        PUSH    {r4,r5,lr}
        LDRB    r2,[r1,#0]
        CMP     r2,#0
        BEQ     |L0.54|
        B       |L0.46|
|L0.10|
        MOVS    r3,r0
        MOVS    r2,r1
        B       |L0.20|
|L0.16|
        ADDS    r3,r3,#1
        ADDS    r2,r2,#1
|L0.20|
        LDRB    r4,[r3,#0]
        CMP     r4,#0
        BEQ     |L0.38|
        LDRB    r5,[r2,#0]
        CMP     r5,#0
        BEQ     |L0.54|
        CMP     r4,r5
        BEQ     |L0.16|
        B       |L0.44|
|L0.38|
        LDRB    r2,[r2,#0]
        CMP     r2,#0
        BEQ     |L0.54|
|L0.44|
        ADDS    r0,r0,#1
|L0.46|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        BNE     |L0.10|
        MOVS    r0,#0
|L0.54|
        POP     {r4,r5,pc}
```

100.2.5 Optimizing GCC 4.9.1 (ARM64)

指令清单 100.1 Optimizing GCC 4.9.1 (ARM64)

```
func:
        ldrb    w6, [x1]
        mov     x2, x0
```

```
        cbz     w6, .L2
        ldrb    w2, [x0]
        cbz     w2, .L24
.L17:
        ldrb    w2, [x0]
        cbz     w2, .L5
        cmp     w6, w2
        mov     x5, x0
        mov     x2, x1
        beq     .L18
        b       .L5
.L4:
        ldrb    w4, [x2]
        cmp     w3, w4
        cbz     w4, .L8
        bne     .L8
.L18:
        ldrb    w3, [x5,1]!
        add     x2, x2, 1
        cbnz    w3, .L4
.L8:
        ldrb    w2, [x2]
        cbz     w2, .L27
.L5:
        ldrb    w2, [x0,1]!
        cbnz    w2, .L17
.L24:
        mov     x2, 0
.L2:
        mov     x0, x2
        ret
.L27:
        mov     x2, x0
        mov     x0, x2
        ret
```

100.2.6　Optimizing GCC 4.4.5 (MIPS)

指令清单 100.2　Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                lb      $v1, 0($a1)
                or      $at, $zero
                bnez    $v1, loc_18
                move    $v0, $a0
locret_10:                              # CODE XREF: f+50
                                        # f+78
                jr      $ra
                or      $at, $zero
loc_18:                                 # CODE XREF: f+8
                lb      $a0, 0($a0)
                or      $at, $zero
                beqz    $a0, locret_94
                move    $a2, $v0
loc_28:                                 # CODE XREF: f+8C
                lb      $a0, 0($a2)
                or      $at, $zero
                beqz    $a0, loc_80
                or      $at, $zero
                bne     $v1, $a0, loc_80
                move    $a3, $a1
                b       loc_60
                addiu   $a2, 1
loc_48:                                 # CODE XREF: f+68
                lb      $t1, 0($a3)
```

```
                or      $at, $zero
                beqz    $t1, locret_10
                or      $at, $zero
                bne     $t0, $t1, loc_80
                addiu   $a2, 1
loc_60:                                 # CODE XREF: f+40
                lb      $t0, 0($a2)
                or      $at, $zero
                bnez    $t0, loc_48
                addiu   $a3, 1
                lb      $a0, 0($a3)
                or      $at, $zero
                beqz    $a0, locret_10
                or      $at, $zero
loc_80:                                 # CODE XREF: f+30
                                        # f+38 ...
                addiu   $v0, 1
                lb      $a0, 0($v0)
                or      $at, $zero
                bnez    $a0, loc_28
                move    $a2, $v0
locret_94:                                 # CODE XREF: f+20
                jr      $ra
                move    $v0, $zero
```

100.3 练习题 2.6

100.3.1 Optimizing MSVC 2010

```
PUBLIC  _f
; Function compile flags: /Ogtpy
_TEXT   SEGMENT
_k0$ = -12              ;size = 4
_k3$ = -8               ;size = 4
_k2$ = -4               ;size = 4
_v$ = 8                 ;size = 4
_k1$ = 12               ;size = 4
_k$ = 12                ;size = 4
_f  PROC

    sub     esp, 12     ; 0000000cH
    mov     ecx, DWORD PTR _v$[esp+8]
    mov     eax, DWORD PTR [ecx]
    mov     ecx, DWORD PTR [ecx+4]
    push    ebx
    push    esi
    mov     esi, DWORD PTR _k$[esp+16]
    push    edi
    mov     edi, DWORD PTR [esi]
    mov     DWORD PTR _k0$[esp+24], edi
    mov     edi, DWORD PTR [esi+4]
    mov     DWORD PTR _k1$[esp+20], edi
    mov     edi, DWORD PTR [esi+8]
    mov     esi, DWORD PTR [esi+12]
    xor     edx, edx
    mov     DWORD PTR _k2$[esp+24], edi
    mov     DWORD PTR _k3$[esp+24], esi
    lea     edi, DWORD PTR [edx+32]
$LL8@f:
    mov     esi, ecx
    shr     esi, 5
```

```
    add     esi, DWORD PTR _k1$[esp+20]
    mov     ebx, ecx
    shl     ebx, 4
    add     ebx, DWORD PTR _k0$[esp+24]
    sub     edx, 1640531527   ; 61c88647H
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+ecx]
    xor     esi, ebx
    add     eax, esi
    mov     esi, eax
    shr     esi, 5
    add     esi, DWORD PTR _k3$[esp+24]
    mov     ebx, eax
    shl     ebx, 4
    add     ebx, DWORD PTR _k2$[esp+24]
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+eax]
    xor     esi, ebx
    add     ecx, esi
    dec     edi
    jne     SHORT $LL8@f
    mov     edx, DWORD PTR _v$[esp+20]
    pop     edi
    pop     esi
    mov     DWORD PTR [edx], eax
    mov     DWORD PTR [edx+4], ecx
    pop     ebx
    add     esp, 12             ; 0000000cH
    ret     0
_f    ENDP
```

100.3.2 Optimizing Keil (ARM mode)

```
        PUSH    {r4-r10,lr}
        ADD     r5,r1,#8
        LDM     r5,{r5,r7}
        LDR     r2,[r0,#4]
        LDR     r3,[r0,#0]
        LDR     r4,|L0.116|
        LDR     r6,[r1,#4]
        LDR     r8,[r1,#0]
        MOV     r12,#0
        MOV     r1,r12
|L0.40|
        ADD     r12,r12,r4
        ADD     r9,r8,r2,LSL #4
        ADD     r10,r2,r12
        EOR     r9,r9,r10
        ADD     r10,r6,r2,LSR #5
        EOR     r9,r9,r10
        ADD     r3,r3,r9
        ADD     r9,r5,r3,LSL #4
        ADD     r10,r3,r12
        EOR     r9,r9,r10
        ADD     r10,r7,r3,LSR #5
        EOR     r9,r9,r10
        ADD     r1,r1,#1
        CMP     r1,#0x20
        ADD     r2,r2,r9
        STRCS   r2,[r0,#4]
        STRCS   r3,[r0,#0]
        BCC     |L0.40|
        POP     {r4-r10,pc}
|L0.116|
        DCD     0x9e3779b9
```

100.3.3 Optimizing Keil (Thumb mode)

```
        PUSH    {r1-r7,lr}
        LDR     r5,|L0.84|
        LDR     r3,[r0,#0]
        LDR     r2,[r0,#4]
        STR     r5,[sp,#8]
        MOVS    r6,r1
        LDM     r6,{r6,r7}
        LDR     r5,[r1,#8]
        STR     r6,[sp,#4]
        LDR     r6,[r1,#0xc]
        MOVS    r4,#0
        MOVS    r1,r4
        MOV     lr,r5
        MOV     r12,r6
        STR     r7,[sp,#0]
|L0.30|
        LDR     r5,[sp,#8]
        LSLS    r6,r2,#4
        ADDS    r4,r4,r5
        LDR     r5,[sp,#4]
        LSRS    r7,r2,#5
        ADDS    r5,r6,r5
        ADDS    r6,r2,r4
        EORS    r5,r5,r6
        LDR     r6,[sp,#0]
        ADDS    r1,r1,#1
        ADDS    r6,r7,r6
        EORS    r5,r5,r6
        ADDS    r3,r5,r3
        LSLS    r5,r3,#4
        ADDS    r6,r3,r4
        ADD     r5,r5,lr
        EORS    r5,r5,r6
        LSRS    r6,r3,#5
        ADD     r6,r6,r12
        EORS    r5,r5,r6
        ADDS    r2,r5,r2
        CMP     r1,#0x20
        BCC     |L0.30|
        STR     r3,[r0,#0]
        STR     r2,[r0,#4]
        POP     {r1-r7,pc}
|L0.84|
        DCD     0x9e3779b9
```

100.3.4 Optimizing GCC 4.9.1 (ARM64)

指令清单 100.3 Optimizing GCC 4.9.1 (ARM64)

```
f:
        ldr     w3, [x0]
        mov     w4, 0
        ldr     w2, [x0,4]
        ldr     w10, [x1]
        ldr     w9, [x1,4]
        ldr     w8, [x1,8]
        ldr     w7, [x1,12]
.L2:
        mov     w5, 31161
        add     w6, w10, w2, lsl 4
```

```
        movk    w5, 0x9e37, lsl 16
        add     w1, w9, w2, lsr 5
        add     w4, w4, w5
        eor     w1, w6, w1
        add     w5, w2, w4
        mov     w6, 14112
        eor     w1, w1, w5
        movk    w6, 0xc6ef, lsl 16
        add     w3, w3, w1
        cmp     w4, w6
        add     w5, w3, w4
        add     w6, w8, w3, lsl 4
        add     w1, w7, w3, lsr 5
        eor     w1, w6, w1
        eor     w1, w1, w5
        add     w2, w2, w1
        bne     .L2
        str     w3, [x0]
        str     w2, [x0,4]
        ret
```

100.3.5　Optimizing GCC 4.4.5 (MIPS)

指令清单 100.4　Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                lui     $t2, 0x9E37
                lui     $t1, 0xC6EF
                lw      $v0, 0($a0)
                lw      $v1, 4($a0)
                lw      $t6, 0xC($a1)
                lw      $t5, 0($a1)
                lw      $t4, 4($a1)
                lw      $t3, 8($a1)
                li      $t2, 0x9E3779B9
                li      $t1, 0xC6EF3720
                move    $a1, $zero
loc_2C:                                 # CODE XREF: f+6C
                addu    $a1, $t2
                sll     $a2, $v1, 4
                addu    $t0, $a1, $v1
                srl     $a3, $v1, 5
                addu    $a2, $t5
                addu    $a3, $t4
                xor     $a2, $t0, $a2
                xor     $a2, $a3
                addu    $v0, $a2
                sll     $a3, $v0, 4
                srl     $a2, $v0, 5
                addu    $a3, $t3
                addu    $a2, $t6
                xor     $a2, $a3, $a2
                addu    $a3, $v0, $a1
                xor     $a2, $a3
                bne     $a1, $t1, loc_2C
                addu    $v1, $a2
                sw      $v1, 4($a0)
                jr      $ra
                sw      $v0, 0($a0)
```

100.4　练习题 2.13

下述程序采用了一种加密算法。这种算法的名称是什么？

100.4.1 Optimizing MSVC 2012

```
_in$ = 8                                                ; size = 2
_f      PROC
        movzx   ecx, WORD PTR _in$[esp-4]
        lea     eax, DWORD PTR [ecx*4]
        xor     eax, ecx
        add     eax, eax
        xor     eax, ecx
        shl     eax, 2
        xor     eax, ecx
        and     eax, 32                                 ; 00000020H
        shl     eax, 10                                 ; 0000000aH
        shr     ecx, 1
        or      eax, ecx
        ret     0
_f      ENDP
```

100.4.2 Keil (ARM mode)

```
f PROC
        EOR     r1,r0,r0,LSR #2
        EOR     r1,r1,r0,LSR #3
        EOR     r1,r1,r0,LSR #5
        AND     r1,r1,#1
        LSR     r0,r0,#1
        ORR     r0,r0,r1,LSL #15
        BX      lr
        ENDP
```

100.4.3 Keil (Thumb mode)

```
f PROC
        LSRS    r1,r0,#2
        EORS    r1,r1,r0
        LSRS    r2,r0,#3
        EORS    r1,r1,r2
        LSRS    r2,r0,#5
        EORS    r1,r1,r2
        LSLS    r1,r1,#31
        LSRS    r0,r0,#1
        LSRS    r1,r1,#16
        ORRS    r0,r0,r1
        BX      lr
        ENDP
```

100.4.4 Optimizing GCC 4.9.1 (ARM64)

```
f:
        uxth    w1, w0
        lsr     w2, w1, 3
        lsr     w0, w1, 1
        eor     w2, w2, w1, lsr 2
        eor     w2, w1, w2
        eor     w1, w2, w1, lsr 5
        and     w1, w1, 1
        orr     w0, w0, w1, lsl 15
        ret
```

100.4.5　Optimizing GCC 4.4.5 (MIPS)

指令清单 100.5　Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:
                andi    $a0, 0xFFFF
                srl     $v1, $a0, 2
                srl     $v0, $a0, 3
                xor     $v0, $v1, $v0
                xor     $v0, $a0, $v0
                srl     $v1, $a0, 5
                xor     $v0, $v1
                andi    $v0, 1
                srl     $a0, 1
                sll     $v0, 15
                jr      $ra
                or      $v0, $a0
```

100.5　练习题 2.14

下面这段程序采用了另一种著名算法。函数把两个输入变量输出为一个返回值。

100.5.1　MSVC 2012

```
_rt$1 = -4                                              ;size=4
_rt$2 = 8                                               ;size=4
_x$ = 8                                                 ;size=4
_y$ = 12                                                ;size=4
?f@@YAIII@Z PROC                                        ; f
        push    ecx
        push    esi
        mov     esi, DWORD PTR _x$[esp+4]
        test    esi, esi
        jne     SHORT $LN7@f
        mov     eax, DWORD PTR _y$[esp+4]
        pop     esi
        pop     ecx
        ret     0
$LN7@f:
        mov     edx, DWORD PTR _y$[esp+4]
        mov     eax, esi
        test    edx, edx
        je      SHORT $LN8@f
        or      eax, edx
        push    edi
        bsf     edi, eax
        bsf     eax, esi
        mov     ecx, eax
        mov     DWORD PTR _rt$1[esp+12], eax
        bsf     eax, edx
        shr     esi, cl
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        je      SHORT $LN22@f
$LN23@f:
        jbe     SHORT $LN2@f
        xor     esi, edx
        xor     edx, esi
        xor     esi, edx
```

```
$LN2@f:
        cmp     esi, 1
        je      SHORT $LN22@f
        sub     edx, esi
        bsf     eax, edx
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        jne     SHORT $LN23@f
$LN22@f:
        mov     ecx, edi
        shl     esi, cl
        pop     edi
        mov     eax, esi
$LN8@f:
        pop     esi
        pop     ecx
        ret     0
?f@@YAIII@Z ENDP
```

100.5.2 Keil (ARM mode)

```
||f1|| PROC
        CMP     r0,#0
        RSB     r1,r0,#0
        AND     r0,r0,r1
        CLZ     r0,r0
        RSBNE   r0,r0,#0x1f
        BX      lr
        ENDP

f PROC
        MOVS    r2,r0
        MOV     r3,r1
        MOVEQ   r0,r1
        CMPNE   r3,#0
        PUSH    {lr}
        POPEQ   {pc}
        ORR     r0,r2,r3
        BL      ||f1||
        MOV     r12,r0
        MOV     r0,r2
        BL      ||f1||
        LSR     r2,r2,r0
|L0.196|
        MOV     r0,r3
        BL      ||f1||
        LSR     r0,r3,r0
        CMP     r2,r0
        EORHI   r1,r2,r0
        EORHI   r0,r0,r1
        EORHI   r2,r1,r0
        BEQ     |L0.240|
        CMP     r2,#1
        SUBNE   r3,r0,r2
        BNE     |L0.196|
|L0.240|
        LSL     r0,r2,r12
        POP     {pc}
        ENDP
```

100.5.3 GCC 4.6.3 for Raspberry Pi (ARM mode)

```
f:
        subs    r3, r0, #0
```

```
        beq     .L162
        cmp     r1, #0
        moveq   r1, r3
        beq     .L162
        orr     r2, r1, r3
        rsb     ip, r2, #0
        and     ip, ip, r2
        cmp     r2, #0
        rsb     r2, r3, #0
        and     r2, r2, r3
        clz     r2, r2
        rsb     r2, r2, #31
        clz     ip, ip
        rsbne   ip, ip, #31
        mov     r3, r3, lsr r2
        b       .L169
.L171:
        eorhi   r1, r1, r2
        eorhi   r3, r1, r2
        cmp     r3, #1
        rsb     r1, r3, r1
        beq     .L167
.L169:
        rsb     r0, r1, #0
        and     r0, r0, r1
        cmp     r1, #0
        clz     r0, r0
        mov     r2, r0
        rsbne   r2, r0, #31
        mov     r1, r1, lsr r2
        cmp     r3, r1
        eor     r2, r1, r3
        bne     .L171
.L167:
        mov     r1, r3, asl ip
.L162:
        mov     r0, r1
        bx      lr
```

100.5.4　Optimizing GCC 4.9.1 (ARM64)

指令清单 100.6　Optimizing GCC 4.9.1 (ARM64)

```
f:
        mov     w3, w0
        mov     w0, w1
        cbz     w3, .L8
        mov     w0, w3
        cbz     w1, .L8
        mov     w6, 31
        orr     w5, w3, w1
        neg     w2, w3
        neg     w7, w5
        and     w2, w2, w3
        clz     w2, w2
        sub     w2, w6, w2
        and     w5, w7, w5
        mov     w4, w6
        clz     w5, w5
        lsr     w0, w3, w2
        sub     w5, w6, w5
        b       .L13
.L22:
        bls     .L12
        eor     w1, w1, w2
        eor     w0, w1, w2
```

```
.L12:
        cmp     w0, 1
        sub     w1, w1, w0
        beq     .L11
.L13:
        neg     w2, w1
        cmp     w1, wzr
        and     w2, w2, w1
        clz     w2, w2
        sub     w3, w4, w2
        csel    w2, w3, w2, ne
        lsr     w1, w1, w2
        cmp     w0, w1
        eor     w2, w1, w0
        bne     .L22
.L11:
        lsl     w0, w0, w5
.L8:
        ret
```

100.5.5 Optimizing GCC 4.4.5 (MIPS)

指令清单 100.7 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:

var_20          = -0x20
var_18          = -0x18
var_14          = -0x14
var_10          = -0x10
var_C           = -0xC
var_8           = -8
var_4           =-4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x30
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x30+var_4($sp)
                sw      $s4, 0x30+var_8($sp)
                sw      $s3, 0x30+var_C($sp)
                sw      $s2, 0x30+var_10($sp)
                sw      $s1, 0x30+var_14($sp)
                sw      $s0, 0x30+var_18($sp)
                sw      $gp, 0x30+var_20($sp)
                move    $s0, $a0
                beqz    $a0, loc_154
                move    $s1, $a1
                bnez    $a1, loc_178
                or      $s2, $a1, $a0
                move    $s1, $a0

loc_154:                                # CODE XREF: f+2C
                lw      $ra, 0x30+var_4($sp)
                move    $v0, $s1
                lw      $s4, 0x30+var_8($sp)
                lw      $s3, 0x30+var_C($sp)
                lw      $s2, 0x30+var_10($sp)
                lw      $s1, 0x30+var_14($sp)
                lw      $s0, 0x30+var_18($sp)
                jr      $ra
                addiu   $sp, 0x30

loc_178:                                # CODE XREF: f+34
```

```
                lw      $t9, (__clzsi2 & 0xFFFF)($gp)
                negu    $a0, $s2
                jalr    $t9
                and     $a0, $s2
                lw      $gp, 0x30+var_20($sp)
                bnez    $s2, loc_20C
                li      $s4, 0x1F
                move    $s4, $v0
loc_198:                                # CODE XREF: f:loc_20C
                lw      $t9, (__clzsi2 & 0xFFFF)($gp)
                negu    $a0, $s0
                jalr    $t9
                and     $a0, $s0
                nor     $v0, $zero, $v0
                lw      $gp, 0x30+var_20($sp)
                srlv    $s0, $v0
                li      $s3, 0x1F
                li      $s2, 1
loc_1BC:                                # CODE XREF: f+F0
                lw      $t9, (__clzsi2 & 0xFFFF)($gp)
                negu    $a0, $s1
                jalr    $t9
                and     $a0, $s1
                lw      $gp, 0x30+var_20($sp)
                beqz    $s1, loc_1DC
                or      $at, $zero
                subu    $v0, $s3, $v0
loc_1DC:                                # CODE XREF: f+BC
                srlv    $s1, $v0
                xor     $v1, $s1, $s0
                beq     $s0, $s1, loc_214
                sltu    $v0, $s1, $s0
                beqz    $v0, loc_1FC
                or      $at, $zero
                xor     $s1, $v1
                xor     $s0, $s1, $v1
loc_1FC:                               # CODE XREF: f+D8
                beq     $s0, $s2, loc_214
                subu    $s1, $s0
                b       loc_1BC
                or      $at, $zero
loc_20C:                                # CODE XREF: f+78
                b       loc_198
                subu    $s4, $v0
loc_214:                                # CODE XREF: f+D0
                                        # f:loc_1FC
                lw      $ra, 0x30+var_4($sp)
                sllv    $s1, $s0, $s4
                move    $v0, $s1
                lw      $s4, 0x30+var_8($sp)
                lw      $s3, 0x30+var_C($sp)
                lw      $s2, 0x30+var_10($sp)
                lw      $s1, 0x30+var_14($sp)
                lw      $s0, 0x30+var_18($sp)
                jr      $ra
                addiu   $sp, 0x30
```

100.6　练习题 2.15

这个程序实现了一种著名的算法。请问，这个算法的名称是什么？

在 x86 平台上，程序使用 FPU 进行运算；而在 x64 平台上，程序使用的是 SIMD 指令集。这属于正常

现象，详细介绍请参见本书第 27 章。

100.6.1 Optimizing MSVC 2012 x64

```
__real@412e848000000000 DQ 0412e848000000000r    ; 1e+006
__real@4010000000000000 DQ 04010000000000000r    ;4
__real@4008000000000000 DQ 04008000000000000r    ;3
__real@3f800000 DD 03f800000r                    ;1

tmp$1 = 8
tmp$2 = 8
f       PROC
        movsdx  xmm3, QWORD PTR __real@4008000000000000
        movss   xmm4, DWORD PTR __real@3f800000
        mov     edx, DWORD PTR ?RNG_state@?1??get_rand@@9@9
        xor     ecx, ecx
        mov     r8d, 200000                              ; 00030d40H
        npad    2 ; align next label
$LL4@f:
        imul    edx, 1664525                             ; 0019660dH
        add     edx, 1013904223                          ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                             ; 007fffffH
        imul    edx, 1664525                             ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                          ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                             ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN3@f
        inc     ecx
$LN3@f:
        imul    edx, 1664525                             ; 0019660dH
        add     edx, 1013904223                          ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                             ; 007fffffH
        imul    edx, 1664525                             ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                          ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                             ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
```

```
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN15@f
        inc     ecx
$LN15@f:
        imul    edx, 1664525                    ; 0019660dH
        add     edx, 1013904223                 ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                    ; 007fffffH
        imul    edx, 1664525                    ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                 ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                    ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN16@f
        inc     ecx
$LN16@f:
        imul    edx, 1664525                    ; 0019660dH
        add     edx, 1013904223                 ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                    ; 007fffffH
        imul    edx, 1664525                    ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                 ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                    ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN17@f
        inc     ecx
$LN17@f:
        imul    edx, 1664525                    ; 0019660dH
        add     edx, 1013904223                 ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                    ; 007fffffH
        imul    edx, 1664525                    ; 0019660dH
```

```
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN18@f
        inc     ecx
$LN18@f:
        dec     r8
        jne     $LL4@f
        movd    xmm0, ecx
        mov     DWORD PTR ?RNG_state@?1??get_rand@@9@9, edx
        cvtdq2ps xmm0, xmm0
        cvtps2pd xmm1, xmm0
        mulsd   xmm1, QWORD PTR __real@4010000000000000
        divsd   xmm1, QWORD PTR __real@412e848000000000
        cvtpd2ps xmm0, xmm1
        ret     0
f       ENDP
```

100.6.2 Optimizing GCC 4.4.6 x64

```
f1:
        mov     eax, DWORD PTR v1.2084[rip]
        imul    eax, eax, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2084[rip], eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [rsp-4], eax
        movss   xmm0, DWORD PTR [rsp-4]
        subss   xmm0, DWORD PTR .LC0[rip]
        ret
f:
        push    rbp
        xor     ebp, ebp
        push    rbx
        xor     ebx, ebx
        sub     rsp, 16
.L6:
        xor     eax, eax
        call    f1
        xor     eax, eax
        movss   DWORD PTR [rsp], xmm0
        call    f1
        movss   xmm1, DWORD PTR [rsp]
        mulss   xmm0, xmm0
        mulss   xmm1, xmm1
        lea     eax, [rbx+1]
        addss   xmm1, xmm0
        movss   xmm0, DWORD PTR .LC1[rip]
        ucomiss xmm0, xmm1
```

```
        cmova   ebx, eax
        add     ebp, 1
        cmp     ebp, 1000000
        jne     .L6
        cvtsi2ss        xmm0, ebx
        unpcklps        xmm0, xmm0
        cvtps2pd        xmm0, xmm0
        mulsd   xmm0, QWORD PTR .LC2[rip]
        divsd   xmm0, QWORD PTR .LC3[rip]
        add     rsp, 16
        pop     rbx
        pop     rbp
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
v1.2084:
        .long   305419896
.LC0:
        .long   1077936128
.LC1:
        .long   1065353216
.LC2:
        .long   0
        .long   1074790400
.LC3:
        .long   0
        .long   1093567616
```

100.6.3 Optimizing GCC 4.8.1 x86

```
f1:
        sub     esp, 4
        imul    eax, DWORD PTR v1.2023, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2023, eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [esp], eax
        fld     DWORD PTR [esp]
        fsub    DWORD PTR .LC0
        add     esp, 4
        ret
f:
        push    esi
        mov     esi, 1000000
        push    ebx
        xor     ebx, ebx
        sub     esp, 16
.L7:
        call    f1
        fstp    DWORD PTR [esp]
        call    f1
        lea     eax, [ebx+1]
        fld     DWORD PTR [esp]
        fmul    st, st(0)
        fxch    st(1)
        fmul    st, st(0)
        faddp   st(1), st
        fld1
        fucomip st, st(1)
        fstp    st(0)
        cmova   ebx, eax
        sub     esi, 1
        jne     .L7
        mov     DWORD PTR [esp+4], ebx
        fild    DWORD PTR [esp+4]
```

```
        fmul    DWORD PTR .LC3
        fdiv    DWORD PTR .LC4
        fstp    DWORD PTR [esp+8]
        fld     DWORD PTR [esp+8]
        add     esp, 16
        pop     ebx
        pop     esi
        ret
v1.2023:
        .long   305419896
.LC0:
        .long   1077936128
.LC3:
        .long   1082130432
.LC4:
        .long   1232348160
```

100.6.4 Keil（ARM 模式）：面向 Cortex-R4F CPU 的代码

```
f1      PROC
        LDR       r1,|L0.184|
        LDR       r0,[r1,#0]  ; v1
        LDR       r2,|L0.188|
        VMOV.F32  s1,#3.00000000
        MUL       r0,r0,r2
        LDR       r2,|L0.192|
        ADD       r0,r0,r2
        STR       r0,[r1,#0]  ; v1
        BFC       r0,#23,#9
        ORR       r0,r0,#0x40000000
        VMOV      s0,r0
        VSUB.F32  s0,s0,s1
        BX        lr
        ENDP

f       PROC
        PUSH      {r4,r5,lr}
        MOV       r4,#0
        LDR       r5,|L0.196|
        MOV       r3,r4
|L0.68|
        BL        f1
        VMOV.F32  s2,s0
        BL        f1
        VMOV.F32  s1,s2
        ADD       r3,r3,#1
        VMUL.F32  s1,s1,s1
        VMLA.F32  s1,s0,s0
        VMOV      r0,s1
        CMP       r0,#0x3f800000
        ADDLT     r4,r4,#1
        CMP       r3,r5
        BLT       |L0.68|
        VMOV      s0,r4
        VMOV.F64  d1,#4.00000000
        VCVT.F32.S32 s0,s0
        VCVT.F64.F32 d0,s0
        VMUL.F64  d0,d0,d1
        VLDR      d1,|L0.200|
        VDIV.F64  d2,d0,d1
        VCVT.F32.F64 s0,d2
        POP       {r4,r5,pc}
        ENDP

|L0.184|
        DCD       ||.data||
```

```
|L0.188|
        DCD       0x0019660d
|L0.192|
        DCD       0x3c6ef35f
|L0.196|
        DCD       0x000f4240
|L0.200|
        DCFD      0x412e848000000000 ; 1000000

        DCD       0x00000000
        AREA ||.data||, DATA, ALIGN=2
v1
        DCD       0x12345678
```

100.6.5　Optimizing GCC 4.9.1 (ARM64)

指令清单 100.8　Optimizing GCC 4.9.1 (ARM64)

```
f1:
        adrp    x2, .LANCHOR0
        mov     w3, 26125
        mov     w0, 62303
        movk    w3, 0x19, lsl 16
        movk    w0, 0x3c6e, lsl 16
        ldr     w1, [x2,#:lo12:.LANCHOR0]
        fmov    s0, 3.0e+0
        madd    w0, w1, w3, w0
        str     w0, [x2,#:lo12:.LANCHOR0]
        and     w0, w0, 8388607
        orr     w0, w0, 1073741824
        fmov    s1, w0
        fsub    s0, s1, s0
        ret
mail_function:
        adrp    x7, .LANCHOR0
        mov     w3, 16960
        movk    w3, 0xf, lsl 16
        mov     w2, 0
        fmov    s2, 3.0e+0
        ldr     w1, [x7,#:lo12:.LANCHOR0]
        fmov    s3, 1.0e+0

.L5:
        mov     w6, 26125
        mov     w0, 62303
        movk    w6, 0x19, lsl 16
        movk    w0, 0x3c6e, lsl 16
        mov     w5, 26125
        mov     w4, 62303
        madd    w1, w1, w6, w0
        movk    w5, 0x19, lsl 16
        movk    w4, 0x3c6e, lsl 16
        and     w0, w1, 8388607
        add     w6, w2, 1
        orr     w0, w0, 1073741824
        madd    w1, w1, w5, w4
        fmov    s0, w0
        and     w0, w1, 8388607
        orr     w0, w0, 1073741824
        fmov    s1, w0
        fsub    s0, s0, s2
        fsub    s1, s1, s2
        fmul    s1, s1, s1
        fmadd   s0, s0, s0, s1
        fcmp    s0, s3
```

```
        csel    w2, w2, w6, pl
        subs    w3, w3, #1
        ben     .L5
        scvtf   s0, w2
        str     w1, [x7, #:lo12:.LANCHOR]]
        fmov    d1, 4.0e+0
        fcvt    d0, s0
        fmul    d0, d0, d1
        ldr     d1, .LC0
        fdiv    d0, d0, d1
        fcvt    s0, d0
        ret
.LC0:
        .word   0
        .word   1093567616
.V1:
        .word   1013904223
.V2:
        .word   1664525
.LANCHOR0: = . + 0
v3.3095
        .word   305419896
```

100.6.6 Optimizing GCC 4.4.5 (MIPS)

指令清单 100.9 Optimizing GCC 4.4.5 (MIPS) (IDA)

```
f1:
        mov     eax, DWORD PTR v1.2084[rip]
        imul    eax, eax, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2084[rip], eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [rsp-4], eax
        movss   xmm0, DWORD PTR [rsp-4]
        subss   xmm0, DWORD PTR .LC0[rip]
        ret
f:
        push    rbp
        xor     ebp, ebp
        push    rbx
        xor     ebx, ebx
        sub     rsp, 16
.L6:
        xor     eax, eax
        call    f1
        xor     eax, eax
        movss   DWORD PTR [rsp], xmm0
        call    f1
        movss   xmm1, DWORD PTR [rsp]
        mulss   xmm0, xmm0
        mulss   xmm1, xmm1
        lea     eax, [rbx+1]
        addss   xmm1, xmm0
        movss   xmm0, DWORD PTR .LC1[rip]
        ucomiss xmm0, xmm1
        cmova   ebx, eax
        add     ebp, 1
        cmp     ebp, 1000000
        jne     .L6
        cvtsi2ss        xmm0, ebx
        unpcklps        xmm0, xmm0
        cvtps2pd        xmm0, xmm0
        mulsd   xmm0, QWORD PTR .LC2[rip]
```

```
        divsd   xmm0, QWORD PTR .LC3[rip]
        add     rsp, 16
        pop     rbx
        pop     rbp
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
v1.2084:
        .long   305419896
.LC0:
        .long   1077936128
.LC1:
        .long   1065353216
.LC2:
        .long   0
        .long   1074790400
.LC3:
        .long   0
        .long   1093567616
```

100.7　练习题 2.16

这个题目是一个著名的函数。请问它的计算结果是什么？如果输入了 4 和 2，该程序会出现栈溢出问题。为什么会这样，代码里有错误么？

100.7.1　Optimizing MSVC 2012 x64

```
m$ = 48
n$ = 56
f       PROC
$LN14:
        push    rbx
        sub     rsp, 32
        mov     eax, edx
        mov     ebx, ecx
        test    ecx, ecx
        je      SHORT $LN11@f
$LL5@f:
        test    eax, eax
        jne     SHORT $LN1@f
        mov     eax, 1
        jmp     SHORT $LN12@f
$LN1@f:
        lea     edx, DWORD PTR [rax-1]
        mov     ecx, ebx
        call    f
$LN12@f:
        dec     ebx
        test    ebx, ebx
        jne     SHORT $LL5@f
$LN11@f:
        inc     eax
        add     rsp, 32
        pop     rbx
        ret     0
f       ENDP
```

100.7.2　Optimizing Keil (ARM mode)

```
f PROC
        PUSH    {r4,lr}
```

```
        MOVS    r4,r0
        ADDEQ   r0,r1,#1
        POPEQ   {r4,pc}
        CMP     r1,#0
        MOVEQ   r1,#1
        SUBEQ   r0,r0,#1
        BEQ     |L0.48|
        SUB     r1,r1,#1
        BL      f
        MOV     r1,r0
        SUB     r0,r4,#1
|L0.48|
        POP     {r4,lr}
        B       f
        ENDP
```

100.7.3 Optimizing Keil（Thumb mode）

```
f PROC
        PUSH    {r4,lr}
        MOVS    r4,r0
        BEQ     |L0.26|
        CMP     r1,#0
        BEQ     |L0.30|
        SUBS    r1,r1,#1
        BL      f
        MOVS    r1,r0
|L0.18|
        SUBS    r0,r4,#1
        BL      f
        POP     {r4,pc}
|L0.26|
        ADDS    r0,r1,#1
        POP     {r4,pc}
|L0.30|
        MOVS    r1,#1
        B       |L0.18|
        ENDP
```

100.7.4 Non-optimizing GCC 4.9.1（ARM64）

指令清单 100.10 Non-optimizing GCC 4.9.1（ARM64）

```
f:
        stp     x29, x30, [sp, -48]!
        add     x29, sp, 0
        str     x19, [sp,16]
        str     w0, [x29,44]
        str     w1, [x29,40]
        ldr     w0, [x29,44]
        cmp     w0, wzr
        bne     .L2
        ldr     w0, [x29,40]
        add     w0, w0, 1
        b       .L3
.L2:
        ldr     w0, [x29,40]
        cmp     w0, wzr
        bne     .L4
        ldr     w0, [x29,44]
        sub     w0, w0, #1
        mov     w1, 1
        bl      ack
        b       .L3
```

```
.L4:
        ldr     w0, [x29,44]
        sub     w19, w0, #1
        ldr     w0, [x29,40]
        sub     w1, w0, #1
        ldr     w0, [x29,44]
        bl      ack
        mov     w1, w0
        mov     w0, w19
        bl      ack
.L3:
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
```

100.7.5　Optimizing GCC 4.9.1（ARM64）

开启优化模式之后，GCC 生成的代码反而更长了。这是为什么？

指令清单 100.11　Optimizing GCC 4.9.1（ARM64）

```
ack:
        stp     x29, x30, [sp, -160]!
        add     x29, sp, 0
        stp     d8, d9, [sp,96]
        stp     x19, x20, [sp,16]
        stp     d10, d11, [sp,112]
        stp     x21, x22, [sp,32]
        stp     d12, d13, [sp,128]
        stp     x23, x24, [sp,48]
        stp     d14, d15, [sp,144]
        stp     x25, x26, [sp,64]
        stp     x27, x28, [sp,80]
        cbz     w0, .L2
        sub     w0, w0, #1
        fmov    s10, w0
        b       .L4
.L46:
        fmov    w0, s10
        mov     w1, 1
        sub     w0, w0, #1
        fmov    s10, w0
        fmov    w0, s13
        cbz     w0, .L2
.L4:
        fmov    s13, s10
        cbz     w1, .L46
        sub     w1, w1, #1
        fmov    s11, s10
        b       .L7
.L48:
        fmov    w0, s11
        mov     w1, 1
        sub     w0, w0, #1
        fmov    s11, w0
        fmov    w0, s14
        cbz     w0, .L47
.L7:
        fmov    s14, s11
        cbz     w1, .L48
        sub     w1, w1, #1
        fmov    s12, s11
        b       .L10
.L50:
        fmov    w0, s12
        mov     w1, 1
```

```
        sub     w0, w0, #1
        fmov    s12, w0
        fmov    w0, s15
        cbz     w0, .L49
.L10:
        fmov    s15, s12
        cbz     w1, .L50
        sub     w1, w1, #1
        fmov    s8, s12
        b       .L13
.L52:
        fmov    w0, s8
        mov     w1, 1
        sub     w0, w0, #1
        fmov    s8, w0
        fmov    w0, s9
        cbz     w0, .L51
.L13:
        fmov    s9, s8
        cbz     w1, .L52
        sub     w1, w1, #1
        fmov    w22, s8
        b       .L16
.L54:
        mov     w1, 1
        sub     w22, w22, #1
        cbz     w28, .L53
.L16:
        mov     w28, w22
        cbz     w1, .L54
        sub     w1, w1, #1
        mov     w21, w22
        b       .L19
.L56:
        mov     w1, 1
        sub     w21, w21, #1
        cbz     w24, .L55
.L19:
        mov     w24, w21
        cbz     w1, .L56
        sub     w1, w1, #1
        mov     w20, w21
        b       .L22
.L58:
        mov     w1, 1
        sub     w20, w20, #1
        cbz     w25, .L57
.L22:
        mov     w25, w20
        cbz     w1, .L58
        sub     w1, w1, #1
        mov     w26, w20
        b       .L25
.L60:
        mov     w1, 1
        sub     w26, w26, #1
        cbz     w27, .L59
.L25:
        mov     w27, w26
        cbz     w1, .L60
        sub     w1, w1, #1
        mov     w19, w26
        b       .L28
.L62:
        mov     w23, w19
        mov     w1, 1
        sub     w19, w19, #1
```

```
        cbz     w23, .L61
.L28:
        add     w0, w19, 1
        cbz     w1, .L62
        sub     w1, w1, #1
        mov     w23, w19
        sub     w19, w19, #1
        bl      ack
        mov     w1, w0
        cbnz    w23, .L28
.L61:
        add     w1, w1, 1
        sub     w26, w26, #1
        cbnz    w27, .L25
.L59:
        add     w1, w1, 1
        sub     w20, w20, #1
        cbnz    w25, .L22
.L57:
        add     w1, w1, 1
        sub     w21, w21, #1
        cbnz    w24, .L19
.L55:
        add     w1, w1, 1
        sub     w22, w22, #1
        cbnz    w28, .L16
.L53:
        fmov    w0, s8
        add     w1, w1, 1
        sub     w0, w0, #1
        fmov    s8, w0
        fmov    w0, s9
        cbnz    w0, .L13
.L51:
        fmov    w0, s12
        add     w1, w1, 1
        sub     w0, w0, #1
        fmov    s12, w0
        fmov    w0, s15
        cbnz    w0, .L10
.L49:
        fmov    w0, s11
        add     w1, w1, 1
        sub     w0, w0, #1
        fmov    s11, w0
        fmov    w0, s14
        cbnz    w0, .L7
.L47:
        fmov    w0, s10
        add     w1, w1, 1
        sub     w0, w0, #1
        fmov    s10, w0
        fmov    w0, s13
        cbnz    w0, .L4
.L2:
        add     w0, w1, 1
        ldp     d8, d9, [sp,96]
        ldp     x19, x20, [sp,16]
        ldp     d10, d11, [sp,112]
        ldp     x21, x22, [sp,32]
        ldp     d12, d13, [sp,128]
        ldp     x23, x24, [sp,48]
        ldp     d14, d15, [sp,144]
        ldp     x25, x26, [sp,64]
        ldp     x27, x28, [sp,80]
        ldp     x29, x30, [sp], 160
        ret
```

100.7.6 Non-optimizing GCC 4.4.5 (MIPS)

指令清单 100.12 Non-optimizing GCC 4.4.5 (MIPS) (IDA)

```
f:                                      # CODE XREF: f+64
                                        # f+94 ...

var_C           = -0xC
var_8           = -8
var_4           = -4
arg_0           = 0
arg_4           = 4

                addiu   $sp, -0x28
                sw      $ra, 0x28+var_4($sp)
                sw      $fp, 0x28+var_8($sp)
                sw      $s0, 0x28+var_C($sp)
                move    $fp, $sp
                sw      $a0, 0x28+arg_0($fp)
                sw      $a1, 0x28+arg_4($fp)
                lw      $v0, 0x28+arg_0($fp)
                or      $at, $zero
                bnez    $v0, loc_40
                or      $at, $zero
                lw      $v0, 0x28+arg_4($fp)
                or      $at, $zero
                addiu   $v0, 1
                b       loc_AC
                or      $at, $zero

loc_40:                                 # CODE XREF: f+24
                lw      $v0, 0x28+arg_4($fp)
                or      $at, $zero
                bnez    $v0, loc_74
                or      $at, $zero
                lw      $v0, 0x28+arg_0($fp)
                or      $at, $zero
                addiu   $v0, -1
                move    $a0, $v0
                li      $a1, 1
                jal     f
                or      $at, $zero
                b       loc_AC
                or      $at, $zero

loc_74:                                 # CODE XREF: f+48
                lw      $v0, 0x28+arg_0($fp)
                or      $at, $zero
                addiu   $s0, $v0, -1
                lw      $v0, 0x28+arg_4($fp)
                or      $at, $zero
                addiu   $v0, -1
                lw      $a0, 0x28+arg_0($fp)
                move    $a1, $v0
                jal     f
                or      $at, $zero
                move    $a0, $s0
                move    $a1, $v0
                jal     f
                or      $at, $zero

loc_AC:                                 # CODE XREF: f+38
                                        # f+6C
                move    $sp, $fp
                lw      $ra, 0x28+var_4($sp)
```

```
lw      $fp, 0x28+var_8($sp)
lw      $s0, 0x28+var_C($sp)
addiu   $sp, 0x28
jr      $ra
or      $at, $zero
```

100.8　练习题 2.17

下列程序向 stdout 输出信息，而且每次输出的结果还不一样。请问它输出的是什么信息？

请下载编译后的可执行文件:

- Linux x64（go.yurichev.com/17170）。
- Mac OS X（go.yurichev.com/17171）。
- Linux MIPS（go.yurichev.com/17172）。
- Win32（go.yurichev.com/17173）。
- Win64（go.yurichev.com/17174）。

可能有个别版本的 Windows 无法执行这个程序。如果发生这种情况，请下载 MSVC 2012 redist.（http://www. microsoft.com/en-us/download/details.aspx?id=30679）。

100.9　练习题 2.18

下列程序会验证密码。请找到它的密码。

另外，它可以接受的密码不是唯一的。请尽可能地多列举一些密码。

您还可以对它进行修改，改变程序的密码：

- Win32（go.yurichev.com/17175）。
- Linux x86（go.yurichev.com/17176）。
- Mac OS X（go.yurichev.com/17177）。
- Linux MIPS（go.yurichev.com/17178）。

100.10　练习题 2.19

这组题目和练习题 2.18 的练习内容相同：

- Win32（go.yurichev.com/17179）。
- Linux x86（go.yurichev.com/17180）。
- Mac OS X（go.yurichev.com/17181）。
- Linux MIPS（go.yurichev.com/17182）。

100.11　练习题 2.20

下列程序向 stdout 输出信息。请问它输出的是什么信息？

- Linux x64（go.yurichev.com/17183）。
- Mac OS X（go.yurichev.com/17184）。
- Linux ARM Raspberry Pi（go.yurichev.com/17185）。
- Linux MIPS（go.yurichev.com/17186）。
- Win64（go.yurichev.com/17187）。

第 101 章　高难度练习题

这种难度题目的解答时间会很长。解答时间可能长达一整天。

101.1　练习题 3.2

下列可执行程序实现了某种著名的加密机制。请问它实现的是什么算法？

- Windows x86（go.yurichev.com/17188）。
- Linux x86（go.yurichev.com/17189）。
- Mac OS X（x64）（go.yurichev.com/17190）。
- Linux MIPS（go.yurichev.com/17191）。

101.2　练习题 3.3

下列程序可打开并读取某个文件，而后计算某种值并在屏幕上输出浮点数。请问它实现的是什么功能？

- Windows x86（go.yurichev.com/17192）。
- Linux x86（go.yurichev.com/17193）。
- Mac OS X（x64）（go.yurichev.com/17194）。
- Linux MIPS（go.yurichev.com/17195）。

101.3　练习题 3.4

这是一个用密码加、解密文件的工具。虽然我们找到了密文，但是找不到加密密码。此外，我们还知道原文是英文的文本文件。虽然程序采用了较强的加密机制，但是它存在严重的功能缺陷。这种缺陷大大降低了解密的难度。

请找到程序的缺陷，并把密文还原为明文。

- Windows x86（go.yurichev.com/17196）。
- 密文下载地址：http://go.yurichev.com/17197。

101.4　练习题 3.5

下列程序实现了版权保护机制。它会读取 key 文件，核对其中的用户名和序列号。

本题的任务分为两个：

- （低难度）使用 tracer 或别的 debugger，强制程序认可篡改过的 key 文件。
- （中等难度）修改用户名，但是不得修改可执行程序。

程序的下载地址如下：

- Windows x86（go.yurichev.com/17198）。
- Linux x86（go.yurichev.com/17199）。
- Mac OS X（x64）（go.yurichev.com/17200）。

- Linux MIPS（go.yurichev.com/17201）。
- Key 文件（go.yurichev.com/17202）。

101.5　练习题 3.6

下列程序属于羽量级的 web 服务器程序。虽然它支持静态文件，但是不支持 cgi 等动态脚本。这个程序里有 4 个以上的安全漏洞。找到这些漏洞，并且想办法利用它们攻陷服务器。

- Windows x86（go.yurichev.com/17203）。
- Linux x86（go.yurichev.com/17204）。
- Mac OS X（x64）（go.yurichev.com/17205）。

101.6　练习题 3.8

下列程序实现了著名的数据压缩算法。或许是因为原作者在输入代码时敲错了按键，它的解压缩功能存在问题。我们能够在执行过程中看到它的 bug。

压缩之前的原文件：go.yurichev.com/17206。

压缩之后的压缩包：go.yurichev.com/17207。

解压之后的（故障）文件：go.yurichev.com/17208。

请找到程序中的 bug。如果可能的话，还请修改可执行文件，修补这个 bug。

- Windows x86（go.yurichev.com/17209）。
- Linux x86（go.yurichev.com/17210）。
- Mac OS X（x64）（go.yurichev.com/17211）。

第 102 章　Crackme/Keygenme

有关软件知识产权保护措施的相关分析，请参见：http://crackmes.de/users/yonkie/。

附录 A　x86

A.1　数据类型

16 位（8086/80286）、32 位（80386 等）和 64 位系统常用的数据类型有：

byte（字节）：8 位数据。声明字节型数组和变量的汇编伪指令是 DB。计算机使用寄存器的低 8 位空间存储字节型数据。也就是说，字节型数据通常存储在（寄存器助记符）AL/BL/CL/DL/AH/BH/ CH/DH/SIL/DIL/R*L 里。

word（字）：16 位数据。声明字型数组和变量的汇编伪指令是 DW。计算机使用寄存器的 16 位空间存储 word 型数据。也就是说，字型数据通常存储在（寄存器助记符）AX/BX/CX/DX/SI/DI/R*W 里。

dword/double word：32 位数据。声明 DWord 型数组和变量的汇编伪指令是 DD。x86 CPU 的标准寄存器及 x64 CPU 寄存器的 32 位空间都可存储 DWord 型数据。16 位应用程序则用寄存器对来存储 DWord 型数据。

qword/quad word：64 位数据。声明 QWord 型数组和变量的汇编伪指令是 DQ。32 位系统使用一对寄存器来存储 QWord 型数据。

tbyte（10 字节型）：80 位，即 10 字节数据。符合 IEEE 754 标准的 FPU 寄存器都采用这种类型的数据。

paragraph（16 字节型）：这种类型的数据主要出现在 MS-DOS 操作系统的程序里。

Windows API 所定义的各种同名的数据类型（包括 BYTE WORD DWORD）及其数据存储空间，同样遵循上述标准。

A.2　通用寄存器

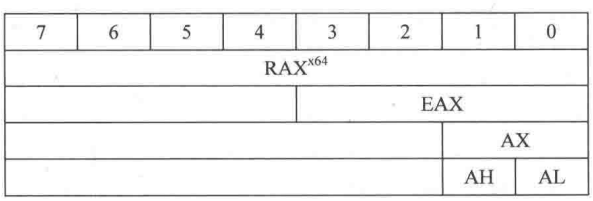
x86-64 的 CPU 可以直接调用多数通用寄存器的 8 位（byte）和 16 位（word）存储空间。通用寄存器有前向兼容的特性（可兼容最早的 8080CPU）。早期的 8 位 CPU（例如 8080）可以用一对 8 位寄存器存储一个 16 位数据。如此一来，面向早期 8080 平台的程序就可以照常访问 16 位寄存器的低 8 位空间、高 8 位空间，还能够把这两个寄存器的数据当作一个整体的 16 位寄存器使用。x86 平台的这种前向兼容的特性，或许是为了方便人们在不同平台上移植程序。而采用 RISC 精简指令集的 CPU 则通常没有这种特性。

此外，x86-64 CPU 上有 R 开头的寄存器，而 80386 以后的 CPU 都有 E 开头的寄存器。可见，R-寄存器属于 64 位寄存器，而 E-寄存器属于 32 位寄存器。

x86-64 CPU 还比 x86 CPU 多了 8 个通用寄存器，即 R8～R15 寄存器。

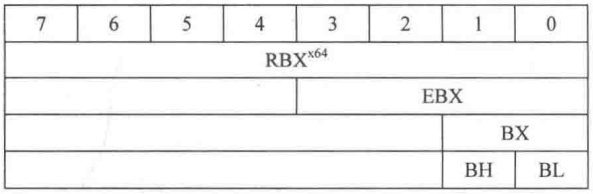
在 Intel 官方手册中，这些寄存器的低 8 位空间（byte）的助记符带有“L”后缀（例如 R8L），而 IDA 程序则给它们加上了后缀“B”（例如 R8B）。

A.2.1　RAX/EAX/AX/AL

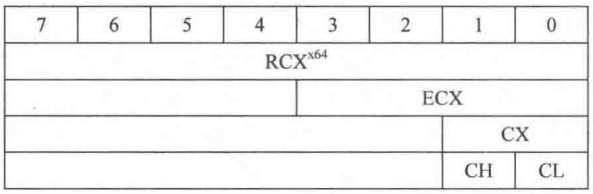


又称累加寄存器（Accumulator）。函数的返回值通常保存在这个寄存器里。

A.2.2 RBX/EBX/BX/BL

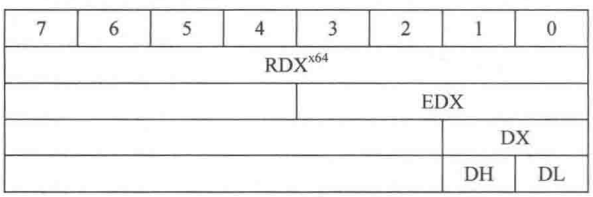


A.2.3 RCX/ECX/CX/CL

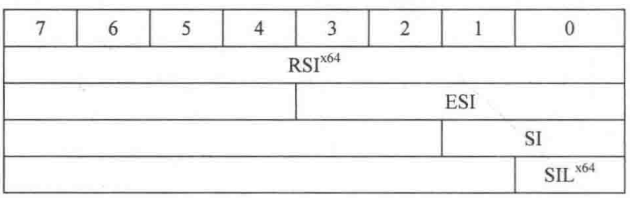


又称计数器（Counter）。“REP”开头的指令、位移运算（SHL/SHR/RxL/RxR）通常都会影响这个寄存器的状态。

A.2.4 RDX/EDX/DX/DL

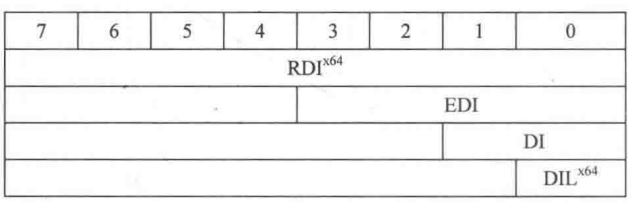


A.2.5 RSI/ESI/SI/SIL



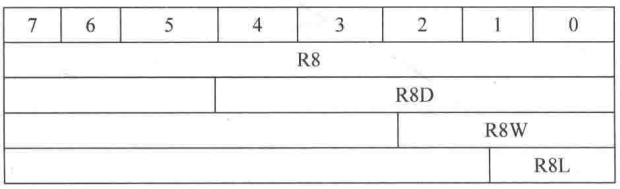
又称源寄存器（Source），是 REP MOVSx 和 REP CMPSx 指令默认的数据源。

A.2.6 RDI/EDI/DI/DIL

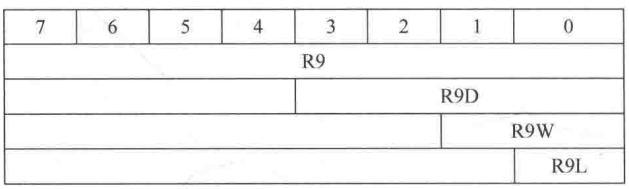


又称目标寄存器（Destination），是 REP MOVSx、REP STOSx 指令默认的目标寄存器。

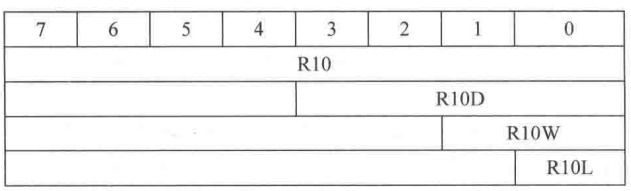
A.2.7 R8/R8D/R8W/R8L



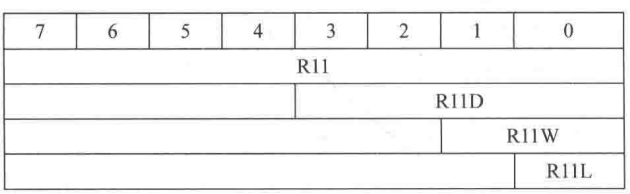
A.2.8 R9/R9D/R9W/R9L



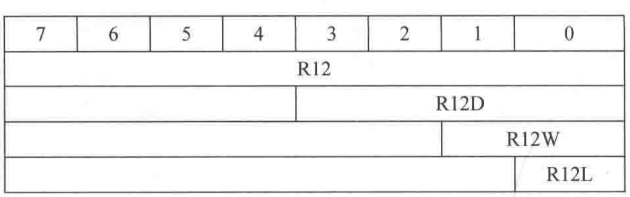
A.2.9 R10/R10D/R10W/R10L



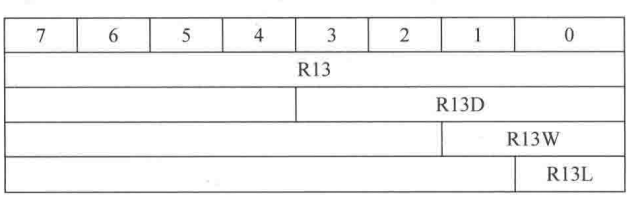
A.2.10 R11/R11D/R11W/R11L



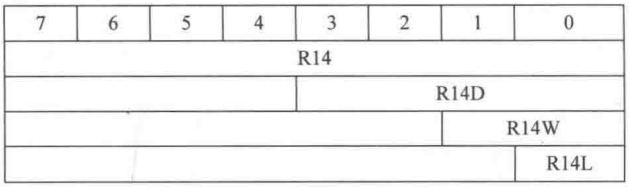
A.2.11 R12/R12D/R12W/R12L



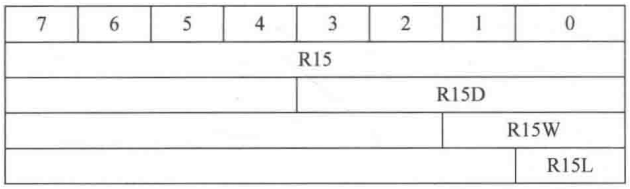
A.2.12 R13/R13D/R13W/R13L



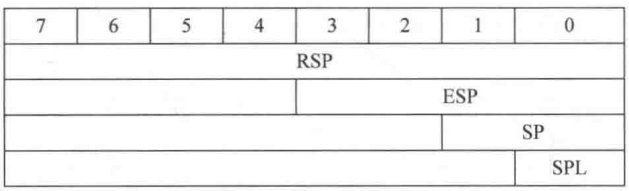
A.2.13 R14/R14D/R14W/R14L



A.2.14 R15/R15D/R15W/R15L

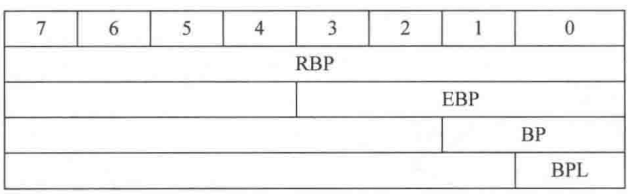


A.2.15 RSP/ESP/SP/SPL



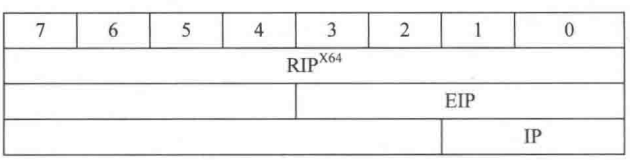
SP 是栈指针 Stack Pointer 的缩写。在初始化之后，它是当前栈地址的指针。

A.2.16 RBP/EBP/BP/BPL



帧指针（Frame Pointer），通常是局部变量的指针。在调用函数时，它也常常用来传递参数。有关这个寄存器的详细介绍，请参照本书的 7.1.2 节。

A.2.17 RIP/EIP/IP



指令指针 instruction pointer 应当总是指向接下来将要执行的那条指令。正常情况下，无法直接干预它的值。但是，下述指令可以等效地实现调整指令指针的功能：

```
MOV EAX, ...
JMP EAX
或者：
PUSH Value
RET
```

A.2.18 段地址寄存器 CS/DS/ES/SS/FS/GS

CS/DS/SS/ES 分别代表 Code Segment 代码段寄存器、Data Segment 数据段寄存器、Stack Segment 堆栈段寄存器和 Extra Segment 附加段寄存器。

在 Win32 系统里，FS 附加段寄存器（Extra Segment Register）承担 TLS（线程本地存储/Thread Local Storage）的角色；而在 Linux 系统里，GS（另一个附加段寄存器）承担这个角色。早期，这两个寄存器用于实现段式寻址；而现在，它们用于提供更为快速的 TLS 和 TIB（线程信息块/ThreadInformationBlock）功能。有关段地址寄存器的详细介绍，请参见本书第 94 章。

A.2.19 标识寄存器

标识寄存器即 Eflags。

Bit 位（及掩码）	缩写（及含义）	描　述
0（1）	CF（进/借位）	除了常规计算指令之外，专门操作 CF 的指令还有 CLC/STC/CMC
2（4）	PF（奇偶标识位）	参见 17.7.1 节
4（0x10）	AF（辅助进/借位标识）	
6（0x40）	ZF（零标识位）	ZF 用来反映运算结果是否为 0。如果运算结果为 0，则其值为 1，否则其值为 0
7（0x80）	SF（符号位）	
8（0x100）	TF（追踪标识）	当追踪标志 TF 被置为 1 时，CPU 进入单步执行方式，即每执行一条指令，产生一个单步中断请求。这种方式主要用于程序的调试
9（0x200）	IF（中断允许标识）	中断允许标志用来决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求。CLI/STI 指令可对它进行赋值
10（0x400）	DF（方向标识）	决定在执行串操作指令（REP MOVSx、REP CMPSx、REP LODSx 和 REP SCASx）时有关指针寄存器发生调整的方向。CLD/STD 指令可对它进行赋值
11（0x800）	OF（溢出标识）	
12,13（0x3000）	IOPL（I/O 特权标识）80286	
14（0x4000）	NT（嵌套任务标志）80286	
16（0x10000）	RF（重启标识）80386	重启动标识用来控制是否接受调试。如果它的值为 1，那么 CPU 将忽略 DRx 中的硬件断点调试功能
17（0x20000）	VM（虚拟 8086 方式标志）80386	
18（0x40000）	AC（对准校验方式位）80486	
19（0x80000）	VIF（虚拟中断标志）Pentium	
20（0x100000）	VIP（虚拟中断未决标志）Pentium	
21（0x200000）	ID（标识标志）Pentium	

其余的标识位都是保留标识位。

A.3 FPU 寄存器

FPU 栈由 8 个 80 位寄存器构成，这 8 个寄存器分别叫作 ST（0）～ST（7）。IDA 把 ST（0）显示为 ST。FPU 寄存器用于存储符合 IEEE 754 标准的 long double 型数据。这种数据的格式如下表所示。

第 79 位	第 78-64 位	第 63 位	第 62-0 位
符号位	指数位	整数位	尾数（小数）位

A.3.1 控制字寄存器（16位）

FPU的控制字（Control Word）用于控制FPU的行为。

位	缩写（及含义）	描　述
0	IM（无效操作掩码）	
1	DM（操作数规格异常掩码）	
2	ZM（除数为0的掩码）	
3	OM（上溢/溢出掩码）	
4	UM（下溢/溢出掩码）	
5	PM（精度异常掩码）	
7	IEM（异常中断位/软件处理控制位）	第0～5位掩码控制功能的总开关，现在的FPU已经不可对其赋值。若IEM为0，则由FPU处理所有的异常信息，从而对软件屏蔽了所有的错误信息。默认值为1
8，9	PC（精度控制）	00：IEEE单精度24位（REAL4） 01：保留 10：IEEE双精度53位（REAL8） 11：IEEE扩展双精度64位（REAL10）
10，11	RC（舍入控制）	00：就近舍入（默认） 01：向−∞舍入 10：向+∞舍入 11：截断（向零舍入）
12	IC（无限/∞控制位）	0：按照unsigned处理±∞（初始态） 1：按照signed处理∞

若PM、UM、OM、ZM、DM、IM字段（第0～5位）设置为1，则由FPU处理异常信息（对软件屏蔽了错误信息）；若某位设置为0，则FPU将会在遇到相应异常时进行中断、释放异常信息给应用程序，程序在处理之后再把控制权返还给FPU。

A.3.2 状态字寄存器（16位）

FPU的状态寄存器又称Fstate，属于只读寄存器。

位序	缩写（及含义）	描　述
15	B（忙）	1：FPU正在进行运算 0：FPU可进行下次运算
14	C3（条件代码位C3）	
13，12，11	TOP（栈顶指针）	ST（0）使用的物理寄存器
10	C2（条件代码位C2）	
9	C1（条件代码位C1）	
8	C0（条件代码位C0）	
7	IR（中断请求）	
6	SF（栈异常）	
5	P（精度）	
4	U（下溢）	
3	O（上溢）	
2	Z（运算结果为零）	
1	D（操作数规格异常）	
0	I（无效操作）	

状态位 SF、P、U、O、Z、D、I 用于异常反馈。

有关 C3、C2、C1、C0 的更详细介绍，请参见本书的 17.7.1 节。

在软件使用 st（x）时，FPU 会计算 x 与栈顶指针序号的和，必要的时候还会再计算 8 的模（余数），以此确定栈指针的物理寄存器地址。

A.3.3 标记字寄存器（16 位）

标志字寄存器总共 16 位。每 2 位为一组，表示 FPU 数据寄存器的使用情况。

位　序	描　述
15，14	Tag（7）
13，12	Tag（6）
11，10	Tag（5）
9，8	Tag（4）
7，6	Tag（3）
5，4	Tag（2）
3，2	Tag（1）
1，0	Tag（0）

Tag（x）存储着 FPU 物理寄存器 R（x）[①]的状态码。

其各值的代表含义是：

- 00：该寄存器存储着非零的值。
- 01：该寄存器存储的值为零。
- 10：寄存器的值为特殊的值，NAN、∞或者无效操作数。
- 11：寄存器为空。

A.4 SIMD 寄存器

A.4.1 MMX 寄存器

MMX 寄存器由 8 个 64 位寄存器（MM0～MM7）组成。

A.4.2 SSE 与 AVX 寄存器

SSE 都有 XMM0～XMM7 这 8 个 128 位寄存器，x86-64 系统还有额外的 8 个寄存器（XMM8～XMM15）。而支持 AVX 指令集的 CPU，它们把 XMM＊寄存器扩充为 256 位寄存器。

A.5 FPU 调试寄存器

调试寄存器（Debugging registers）用于实现基于硬件的断点控制。

- DR0 为第 1 个断点的地址（线性地址）。
- DR1 为第 2 个断点的地址。
- DR2 为第 3 个断点的地址。
- DR3 为第 4 个断点的地址。

① 请注意，Tag（x）描述的不是 FPU 逻辑寄存器 ST（x）的状态。

- DR6 为调试状态寄存器。在调试过程异常时，它负责报告产生异常的原因。
- DR7 为用于控制断点调试。

A.5.1 DR6 规格

位序（掩码）	描　述
0（1）	B0：触发了断点 DR0
1（2）	B1：触发了断点 DR1
2（4）	B2：触发了断点 DR2
3（8）	B3：触发了断点 DR3
13（0x2000）	BD：仅在 DR7 的 GD 为 1 的情况下有效。只有当下一条指令要访问到某一个调试寄存器的时候，BD 位才被置位（1）
14（0x4000）	BS：当进行单步调试的时候，即 EFLAGS 的 TF 标识位被置位的时候，BS 才被置位。单步调试具有最高的调试优先级，不受其他标识位影响
15（0x8000）	BT：任务切换标识位

单步调试断点是在执行一条指令之后发生的断点。设置 EFLAGS（附录 A.2.19）的 TF 标识，即可实现单步调试。

A.5.2 DR7 规格

DR7 用于控制断点类型。

位序（掩码）	描　述
0（1）	L0：在当前任务的 DR0 处设置断点
1（2）	G0：在所有任务中都设置 DR0 的断点
2（4）	L1：在当前任务的 DR1 处设置断点
3（8）	G1：在所有任务中都设置 DR1 的断点
4（0x10）	L2：在当前任务的 DR2 处设置断点
5（0x20）	G2：在所有任务中都设置 DR2 的断点
6（0x40）	L3：在当前任务的 DR3 处设置断点
7（0x80）	G3：在所有任务中都设置 DR3 的断点
8（0x100）	LE：P6 以及 P6 以后的处理器不支持这个标识位。如果被置位，那么 FPU 将会在当前任务中追踪精确的数据断点
9（0x200）	GE：P6 以及 P6 以后的处理器不支持这个标识位。如果被置位，那么 FPU 将会在所有任务中追踪精确的数据断点
13（0x2000）	GD：如果置位，那么当 MOV 指令修改 DRx 寄存器的值时，FPU 将进行进行异常处理
16，17（0x3000）	断点 DR0 的触发条件
18，19（0xC000）	断点 DR0 的断点长度
20，21（0x30000）	断点 DR1 的触发条件
22，23（0xC0000）	断点 DR1 的断点长度
24，25（0x300000）	断点 DR2 的触发条件
26，27（0xC00000）	断点 DR2 的断点长度
28，29（0x3000000）	断点 DR3 的触发条件
30，31（0xC000000）	断点 DR3 的断点长度

其中，断点 DRx 的触发条件又分为：

- 00：执行指令。
- 01：数据的写操作。
- 10：读写 I/O（User mode 下不可用）。
- 11：读写数据。

可见，FPU 断点的触发条件里没有“读取数据”这一项。

FPU 断点长度的规格如下：

- 00：1 个字节。
- 01：2 个字节。
- 10：32 位系统中未定义，64 位系统中代表 8 字节。
- 11：4 个字节。

A.6 指令

标记为（M）的指令通常都不是编译器生成的指令。这种指令或许属于手写出来的汇编代码，或许属于编译器的内部指令（参见本书第 90 章）。

本节仅列举那些常见指令。如果需要查看完整的指令说明，请参见《Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1，2A，2B，2C，3A，3B，and 3C》和《AMD64 Architecture Programmer's Manual》。

我们是不是也要记住指令的 opcode 呢？除非您专门从事给代码打补丁的工作（参见本书第 89 章第 2 节），否则没那种必要。

A.6.1 指令前缀

LOCK：数据总线封锁前缀。在执行 LOCK 作前缀的汇编指令时，它可起到独占数据总线的作用。简单地说，在执行这种指令时，多处理器的其他 CPU 都将停下来、等该指令执行结束。这种指令常见于各种关键系统、（硬件）信号量和互斥锁。

禁止协处理器修改数据总线上的数据，起到独占总线的作用。该指令的执行不影响任何标志位。它常作为 ADD、AND、BTR、BTS、CMPXCHG、OR、XADD、XOR 指令的前缀。本书的第 68 章第 4 节详细介绍了这种指令。

REP：与 MOVSx 和 STOSx 指令结合使用，以循环的方式进行数据复制及数据存储。在执行 REP 指令时，CX/ECX/RCX 寄存器里存储的值将作为隐含的循环计数器。有关 MOVSx 和 STOSx 指令的详细说明，请参见附录 A.6.2。

REP 指令属于 DF 敏感指令。DF 标识位决定了它的操作方向。

REPE/REPNE：（又称为 REPZ/REPNZ）与 CMPSx 和 SCASx 指令结合使用，以循环的方式进行数值比较。在执行这种指令时，CX/ECX/RCX 寄存器里存储的值将作为隐含的循环计数器。当 ZF 标识位为 0（REPE），或 ZF 标识位为 1（REPNE）时，它将终止循环过程。

有关 CMPSx 和 SCASx 的详细描述，请参见附录 A.6.2 和 A.6.3。

REPE/REPNE 指令属于 DF 敏感指令。DF 标识位决定了它的操作方向。

A.6.2 常见指令

ADC（进位加法运算）：在进行加法运算时，会把 CF 标识位代表的进位加入和中。它常见于较大数值的加法运算。例如，在 32 位系统进行 64 位数值的加法运算时，会组合使用 ADD 和 ADC 指令，如下所示：

```
; 64 位值的运算：val2= val1 + val2.
; .lo 代表低 32 位，.hi 代表高 32 位。
```

```
ADD val1.lo, val2.lo
ADC val1.hi,val2.hi；会使用上一条指令设置的 CF
```

本书的第 24 章有更为详细的使用案例。

ADD：加法运算指令。

AND：逻辑“与”运算指令。

CALL：调用其他函数。相当于“PUSH（CALL 之后的返回地址）；JMP label”。

CMP：比较数值、设置标识位。虽然它的运算过程确是减法运算，但是 SUB 指令保存运算结果（差）、而 CMP 指令不保存运算结果。

DEC：递减运算。它不影响 CF 标识位。

IMUL：有符号数的乘法运算指令。

INC：递增运算。它不影响 CF 标识位。

JCXZ，JECXZ，JRCXZ（M）：当 CX/ECX/RCX=0 时跳转。

JMP：跳转到指定地址。相应的 opcode 中含有转移偏移量（jump offset）。

Jcc：条件转移指令。cc 是 condition code 的缩写。

JAE 即 JNC：（unsigned）在大于或等于的情况下进行跳转；转移条件是 CF=0。

JA 即 JNBE：（unsigned）在大于的情况下进行跳转；转移条件是 CF=0 且 ZF=0。

JBE：（unsigned）在小于或等于的条件下进行跳转；转移条件是 CF=1 或 ZF=1。

JB 即 JC：（unsigned）在小于的情况下进行跳转；转移条件是 CF=1

JC 即 JB：在小于的情况下进行跳转；转移条件是 CF=1。

JE 即 JZ：在相等的情况下进行跳转；转移条件是 ZF=1。

JGE：（signed）在大于或等于的情况下进行跳转；转移条件是 SF=OF。

JG：（signed）在大于的情况下进行跳转；转移条件是 ZF=0 且 SF=OF。

JLE：（signed）在小于或等于的情况下进行跳转；转移条件是 ZF=1 或 SF≠OF。

JL：（signed）在小于的条件下进行跳转；转移条件是 SF≠OF。

JNAE 即 JC：（unsigned）在小于（不大于且不相等）的情况下进行跳转；转移条件是 CF=1。

JNA：（unsigned）在不大于的情况下进行跳转；转移条件是 CF=1 或 ZF=1。

JNBE：（unsigned）在大于的情况下进行转移；转移条件是 CF=0 且 ZF=0。

JNB 即 JNC：（unsigned）在不小于的情况下进行跳转；转移条件是 CF=0。

JNC 即 JAE：等同于 JNB；转移条件是 CF=0。

JNE 即 JNZ：在不相等的情况下进行跳转；转移条件是 ZF=0。

JNGE：（signed）在不大于且不等于的情况下进行跳转；转移条件是 SF≠OF。

JNG：（signed）在不大于的情况下进行跳转；转移条件是 ZF=1 或 SF≠OF。

JNLE：（signed）在不大于且不相等的情况下进行跳转；转移条件是 ZF=0 且 SF=OF。

JNL：（signed）在不小于的情况下进行跳转；转移条件是 SF=OF。

JNO：在不溢出的情况下进行跳转；转移条件是 OF=0。

JNS：在 SF 标识位为 0 的情况下进行跳转。

JNZ 即 JNE：在不大于且不等于的情况下进行跳转；转移条件是 ZF=0

JO：在溢出的情况下进行跳转；转移条件是 OF=1。

JPO：在 PF 标识位为零的情况下进行跳转。

JP 即 JPE：在 PF 标识位为 1 的情况下进行跳转。

JS：在 SF 标识位为 1 的情况下进行跳转。

JZ 即 JE：在操作数相等的情况下进行跳转；转移条件是 ZF=1。

LAHF：标识位读取指令。它把标识位复制到 AH 寄存器。数权关系如下表所示。

7	6	5	4	3	2	1	0
SF	ZF		AF		PF		CF

LEAVE：等效于“MOV ESP，EBP”“POP EBP”指令的组合。即，这条指令释放当前子程序在堆栈中的局部变量，恢复栈指针（stack pointer/ESP）和 EBP 寄存器的初始状态。

LEA：有效（偏移）地址传送指令。

这个指令并非调用寄存器的值，也不会进行地址以外的求值运算。它可利用数组地址、元素索引号和元素空间进行混合运算，求得某个元素的有效地址。

所以，MOV 和 LEA 指令有巨大的差别：MOV 指令会把操作数的值当作地址、而后对这个地址的值进行读写操作；而 LEA 就对操作数的地址进行直接处理。

因此，LEA 指令也经常用于各种常规计算。

LEA 指令有一个重要的特点——它不会影响 CPU 标识位的状态。对于 OOE（乱序方式执行的指令）处理器来说，这一特性有利于大幅度降低数据依赖性。

```
int f(int a, int b)
{
        return a*8+b;
};
```

使用 MSVC 2010（启用优化功能）编译上述程序，可得到：

指令清单 A.1 优化 MSVC 2010

```
_a$ = 8                                                    ; size = 4
_b$ = 12                                                   ; size = 4
_f      PROC
        mov     eax, DWORD PTR _b$[esp-4]
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [eax+ecx*8]
        ret     0
_f      ENDP
```

Intel C++编译器生成的汇编代码更为烦琐。例如，在编译下述源代码时：

```
int f1(int a)
{
        return a*13;
};
```

Intel C++生成的汇编代码为：

指令清单 A.2 Intel C++2011

```
_f1     PROC NEAR
        mov       ecx, DWORD PTR [4+esp]            ; ecx = a
        lea       edx, DWORD PTR [ecx+ecx*8]   ; edx = a*9
        lea       eax, DWORD PTR [edx+ecx*4]   ; eax = a*9 + a*4 = a*13
        ret
```

即使如此，两条 LEA 指令的执行效率仍然超过了单条 IMUL 指令。

MOVSB/MOVSW/MOVSD/MOVSQ：复制 8 位单字节数据（Byte）/16 位 Word 数据（Word）/32 位双字型数据（Dword）/64 位四字型数据（Qword）的指令。默认情况下，它将把 SI/ESI/RSI 寄存器里的值当作源操作数的地址，目标操作数的地址将取自 DI/EDI/RDI 寄存器。

在与 REP 前缀组合使用时，它会把 CX/ECX/RCX 作为循环控制变量进行循环操作。这种情况下，它就像 C 语言中的 memcpy() 函数那样工作。如果编译器在编译阶段能够确定每个模块的大小，编译器通常

使用 REP MOVSx 指令以内连函数的形式实现 memcpy()。

例如，memcpy（EDI，ESI，15）等效于：

```
; copy 15 bytes from ESI to EDI
CLD         ; set direction to "forward"
MOV ECX, 3
REP MOVSD   ; copy 12 bytes
MOVSW       ; copy 2 more bytes
MOVSB       ; copy remaining byte
```

在复制 15 字节的内容时，从寄存器读取的操作效率来看，上述代码的效率要高于 15 次数据读写（MOVSB）的操作效率。

MOVSX：以符号扩展的方法实现 signed 型数据的类型转换（参见本书 15.1.1 节）。

MOVZX：以用零扩展的方法实现 unsigned 型数据的类型转换（参见本书 15.1.1 节）。

MOV：数据传送指令。“MOV”这个名字与“MOVE”（移动）拼写相似，不过它的功能是复制数据而非移动数据。在某些平台上，这条指令的名字是“LOAD”或者某个类似的名字。

值得一提的是，当使用 MOV 指令给 32 位寄存器的低 16 位赋值时，寄存器的高 16 位不会发生变化。而使用 MOV 指令给 64 位寄存器的低 32 位赋值时，寄存器的高 32 位会被清零。

64 位寄存器的高 32 位被自动清零的特性，可能是为了在 x86-64 系统上兼容 32 位程序而有意这样设计的。

MUL：unsigned 型数据的乘法运算指令。

NEG：求补指令（并非补码计算指令）。NEG op 可得到–op。

NOP：NOP 指令。在 x86 平台上的 opcode 是 0x90。这个 opcode 和 XCHG EAX，EAX 的空操作指令相同。这即是说，x86 平台没有 NOP 专用的汇编指令，而 RISC 平台上 NOP 有专用的汇编指令。有关这个指令的详细介绍，请参见本书的第 88 章。

编译器可能会使用 NOP 指令进行 16 字节边界对齐。此外，在手工修改程序时，人们也会使用 NOP 指令进行指令替换，用于屏蔽条件转移之类的汇编指令。

NOT：求反指令/逻辑“非”运算指令。

OR：逻辑“或”运算指令。

POP：出栈指令。它从 SS:［ESP］中取值，再执行 ESP=ESP+4（或 8）的操作。

PUSH：入栈指令。它先进行 ESP=ESP+4（或 8），再向地址 SS:［ESP］存储数据。

RET：子程序返回函数，相当于 POP tmp 或 JMP tmp。

实际上 RET 是汇编语言的宏。在 Windows 和*NIX 环境中，它会被解释为 RETN（“return near”）；在 MS-DOS 的寻址方式里，它被解释为 RETF（参见本书第 94 章）。

RET 指令可以有操作数。在这种情况下，它等同于 POP tmp、ADD ESP op1 及 JMP tmp。在符合调用约定 stdcall 的程序里，每个函数最后的 RET 指令通常都有相应的操作数。有关细节请参见本书的 64.2。

SAHF：标识传送指令。它把 AH 寄存器的值，复制到 CPU 到标识上。对应的数权关系如下表所示。

7	6	5	4	3	2	1	0
SF	ZF		AF		PF		CF

SBB：借位减法运算指令。计算出操作数间的差值之后，如果 CF 标识位为 1，则将差值递减。SBB 指令常见于大型数据的减法运算。例如，在 32 位系统上计算 2 个 64 位数据的差值时，编译器通常组合使用 SUB 和 SBB 指令：

```
; 计算两个 64 位数的差值：val1=val1-val2
; .lo 代表低 32 位； .hi 代表高 32 位
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; 使用了前一个指令设置的 CF 标识位
```

本书第 24 章还有更详细的介绍。

SCASB/SCASW/SCASD/SCASQ（M）：比较 8 位单字节数据（Byte）/16 位 Word 数据（Word）/32 位

双字型数据（Dword）/64 位四字型数据（Qword）的指令。它将 AX/EAX/RAX 寄存器里的值当作源操作数，另一个操作数则取自 DI/EDI/RDI 寄存器。在比较结果之后再设置标识位，设置标识位的方式和 CMP 指令相同。

这些指令通常与 REPNE 指令前缀组合使用。在这种情况下，组合指令将把寄存器 AX/EAX/RAX 里存储的值当作关键字，在缓冲区里进行搜索。此时，REPNE 里的 NE 就意味着：如果值不相等，则继续进行比较（搜索）。

这种指令常常用于实现 strlen() 函数，以判明 ASCIIZ 型字符串的长度。例如：

```
lea     edi, string
mov     ecx, 0FFFFFFFFh ; 扫描 2^32-1 bytes, 即, 接近“无限”
xor     eax, eax        ; 0 作终止符
repne scasb
add     edi, 0FFFFFFFFh ; 修正

; 现在，EDI 寄存器的值指向里 ASCIIZ 字符串的最后一个字符

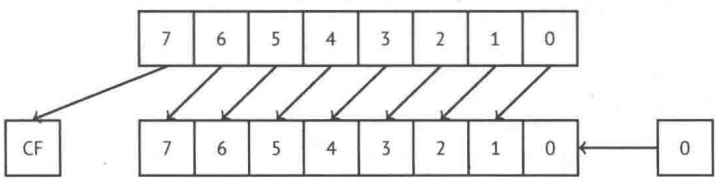
; 接下来将计算字符串的长度
; 目前 ECX = -1-strlen

not     ecx
dec     ecx

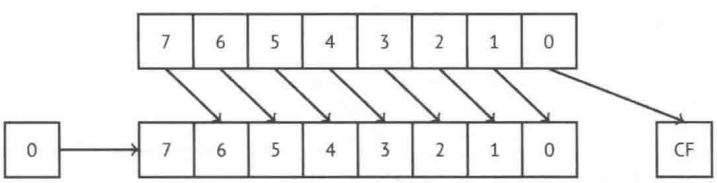
; 此后，ECX 的值是字符串的长度值
```

如果 AX/EAX/RAX 里存储的值不同，那么这个函数的功能就和标准 C 函数 memchr() 一致，即搜索特定 byte。

SHL：逻辑左移指令。



SHR：逻辑右移指令。



指令常用于乘以/除以 2^n^ 乘除运算。此外，它们还常见于字段处理的各种实践方法（请参见本书第 19 章）。

SHRD op1，op2，op3：双精度右移指令。把 op2 右移 op3 位，移位引起的空缺位由 op1 的相应位进行补足。详细介绍，请参见本书第 24 章。

STOSB/STOSW/STOSD/STOSQ：8 位单字节数据（Byte）/16 位 Word 数据（Word）/32 位双字型数据（Dword）/64 位四字型数据（Qword）的输出指令。它把 AX/EAX/RAX 的值当作源操作数，存储到 DI/EDI/RDI 为目的串地址指针所寻址的存储器单元中去。指针 DI 将根据 DF 的值进行自动调整。

在与 REP 前缀组合使用时，它将使用 CX/ECX/RCX 寄存器作循环计数器、进行多次循环；工作方式与 C 语言到 memset() 函数相似。如果编译器能够在早期阶段确定区间大小，那么将通过 REP MOVSx 指令实现 memset() 函数，从而减少代码碎片。

例如，memset（EDI，0xAA，15）对应的汇编指令是：

```
;在 EDI 中存储 15 个 0xAA
CLD                 ; set direction to "forward"
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD           ; write 12 bytes
STOSW               ; write 2 more bytes
STOSB               ; write remaining byte
```

在复制 15 字节的内容时，从寄存器读取的操作效率来看，上述代码的效率要高于 15 次数据读写（REP STOSB）的操作效率。

SUB：减法运算指令。常见的“SUB 寄存器，寄存器”指令可进行寄存器清零。

TEST：测试指令。在设置标识位方面，它和 AND 指令相同，但是它不存储逻辑与的运算结果。详细介绍请参见本书的第 19 章。

XCHG：数据交换指令。

XOR op1，op2：逻辑异或运算指令。常见的“XOR reg，reg”用于寄存器清零。

XOR 指令普遍用于“翻转”特定比特位。无论以哪个操作符作为源数据，只要另外一个操作符（参照数据）的指定位为 1，那么运算结果里的那一位数据都是源数据相应位的非值。

输入 A	输入 B	运 算 结 果
0	0	0
0	1	1
1	0	1
1	1	0

另外，如果参照数据的对应位为 0，那么运算结果里的那一位数据都是源数据相应位的原始值。这是 XOR 操作非常重要的特点，应当熟练掌握。

A.6.3 不常用的汇编指令

BSF：顺向位扫描指令。详情请参见本书 25.2 节。

BSR：逆向位扫描指令。

BSWAP：重新整理字节次序的指令。它以字节为单位逆序重新排列字节序，用于更改数据的字节序。

BTC：位测试并取反的指令。

BTR：位测试并清零的指令。

BTS：位测试并置位的指令。

BT：位测试指令。

CBW/CWD/CWDE/CDQ/CDQE：signed 型数据的类型转换指令：

- CBW：把 AL 中的字节（byte）型数据转换为字（word）型数据，存储于 AX 寄存器。
- CWD：把 AX 中的字（word）型数据转换为双字（Dword）型数据、存储于 DX-AX 寄存器对。
- CWDE：把 AX 中的字（word）型数据转换为双字字（Dword）型数据，存储于 DAX 寄存器。
- CDQ：把 EAX 中的双字（word）型数据转换为四字（Qword）型数据，存储于 EDX：EAX 寄存器对。
- CDQE（x64 指令）：把 EAX 中对双字（Dword）型数据转换为四字（Qword）型数据，并存储于 RAX 寄存器。

上述五个指令均能正确处理 signed 型数据中对符号位、对高位进行正确的填补。详情请参见本书的第 24 章第 5 节。

CLD 清除 DF 标识位。

CLI（M）：清除 IF 标识位。

CMC（M）：变换 CF 标识位。

CMOVcc：条件赋值指令。如果满足相应的条件代码（cc），则进行赋值。有关条件代码 cc 的各种代表意，请参见前文 A.6.2 对 Jcc 的详细说明。

CMPSB/CMPSW/CMPSD/CMPSQ（M）：比较 8 位单字节数据（Byte）/16 位 Word 数据（Word）/32 位双字型数据（Dword）/64 位四字型数据（Qword）的指令。它将 SI/ESI/RSI 寄存器里的值当作源操作数的地址，另一个操作数的地址则取自 DI/EDI/RDI 寄存器。在比较结果之后再设置标识位，设置标识位的方式和 CMP 指令相同。

这些指令通常与指令前缀 REP 组合使用。在这种情况下，组合指令将 CX/ECX/RCX 寄存器的值当作循环计数器进行多次循环比较，直至 ZF 标识位为零。也就是说，它也常用作字符比较（或搜索）。

它的工作方式和 C 语言的 memcmp() 函数相同。

以 Windows NT 内核（WindowsResearchKernel v1.2）为例，base\ntos\rtl\i386\movemem.asm 代码如下所示。

指令清单 A.3 base\ntos\rtl\i386\movemem.asm

```
; ULONG
; RtlCompareMemory (
;     IN PVOID Source1,
;     IN PVOID Source2,
;     IN ULONG Length
;     )
;
; Routine Description:
;
;     This function compares two blocks of memory and returns the number
;     of bytes that compared equal.
;
; Arguments:
;
;     Source1 (esp+4) - Supplies a pointer to the first block of memory to
;         compare.
;
;     Source2 (esp+8) - Supplies a pointer to the second block of memory to
;         compare.
;
;     Length (esp+12) - Supplies the Length, in bytes, of the memory to be
;         compared.
;
; Return Value:
;
;     The number of bytes that compared equal is returned as the function
;     value. If all bytes compared equal, then the length of the original
;     block of memory is returned.
;
;--

RcmSource1      equ     [esp+12]
RcmSource2      equ     [esp+16]
RcmLength       equ     [esp+20]

CODE_ALIGNMENT
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

        push    esi                             ; save registers
        push    edi
        cld                                     ; clear direction
        mov     esi,RcmSource1                  ; (esi) -> first block to compare
        mov     edi,RcmSource2                  ; (edi) -> second block to compare

;
```

```
;    Compare dwords, if any.
;

rcm10:  mov     ecx,RcmLength                   ; (ecx) = length in bytes
        shr     ecx,2                           ; (ecx) = length in dwords
        jz      rcm20                           ; no dwords, try bytes
        repe    cmpsd                           ; compare dwords
        jnz     rcm40                           ; mismatch, go find byte

;
; Compare residual bytes, if any.
;

rcm20:  mov     ecx,RcmLength                   ; (ecx) = length in bytes
        and     ecx,3                           ; (ecx) = length mod 4
        jz      rcm30                           ; 0 odd bytes, go do dwords
        repe    cmpsb                           ; compare odd bytes
        jnz     rcm50                           ; mismatch, go report how far we got

;
; All bytes in the block match.
;

rcm30:  mov     eax,RcmLength                   ; set number of matching bytes
        pop     edi                             ; restore registers
        pop     esi                             ;
        stdRET  _RtlCompareMemory

;
; When we come to rcm40, esi (and edi) points to the dword after the
; one which caused the mismatch.  Back up 1 dword and find the byte.
; Since we know the dword didn't match, we can assume one byte won't.
;

rcm40:  sub     esi,4                           ; back up
        sub     edi,4                           ; back up
        mov     ecx,5                           ; ensure that ecx doesn't count out
        repe    cmpsb                           ; find mismatch byte

;
; When we come to rcm50, esi points to the byte after the one that
; did not match, which is TWO after the last byte that did match.
;

rcm50:  dec     esi                             ; back up
        sub     esi,RcmSource1                  ; compute bytes that matched
        mov     eax,esi                         ;
        pop     edi                             ; restore registers
        pop     esi                             ;
        stdRET  _RtlCompareMemory

stdENDP _RtlCompareMemory
```

当内存块的尺寸是 4 的倍数时，这个函数将使用 32 位字型数据的比较指令 CMPSD，否则使用逐字比较指令 CMPSB。

CPUID：返回 CPU 信息的指令。详情请参见本书的 21.6.1 节。

DIV：无符号型数据的除法指令。

IDIV：有符号型数据的除法指令。

INT（M）：INT x 的功能相当于 16 位系统中的 PUSHF；CALL dwordptr[x*4]。在 MS-DOS 中，INT 指令普遍用于系统调用（syscall）。它调用 AX/BX/CX/DX/SI/DI 寄存器里的中断参数，然后跳转到中断向量表[①]。因为 INT 的 opcode 很短（2 字节），而且通过中断调用 MS-DOS 服务的应用程序不必去判断系统服务的入口地址，

① Interrupt Vector Table。它位于地址空间的开头部分，是实模式中断机制的重要组成部分，表中记录所有中断号对应的中断服务程序的内存地址。

所以 INT 指令曾盛行一时。中断处理程序通过使用 IRET 指令即可返回程序的控制流。

最常被调用的 MS-DOS 中断是第 0x21 号中断，它负责着大量的 API 接口。有关 MS-DOS 各中断的完整列表，请参见 Ralf Brown 撰写的《The x86 Interrupt List》[①]。

在 MS-DOS 之后，早期的 Linux 和 Windows（参见本书第 66 章）系统仍然使用 INT 指令进行系统调用。近些年来，它们逐渐使用 SYSENTER 或 SYSCALL 指令替代了 INT 指令。

INT 3（M）：这条指令有别于其他的 INT 指令。它的 opcde 只有 1 个字节，即（0xCC），普遍用于程序调试。一般来说，调试程序就是在需要进行调试的断点地址写上 0xCC（opcode 替换）。当被调试程序执行 INT3 指令而导致异常时，调试器就会捕捉这个异常从而停在断点处，然后将断点处的指令恢复成原来指令。

在 Windows NT 系统里，当 CPU 执行这条指令时，系统将会抛出 EXCEPTION_BREAKPOINT 异常。如果运行了主机调试程序/host debugger，那么这个调试事件将会被主机调试程序拦截并处理；否则，Windows 系统将会调用系统上注册了的某个调试器/system debugger 进行响应。如果安装了 MSVS（Microsoft Visual Studio），在执行 INT3 时，Windows 可能会启动 MSVS 的 debugger，继而调试这个进程。这种调试方法改变了原程序的指令，容易被软件检测到。人们开发出了很多反调试技术，通过检查加载代码的完整性防止他人进行逆向工程的研究。

MSVC 编译器有 INT3 对应的编译器内部函数——__debugbreak()。[②]

Kernel32.dll 里还有 win32 的系统函数 DebugBreak()，专门执行 INT 3。[③]

IN（M）：数据输入指令，用于从外设端口读取数据。这个指令常用于 OS 驱动程序和 MS-DOS 的应用程序。详细介绍请参见本书的 78.3 节。

IRET：在 MS-DOS 环境中调用 INT 中断之后，IRET 指令负责返还中断处理程序（interrupt handler）。它相当于 POP tmp；POPF；JMP tmp。

LOOP（M）：递减计数器 CX/ECX/RCX，在计数器不为零的情况下进行跳转。

OUT（M）：数据输出指令，用于向外设端口传输数据。这个指令常用于 OS 驱动程序和 MS-DOS 的应用程序。详细介绍请参见本书的 78.3 节。

POPA（M）：从数据栈中读取（恢复）（R|E）DI、（R|E）SI、（R|E）BP、（R|E）BX、（R|E）DX、（R|E）CX、（R|E）AX 寄存器的值。

POPCNT：它的名称是“population count”的缩写。该指令一般翻译为“位 1 计数”。既是说，它负责统计有多少个“为 1 的位”。它的英文外号称为“hamming weight”和“NSA 指令”。这种外号来自于下述轶闻（Bruce Schneier《Applied Cryptography：Protocols，Algorithms，and Source Code in C》1994.）：

> This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as “population count.” It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I’ve heard this called the canonical NSA instruction, demanded by almost all computer contracts.

更多信息请参阅参考资料［Sch94］。

POPF：从数据栈中读取标识位，即恢复 EFLAGS 寄存器。

---

① http://www.cs.cmu.edu/~ralf/files.html。

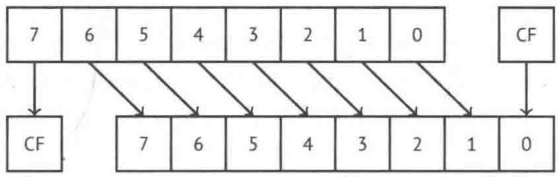
② 编译器内部函数指 compiler intrinsic，本书有详细介绍。它属于编译器有关的内部函数，基本不会被常规库函数调用。编译器为其生成特定的机械码，并非直接调用它的函数。内部函数常用于实现与特定 CPU 有关的伪函数。_debugbreak 的介绍请参见 http://msdn.microsoft.com/en-us/library/f408b4et.aspx。

③ 请参见 http://msdn.microsoft.com/en-us/library/windows/desktop/ms679297%28v=vs.85%29.aspx。

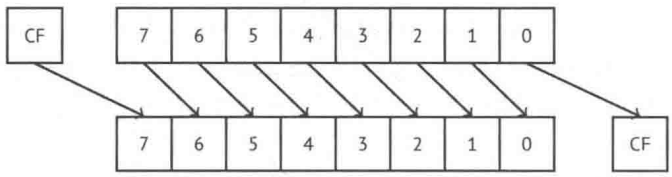
PUSHA（M）：把（R|E）AX、（R|E）CX、（R|E）DX、（R|E）BX、（R|E）BP、（R|E）SI、（R|E）DI寄存器的值，依次保存在数据栈里。

PUSHF：把标识位保存到数据栈里，即存储 EFLAGS 寄存器的值。

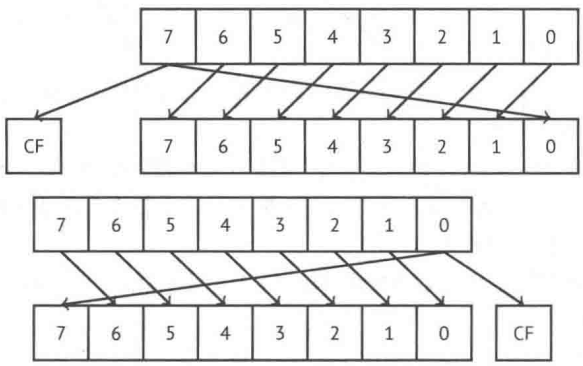
RCL（M）：带进位的循环左移指令，通过 CF 标识位实现。



RCR（M）：带进位的循环右移指令，通过 CF 标识位实现。



ROL/ROR（M）：循环左/右移。

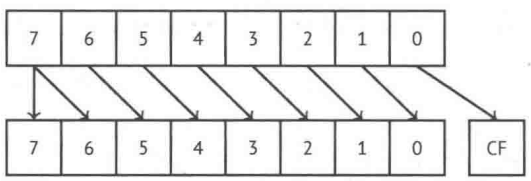


几乎所有的 CPU 都有这些循环位移指令。但是 C/C++语言里没有相应的操作指令，所以它们的编译器不会生成这些指令。

为了便于编程人员使用这些指令，至少 MSVC 提供了相应的伪函数（complier intrinsics）_rotl() 和_rotr()，可直接翻译这些指令。详情请参见：http://msdn.microsoft.com/en-us/library/5cc576c4.aspx。

SAL：算术左移指令，等同于逻辑左移 SHL 指令。

SAR：算术右移指令。



本指令通常用于对带符号数减半的运算中，因而在每次右移时，它保持最高位（符号位）不变，并把最低位右移至 CF 中。

SETcc op：在条件表达式 cc 为真的情况下，将目标操作数设置为 1；否则设置目标操作数为 0。这里的目标操作数指向一个字节寄存器（也就是 8 位寄存器）或内存中的一个字节。状态码后缀（cc）指代条

件表达式，可参见附录 A.6.2 的有关介绍。

STC（M）：设置 CF 标识位的指令。

STD（M）：设置 DF 标识位的指令。编译器不会生成这种指令，因此十分罕见。我们可以在 Windows 的内核文件 ntoskrnl.exe 中找到这条指令，也可以在手写的内存复制的汇编代码里看到它。

STI（M）：设置 IF 标识位的指令。

SYSCALL：（AMD）系统调用指令（参见本书第 66 章）。

SYSENTER：（Intel）系统调用指令（参见本书第 66 章）。

UD2（M）：未定义的指令，会产生异常信息，多用于软件测试。

A.6.4 FPU 指令

FPU 指令有很多带有“-R”或“-P”后缀的派生指令。带有 R 后缀的指令，其操作数的排列顺序与常规指令相反。带有 P 后缀的指令，在运行计算功能后，会从栈里抛出一个数据；而带有 PP 后缀的指令则最后抛出两个数据。P/PP 后缀的指令可在计算后释放栈里存储的计算因子。

FABS：计算 ST（0）绝对值的指令。ST(0)=fabs(ST(0))。

FADD op：单因子加法运算指令。ST(0)=op+ST(0)。

FADD *ST（0），ST（i）*：加法运算指令。ST(0)=ST(0)+ST(i)。

FADDP *ST（1）*：相当于 ST(1)=ST(0)+ST(1)；pop。求和之后，再从数据栈里抛出 1 个因子。即，使用计算求得的“和”替换计算因子。

FCHS：求负运算指令。ST(0)= −1xST(0)

FCOM：比较 ST（0）和 ST（1）。

FCOM *op*：比较 ST（0）和 op。

FCOMP：比较 ST（0）和 ST（1）；然后执行 1 次出栈操作。

FCOMPP：比较 ST（0）和 ST（1）；然后执行 2 次出栈操作。

FDIVR *op*：ST(0)=op/ST(0)。

FDIVR *ST（i），ST（j）*：ST(i)=ST(j)/ST(i)。

FDIVRP *op*：ST(0)=op/ST(0)，然后执行 1 次出栈操作。

FDIVRPP *ST（i），ST（j）*：ST(i)=ST(j)/ST(i)，然后执行 2 次出栈操作。

FDIV *op*：ST(0)=ST(0)/op。

FDIV *ST（i），ST（j）*：ST(i)=ST(i)/ST(j)。

FDIVP：ST(1)=ST(0)/ST(1)，然后执行 1 次出栈操作。即，被除数替换为商。

FILD *op*：将整数转化为长精度数据，并存入 ST（0）的指令。

FIST *op*：将 st（0）以整数保存到 op。

FISTP *op*：将 st（0）以整数保存到 op，再从栈里抛出 ST（0）。

FLD1：把 1 推送入栈。

FLDCW *op*：从 16 位的操作数 op 里提取 FPU 控制字（参见附录 A.3）。

FLDZ：把 0 推送入栈。

FLD *op*：把 op 推送入栈。

FMUL *op*：ST(0)＝ST(0)*op。

FMUL *ST（i），ST（j）*：ST(i)=ST(i)*ST(j)。

FMULP *op*：ST(0)＝ST(0)*op；然后执行 1 次出栈操作。

FMULP *ST（i），ST（j）*：ST(i)=ST(i)*ST(j)；然后执行 1 次出栈操作。

FSINCOS：一次计算 Sine 和 Cosine 结果的指令。

调用指令时，ST（0）存储着角度参数 tmp。ST(0)=sin(tmp)；PUSH ST(0)（即 ST1 存储 Sin 值）；之后

再计算 ST(0)=cos(tmp)。

FSQRT：$ST(0) = \sqrt{ST(0)}$。

FSTCW *op*：检查尚未处理的、未被屏蔽的浮点异常，再将 FPU 的控制字（参见附录 A.3）保存到 op。

FNSTCW *op*：将 FPU 的控制字（参见附录 A.3）直接保存到 op。

FSTSW *op*：检查尚未处理的、未被屏蔽的浮点异常，再将 FPU 的状态字（参见附录 A.3）保存到 op。

FNSTSW *op*：将状态字（参见附录 A.3）直接保存到 op。

FST *op*：保存实数 ST(0) 到 op。

FSTP *op*：将 ST(0) 复制给 op，然后执行 1 次出栈操作（ST(0)）。

FSUBR *op*：ST(0)=op−ST(0)。

FSUBR *ST（0），ST（i）*：ST(0)=ST(i)−ST(0)。

FSUBRP：ST(1)=ST(0)−ST(1)，然后执行 1 次出栈操作。即被减数替换为差。

FSUB *op*：ST(0)=ST(0)−op。

FSUB *ST（0），ST（i）*：ST(0)=ST(0)−ST(i)。

FUSBP *ST（1）*：ST(1)=ST(1)−ST(0)。

FUCOM *ST（i）*：比较 ST（0）和 ST（i）。

FUCOM：比较 ST（0）和 ST（1）。

FUCOMP：比较 ST（0）和 ST（1），然后执行 1 次出栈操作。

FUCOMPP：比较 ST（0）和 ST（1），然后从栈里抛出 2 个数据。

上述两个指令与 FCOM 的功能相似，但是它们在处理 QNaN 型数据时不会报错，仅在处理 SNaN 时进行异常汇报。

FXCH *ST（i）*：交换 ST（0）和 ST（i）的数据。

FXCH：交换 ST（0）和 ST（1）的数据。

A.6.5 可屏显的汇编指令（32 位）

在构建 Shellcode 时（参见本书第 82 章），可能会用到下面这个速查表。

ASCII 字符	16 进制码	x86 指令
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3A	CMP
;	3B	CMP
<	3C	CMP
=	3D	CMP
?	3F	AAS
@	40	INC
A	41	INC

续表

ASCII 字符	16 进制码	x86 指令
B	42	INC
C	43	INC
D	44	INC
E	45	INC
F	46	INC
G	47	INC
H	48	DEC
I	49	DEC
J	4A	DEC
K	4B	DEC
L	4C	DEC
M	4D	DEC
N	4E	DEC
O	4F	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH
T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5A	POP
[	5B	POP
\	5C	POP
]	5D	POP
^	5E	POP
_	5F	POP
`	60	PUSHA
a	61	POPA
f	66	32 位运行模式下，把操作数切换为 16 位
g	67	32 位运行模式下，把操作数切换为 16 位
h	68	PUSH
i	69	IMUL
j	6a	PUSH
k	6b	IMUL
p	70	JO
q	71	JNO
r	72	JB
s	73	JAE
t	74	JE

续表

ASCII 字符	16 进制码	x86 指令
u	75	JNE
v	76	JBE
w	77	JA
x	78	JS
y	79	JNS
z	7A	JP

总之，存在对应 ASCII 字符的指令有 AAA、AAS、CMP、DEC、IMUL、INC、JA、JAE、JB、JBE、JE、JNE、JNO、JNS、JO、JP、JS、POP、POPA、PUSH、PUSHA 和 XOR。

附录 B　ARM

B.1　术语

初代 ARM 处理器就是 32 位 CPU。所以它的 word 型数据都是 32 位数据。这方面它和 x86 有所不同。

byte	8 位数据。声明 byte 型数组和变量的指令是 DB。
halfword	16 位数据。声明 halfword 型数组和变量的指令是 DCW。
word	32 位数据。声明 word 型数组和变量的指令是 DCD。
doubleword	64 位数据。
quadword	128 位数据。

B.2　版本差异

- **ARMv4**：开始支持 Thumb 模式的指令。
- **ARMv6**：用于第一代 iPhone、iPhone 3G（这些设备的处理器采用了 Samsung 32 位 RISC ARM 1176IZ（F）-S，支持 Thumb-2 指令）。
- **ARMv7**：实现了 Thumb-2 指令（2003）。用于 iPhone 3GS、iPhone 4、iPad 1（ARM Crotex-A8）、iPad 2（Cortex-A9）和 iPad 3。
- **ARMv7s**：添加了新的指令。用于 iPhone 5、iPhone 5c 和 iPad 4（Apple A6）。
- **ARMv8**：64 位 CPU，又叫作 ARM64、AArch64。用于 iPhone 5S、iPad Air（Apple A7）。64 位平台不支持 Thumb 模式，只支持 ARM 模式的指令（4 字节指令）。

B.3　32 位 ARM（AArch32）

B.3.1　通用寄存器

- R0。函数的返回结果通常通过 R0 传递。
- R1～R12。通用寄存器 GPRs。
- R13。SP（Stack Pointer/栈指针）。
- R14。LR（link register/链接寄存器）。
- R15。PC（程序计数器）。

R0～R3 也被称为临时寄存器（scratch registers）。它们通常用于传递函数的参数；在函数结束时，也不必恢复它们的初始状态。

B.3.2　程序状态寄存器/CPSR

CPSR 的全称是 Current Program Status Register，其规格如下表所示。

位　序	描　述
0～4	M－处理器模式控制位
5	T－Thumb 模式控制位
6	F－FIQ 禁止位
7	I－IRQ/中断禁止位
8	A－imprecise data abort disable
9	E－数据字节序状态位
10..15，25，26	IT－if-then 状态位
16..19	GE－greater-than-or-equal-to
20..23	DNM－Do Not Modify
24	J－Java state
27	Q－sticky overflow 表达式整体溢出（相对于各独立因子溢出）标识位
28	V－Overflow 溢出标识位
29	C－进位/借位/扩展位
30	Z－零标识位
31	N－负号位/小于标识位

B.3.3 VFP（浮点）和 NEON 寄存器

<table>
<tr><td>0～31 bits</td><td>32～64</td><td>65～96</td><td>97～127</td></tr>
<tr><td colspan="4">Q0 128bits</td></tr>
<tr><td colspan="2">D0 64bits</td><td colspan="2">D1</td></tr>
<tr><td>S0 32bits</td><td>S1</td><td>S2</td><td>S3</td></tr>
</table>

S 寄存器为 32 位寄存器，用于存储单精度数。

D 寄存器位 64 位寄存器，用于存储双精度数。

D 寄存器和 S 寄存器的存储地址相通。程序可以通过 S 寄存器的助记符访问 D 寄存器的数据。虽然这种操作毫无实际意义，但是确实可行。

与之相似的是，NEON 的 Q 寄存器是 128 位寄存器，与其他浮点数寄存器共用 CPU 的物理存储空间。

VFP 里有 32 个 S 寄存器，即 S0～S31。VFP v2 里又新增了 16 个 D 寄存器，实际上这些 D 寄存器用的还是 S0～S31 的存储空间。

再后来，VFPv3（NEON 或者叫作“Advanced SIMD”）再次添加了 16 个 D 寄存器，形成了 D0～D31。这几个新增的寄存器，即 D16～D31 寄存器的存储空间，终于和 S 寄存器的存储空间相互独立。

NEON 或 Advanced SIMD 也有 16 个 128 位 Q 寄存器，它们使用的是 D 寄存器（D0～D31）的存储空间。

B.4 64 位 ARM（AArch64）

通用寄存器

64 位 ARM 的寄存器总数，是 32 位 ARM 的两倍之多。

- X0。常用于传递函数运算结果。
- X0～X7。传递函数参数。
- X8。

- X9～X15。临时寄存器，供被调用方函数使用，函数退出时无需恢复。
- X16。
- X17。
- X18。
- X19～X29。被调用方函数使用的寄存器，在函数退出时需要恢复原值。
- X29。用作帧指针/FP（至少 GCC 把它当作 FP）。
- X30。链接寄存器/LR（link register）。
- X31。这个寄存器一直为 0，又称为 XZR、“零寄存器”。它的 32 位（word 型）部分还叫作 WZR。
- SP，64 位系统有单独的栈指针寄存器，不再使用通用寄存器。

详细内容请参见官方的 ARM 资料《Procedure Call Standard for the ARM 64-bit Architecture（AArch64）》（http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf）。

程序还可通过其 W 寄存器这样的助记符访问 64 位 X 寄存器的低 32 位空间。W 寄存器和 X 寄存器的对应关系如下表所示。

<table>
<tr><td>高 32 位部分</td><td>低 32 位部分</td></tr>
<tr><td colspan="2">X0</td></tr>
<tr><td></td><td>W0</td></tr>
</table>

B.5　指令

ARM64 的部分指令存在对应的、带 S 后缀的指令。这个后缀代表着该指令会根据结果设置相应的标识位，而不带 S 后缀的指令就不设置标识位。以 ADD 和 ADDS 指令为例，前者只进行数学运算而不会设置标识位。这种功能划分，为 ARM 系统的预处理功能提供了可靠的保证，绝不会意外地影响到 CMP 指令和其他条件转移需要的各标识位。

Conditional codes 速查表

代　码	描　述	标 识 位
EQ	相等	Z==1
NE	相异	Z==0
CS/HS	Carry 置位/Unsigned，大于或等于	C==1
CC/LO	Carry 为零/Unsigned，小于	C==0
MI	Minus，负数/小于	N==1
PL	Plus，正数或零/大于或等于	N==0
VS	溢出	V==1
VC	未溢出	V==0
HI	Unsigned 大于	C==1 and Z==0
LS	Unsigned 小于或等于	C==0 or Z==1
GE	Signed 大于或等于	N==V
LT	Signed 小于	N !=V
GT	Signed 大于	Z==0 and N==V
LE	Signed 小于或等于	Z==1 or N !=V
None/AL	Always	Any

附录 C　MIPS

C.1　寄存器

MIPS 遵循的调用约定是 O32。

C.1.1　通用寄存器 GPR

编　号	别　名	描　述
$0	$ZERO	总是为零。给这个寄存器赋值到操作相当于 NOP
$1	$AT	汇编宏和伪指令使用到临时寄存器
$2～$3	$V0～$V1	用于传递函数返回值
$4～$7	$A0～$A3	用于传递函数的参数
$8～$15	$T0～$T7	可供临时数据使用
$16～$23	$S0～$S7	可供临时数据使用，被调用方函数必须保全
$24～$25	$T8～$T9	可供临时数据使用
$26～$27	$K0～$K1	OS Kernel 保留
$28	$GP	全局指针/Global Pointer，被调用方函数必须保全 PIC code 以外的值
$29	$SP	栈指针/Stack Pointer
$30	$FP	帧指针/Frame Pointer
$31	$RA	返回地址/Return Address
n/a	PC	PC
n/a	HI	专门存储商或积的高 32 位，可通过 MFHI 访问
n/a	LO	专门存储商或积的低 32 位，可通过 MFLO 访问

C.1.2　浮点寄存器 FPR

名　称	描　述
$F0～$F1	函数返回值
$F2～$F3	未被使用
$F4～$F11	用于临时数据
$F12～$F15	函数的前两个参数
$F16～$F19	用于临时数据
$F20～$F31	用于临时数据，被调用方函数必须保全

C.2　指令

MIPS 的指令分为以下 3 类：

- R-Type，Register/寄存器类指令。此类指令操作 3 个寄存器，具有以下形式：

```
指令目标寄存器，源寄存器 1，源寄存器 2
```

当前两个操作数相同时，IDA 可能会以以下形式进行显示：

```
指令目标寄存器/源寄存器 1，源寄存器 2
```

这种显示风格与 x86 汇编语言的 Intel 语体十分相似。

- I-Type，Immediate/立即数类指令。涉及 2 个寄存器和 1 个立即数。
- J-Type，Jump/转移指令。在 MIPS 转移指令的 opcode 里，共有 26 位空间可存储偏移量的信息。

转移指令

实现转移功能的指令可分为 B 开头的指令（BEQ，B 等）和 J 开头的指令（JAL，JALR 等）。这两种跳转指令的区别在哪里？

B-类转移指令属于 I-type 的指令。也就是说，B-指令的 opcode 里封装有 16 位立即数（偏移量）。而 J 和 JAL 属于 J-type 指令，它们的 opocde 里存有 26 位立即数。

简单地说，B 开头的转移指令可以把转移条件（cc）封装到 opcode 里（B 指令是“BEQ $ZERO, $ZERO, Label”的伪指令）。但是 J 开头的转移指令无法在 opcode 里封装转移条件表达式。

附录 D　部分 GCC 库函数

名　称	描　述
__divdi3	有符号型数据的除法运算
__moddi3	有符号型数据的求模运算（余数）
__udivdi3	无符号型数据的除法运算
__umoddi3	无符号型数据的求模运算（余数）

附录 E　部分 MSVC 库函数

在 MSVC 的库函数里，那些名称里带有“LL”的函数都是操作“long long”型数据、即 64 位数据的函数。

名　称	描　述
__alldiv	有符号数的除法运算
__allmul	乘法运算
__allrem	有符号数的求余运算
__allshl	左位移运算
__aulldiv	无符号数的除法运算
__aullrem	无符号数的求余运算
__aullshr	无符号数的右移运算

其中，乘法运算指令、左位移运算指令不区分有符号数和无符号数，所以此处的两条指令不再区分数据类型。

安装 MSVS 之后，你可以在库文件里找到上述函数的源代码。确切的文件位置是 VC/crt/src/intel/*.asm。

附录F 速 查 表

F.1 IDA

常用的IDA的快捷键如下表所示。

按 键	作 用
空格	切换显示方式
C	转换为代码
D	转换为数据
A	转换为字符
*	转换为数组
U	未定义
O	提取操作数的偏移量
H	把立即数转换为10进制数
R	把立即数转换为字符
B	把立即数转换为2进制数
Q	把立即数转换为16进制数
N	为标签重命名
?	计算器
G	跳转到地址
:	添加注释
Ctrl-X	查看当前函数、标签、变量的参考（显示栈）
X	查看当前函数、标签、变量的参考
Alt-I	搜索常量constant
Ctrl-I	再次搜索常量
Alt-B	搜索byte序列
Ctrl-B	再次搜索byte序列
Alt-T	搜索文本（包括指令中的文本）
Ctrl-T	再次搜索该文本
Alt-P	编辑当前函数
Enter	跳转到函数、变量等对象
Esc	返回
Num −	收缩（函数代码或区域代码）
Num +	展开

如果您明白函数的具体作用，就可以使用IDA的“收缩”功能把函数的代码隐藏起来。笔者编写的IDA脚本程序（https://github.com/yurichev/IDA_scripts）可自动隐藏经常使用的内联代码。

F.2 OllyDbg

OllyDbg 常用的快捷键如下表所示。

快 捷 键	作 用
F7	单步步入/单步调试
F8	单步步过
F9	执行至断点处
Ctrl－F2	重新启动程序

F.3 MSVC 选项

本书常用的 MSVC 选项如下表所示。

选 项	作 用
/O1	创建尺寸最小的文件
/Ob0	禁止内联展开
/Ox	启用最大优化
/GS-	禁用安全检查（缓冲区溢出）
/Fa(file)	创建汇编文件（设置文件名）
/Zi	生成完整的调试信息
/Zp(n)	使封装结构体向 *n* 字节边界对齐
/MD	令可执行程序使用 MSVCR*.DLL

有关 MSVC 的详细介绍，请参见本书的 55.1 节。

F.4 GCC

本书用到的 GCC 选项如下表所示。

选 项	作 用
-Os	优化目标文件大小
-O3	打开所有-O2 的优化选项
-regparm=	设定传递参数的寄存器的数量
-o file	指定输出的文件名
-g	产生带有调试信息的目标代码
-S	仅编译到汇编指令，不进行汇编和链接
-masm=intel	汇编指令采用 intel 语体
-fno-inline	禁止内联函数

F.5 GDB

本书用到的 GDB 指令如下表所示。

选　项	作　用
break filename.c:number	在源程序第 *n* 行处设置断点
break function	在函数入口处设置断点
break *address	在某地址设置断点
b	同 break
p variable	显示变量的值
run	运行
r	同 run
cont	继续运行
c	（同上）
bt	打印当前栈的所有信息
set disassembly-flavor intel	设置为 intel 语体
disas	查看当前函数程序的汇编指令
disas function	查看函数的汇编指令
disas function,+50 disas $eip,+0x10	查看函数的部分汇编指令
disas/r	查看接下来的几条指令
info registers	查看 opcode
info float	显示 CPU 各寄存器的值
info locals	显示 FPU 各寄存器的值
x/w ...	列出全部的局部变量（如果能识别出来）
x/w $rdi	读取内存，并显示为 word 型数据
x/10w ...	从 RDI 指定的地址读取数据，显示为 word
x/s ...	读取并显示 10 个 word 型数据
x/i ...	读取内存并显示为字符串
x/10c ...	读取内存并显示为汇编代码
x/b ...	读取内存，并显示 10 个字符
x/h ...	读取内存，并显示为 byte
x/g ...	读取并显示 16 位 halfword 型数据
finish	读取并显示 giant words（64 位）
next	执行到函数退出位置
step	单条语句执行指令，（不步入函数）
set step-mode on	单步步入的跟踪调试指令
frame n	打开 step-mode 模式
info break	切换栈帧
del n	查看断点
set args ...	删除断点

附录 G　练习题答案

G.1　各章练习

第 3 章

题目 1

对应章节：3.7.1 节

```
MessageBeep (0xFFFFFFFF);  // A simple beep. If the sound card is not available, the sound is ↙
    ↳ generated using the speaker.
```

题目 2

对应章节：3.7.2 节

```
#include <unistd.h>

int main()
{
        sleep(2);
};
```

第 5 章

题目 1

对应章节：5.5.1 节

如果未启用 MSVC 的优化编译功能，程序显示的数字分别是 EBP 的值、RA 和 argc。在命令行中执行相应的程序即可进行验证。

如果启用了 MSVC 的优化编译功能，程序显示的数字分别来自：返回地址 RA、argc 和数组 argv []。GCC 会给 main() 函数的入口分配 16 字节的地址空间，所以输出内容会有不同。

题目 2

对应章节：5.5.2 节

这些都是打印 UNIX 时间的程序，其源代码如下所示：

```
#include <unistd.h>

int main()
{
        sleep(2);
};
```

第 7 章

题目 1

对应章节：7.4.1 节

Linux 下的 GCC 将所有文本字符串信息放置到.rodata 数据段，它是显式只读的（“只读的数据”）：

```
$ objdump -s 1

...
Contents of section .rodata:
 400600 01000200 52657375 6c743a20 25730a00  ....Result: %s..
 400610 48656c6c 6f2c2077 6f726c64 210a00    Hello, world!..
```

第 13 章

题目 1

对应章节：13.5.1 节

提示：函数 printf() 只会被调用 1 次。

第 14 章

题目 3

对应章节：14.4.3 节

```
#include <stdio.h>

int main()
{
        printf ("%d\n", i);
};
```

题目 4

对应章节：14.4.4 节

```
#include <stdio.h>

int main()
{
        int i;
        for (i=1; i<100; i=i+3)
                printf ("%d\n", i);
};
```

第 15 章

题目 1

对应章节：15.2.1 节

这是一个统计 C 字符串中空格的程序，源代码如下：

```
#include <stdio.h>

int main()
{
        int i;
        for (i=100; i>0; i--)
                printf ("%d\n", i);
};
```

第 16 章

题目 1

对应章节：16.3.1 节

```
#include <stdio.h>
```

```
int main()
{
        int i;
        for (i=1; i<100; i=i+3)
                printf ("%d\n", i);
};
```

第 17 章

题目 2

对应章节：17.10.2 节

计算 5 个双精度浮点数的平均值，源代码如下：

```
int f(char *s)
{
        int rt=0;
        for (;*s;s++)
        {
                if (*s==' ')
                        rt++;
        };
        return rt;
};
```

第 18 章

题目 1

对应章节：18.9.1 节

两个 100x200 矩阵的双精度加法运算程序，其源代码如下：

```
#define M   100
#define N   200
void s(double *a, double *b, double *c)
{
  for(int i=0;i<N;i++)
    for(int j=0;j<M;j++)
      *(c+i*M+j)=*(a+i*M+j) + *(b+i*M+j);
};
```

题目 2

对应章节：18.9.2 节

答案：两个矩阵（一个是 100x200 矩阵，另一个是 100x300）的乘法运算程序，生成一个 100x300 的矩阵。其源代码如下所示：

```
#define M    100
#define N    200
#define P    300
void m(double *a, double *b, double *c)
{
  for(int i=0;i<M;i++)
    for(int j=0;j<P;j++)
    {
      *(c+i*M+j)=0;
      for (int k=0;k<N;k++) *(c+i*M+j)+=*(a+i*M+j) * *(b+i*M+j);
    }
};
```

题目 3

对应章节：18.9.3 节

```
double f(double array[50][120], int x, int y)
{
        return array[x][y];
};
```

题目 4

对应章节：18.9.4 节

```
int f(int array[50][60][80], int x, int y, int z)
{
       return array[x][y][z];
};
```

题目 5

对应章节：18.9.5 节

生成乘法九九表的程序

```
int tbl[10][10];

int main()
{
       int x, y;
       for (x=0; x<10; x++)
            for (y=0; y<10; y++)
                  tbl[x][y]=x*y;
};
```

第 19 章

题目 1

对应章节：19.7.1 节

这是一个改变 32 位数值字节序的函数，源代码如下：

```
unsigned int f(unsigned int a)
{
      return ((a>>24)&0xff) | ((a<<8)&0xff0000) | ((a>>8)&0xff00) | ((a<<24)&0xff000000);
};
```

题目 2

对应章节：19.7.2 节

这是一个把 BCD 封装的 32 位值转换为常规格式的函数，源代码如下：

```
#include <stdio.h>

unsigned int f(unsigned int a)
{
       int i=0;
       int j=1;
       unsigned int rt=0;
       for (;i<=28; i+=4, j*=10)
          rt+=((a>>i)&0xF) * j;
       return rt;
};
int main()
{
        // test
        printf ("%d\n", f(0x12345678));
        printf ("%d\n", f(0x1234567));
        printf ("%d\n", f(0x123456));
        printf ("%d\n", f(0x12345));
        printf ("%d\n", f(0x1234));
        printf ("%d\n", f(0x123));
        printf ("%d\n", f(0x12));
```

```
        printf ("%d\n", f(0x1));
};
```

题目 3

对应章节：19.7.3 节

```
#include <windows.h>

int main()
{
        MessageBox(NULL, "hello, world!", "caption",
                MB_TOPMOST | MB_ICONINFORMATION | MB_HELP | MB_YESNOCANCEL);
};
```

题目 4

对应章节：19.7.4 节

这个函数把两个 32 位数相乘，返回 64 位的积。解答这种题目，根据输入输出进行判断的速度比较快。

```
#include <stdio.h>
#include <stdint.h>

// source code taken from
//http://www4.wittenberg.edu/academics/mathcomp/shelburne/comp255/notes/binarymultiplication.pdf

uint64_t mult (uint32_t m, uint32_t n)
{
   uint64_t p = 0; // initialize product p to 0
   while (n != 0) // while multiplier n is not 0
   {
     if (n & 1) // test LSB of multiplier
          p = p + m; // if 1 then add multiplicand m
      m = m << 1; // left shift multiplicand
      n = n >> 1; // right shift multiplier
   }
   return p;
}

int main()
{
   printf ("%d\n", mult (2, 7));
   printf ("%d\n", mult (3, 11));
   printf ("%d\n", mult (4, 111));
};
```

第 21 章

题目 1

对应章节：21.7.1 节

这个程序将显示文件所有人的 user ID。其源代码如下所示：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   struct stat sb;

   if (argc != 2)
   {
      fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
```

```
        return 0;
    }
    if (stat(argv[1], &sb) == -1)
    {
        // error
        return 0;
    }

    printf("%ld\n",(long) sb.st_uid);
}
```

题目 2

对应章节：21.7.2 节

提示：研究的侧重点应当放在 Jcc、MOVSx 和 MOVZX 指令。

```
#include <stdio.h>

struct some_struct
{
        int a;
        unsigned int b;
        float f;
        double d;
        char c;
        unsigned char uc;
};

void f(struct some_struct *s)
{
         if (s->a > 1000)
         {
                 if (s->b > 10)
                 {
                                 printf ("%f\n", s->f * 444 + s->d * 123);
                                 printf ("%c, %d\n", s->c, s->uc);
                 }
                 else
                 {
                                 printf ("error #2\n");
                 };
         }
         else
         {
                 printf ("error #1\n");
         };
};
```

第 41 章

题目 1

对应章节：41.6 节

```
int f(int a)
{
        return a/661;
};
```

第 50 章

题目 1

对应章节：50.5.1 节

源代码：

```
#include <stdio.h>

int is_prime(int number)
{
    int i;
    for (i=2; i<number; i++)
    {
      if (number % i == 0 && i != number)
          return 0;
    }
    return 1;
}

int main()
{
    printf ("%d\n", is_prime(137));
};
```

G.2 初级练习题

G.2.1 练习题 1.1

这是一个返回 2 个数中最大值的函数。

G.2.2 练习题 1.4

程序源代码如下：

```
#include <stdio.h>

int main()
{
    char buf[128];

    printf ("enter password:\n");
    if (scanf ("%s", buf)!=1)
       printf ("no password supplied\n");
    if (strcmp (buf, "metallica")==0)
       printf ("password is correct\n");
    else
       printf ("password is not correct\n");
};
```

G.3 中级练习题

G.3.1 练习题 2.1

这个函数可根据牛顿法计算平方根，其算法摘自 Harold Abelson, Gerald Jay Sussman, and Julie Sussman：《Structure and Interpretation of Computer Programs》（1996）。

程序源代码如下所示：

```
// algorithm taken from SICP book

#include <math.h>

double average(double a, double b)
```

```
{
   return (a + b) / 2.0;
}

double improve (double guess, double x)
{
   return average(guess, x/guess);
}

int good_enough(double guess, double x)
{
   double d = abs(guess*guess - x);
   return (d < 0.001);
};

double square_root(double guess, double x)
{
   while(good_enough(guess, x)==0)
      guess = improve(guess, x);
   return guess;
};

double my_sqrt(double x)
{
   return square_root(1, x);
};

int main()
{
   printf ("%g\n", my_sqrt(123.456));
};
```

G.3.2 练习题 2.4

答案：函数 strstr()。其源代码如下所示：

```
char * strstr (
        const char * str1,
        const char * str2
        )
{
        char *cp = (char *) str1;
        char *s1, *s2;

        if ( !*str2 )
            return((char *)str1);

        while (*cp)
        {
                s1 = cp;
                s2 = (char *) str2;

                while ( *s1 && *s2 && !(*s1-*s2) )
                    s1++, s2++;

                if (!*s2)
                   return(cp);

                cp++;
        }

        return(NULL);

}
```

G.3.3 练习题 2.6

提示：使用 google 搜索程序中的常量。

答案：程序采用的是 TEA（Tiny Encryption Algorithm）加密算法。

C 语言源代码摘自 http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm：

```
void f (unsigned int* v, unsigned int* k) {
    unsigned int v0=v[0], v1=v[1], sum=0, i;          /* set up */
    unsigned int delta=0x9e3779b9;                    /* a key schedule constant */
    unsigned int k0=k[0], k1=k[1], k2=k[2], k3=k[3];  /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

G.3.4 练习题 2.13

答案：线性反馈位移寄存器。详细资料请参见：https://en.wikipedia.org/wiki/Linear_feedback_shift_register。

程序源代码如下所示：

```
// reworked, from https://en.wikipedia.org/wiki/Linear_feedback_shift_register

#include <stdint.h>

uint16_t f(uint16_t in)
{
   /* taps: 16 14 13 11; feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1 */
   unsigned bit  = ((in >> 0) ^ (in >> 2) ^ (in >> 3) ^ (in >> 5) ) & 1;
   return (in >> 1) | (bit << 15);
};

//#define C 0xACE1u
#define C 0xBADFu

int main(void)
{
   uint16_t lfsr = C;
   unsigned period = 0;

   do
   {
      lfsr=f(lfsr);
      printf ("period=0x%x, lfsr=0x%x\n", period, lfsr);
      ++period;
   } while(lfsr != C);

   return 0;
}
```

G.3.5 练习题 2.14

程序采用的是计算最大公因数 GCD 的算法。源代码可参见：http://beginners.re/exercise-solutions/2/14/GCD.c。

G.3.6 练习题 2.15

这是一个使用蒙特卡罗法计算圆周率的程序，源代码可参见：http://beginners.re/exercise-solutions/

2/15/monte.c。

G.3.7 练习题 2.16

阿克曼（Ackermann）函数（https://en.wikipedia.org/wiki/Ackermann_function）。

```
int ack (int m, int n)
{
        if (m==0)
                return n+1;
        if (n==0)
                return ack (m-1, 1);
                return ack(m-1, ack (m, n-1));
};
```

G.3.8 练习题 2.17

程序采用了第 110 号初等元胞自动机的原理。这种算法的介绍可参见：https://en.wikipedia.org/wiki/Rule_110。

程序源代码可参见：http://beginners.re/exercise-solutions/2/17/CA.c。

G.3.9 练习题 2.18

程序源代码可参见：http://beginners.re/exercise-solutions/2/18/。

G.3.10 练习题 2.19

程序源代码可参见：http://beginners.re/exercise-solutions/2/19/。

G.3.11 练习题 2.20

提示：可参考 Hint: On-Line Encyclopedia of Integer Sequences（OEIS））。

答案：考拉兹猜想。请参考源代码中的注释。

程序源代码可参见：http://beginners.re/exercise-solutions/2/20/collatz.c。

G.4 高难度练习题

G.4.1 练习题 3.2

程序采用的是表查询的简易算法。

源代码请参见：go.yurichev.com/17156。

G.4.2 练习题 3.3

源代码请参见：go.yurichev.com/17157。

G.4.3 练习题 3.4

源代码和解密后的文件请参见：go.yurichev.com/17158。

G.4.4 练习题 3.5

提示：我们可以看到，用户名称的字符串并没有占用整个文件。在偏移量 0x7F 之前的、以零终止的字节被程序忽略了。

源代码请参见：go.yurichev.com/17159。

G.4.5 练习题 3.6

源代码请参见：go.yurichev.com/17160。
结合其他练习，您可以掌握修补这个 web 服务器所有漏洞的方法。

G.4.6 练习题 3.8

源代码请参见：go.yurichev.com/17161。

G.5 其他练习题

G.5.1 “扫雷（Windows XP）”

请参见本书第 76 章。
提示：留意边界字节（0x10）。

参 考 文 献

[al12] Nick Montfort 等人撰写的《10 PRINT CHR$(205.5+RND(1)); : GOTO 10》，The MIT Press, 2012。笔者将之收录为 http://go.yurichev.com/ 17286。

[AMD13a] AMD 在 2013 年发布的《AMD64 Architecture Programmer's Manual》。笔者将之收录为：http://go.yurichev.com/17284。

[AMD13b] AMD 在 2013 年发布的《Software Optimization Guide for AMD Family 16h Processors》。笔者将之收录为：http://go.yurichev.com/17285。

[App10] Apple 在 2010 年发布的《iOS ABI Function Call Guide》。笔者将之收录为 http://go.yurichev.com/17276。

[ARM12] ARM 在 2012 年发布的《ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition》。

[ARM13a] ARM 在 2013 年发布的《ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile》。

[ARM13b] ARM 在 2013 年发布的《ELF for the ARM 64-bit Architecture (AArch64)》。笔者将之收录为 http://go.yurichev.com/17288。

[ARM13c] ARM.在 2013 年发布的《Procedure Call Standard for the ARM 64-bit Architecture (AArch64)》。笔者将之收录为：http://go.yurichev. com/17287。

[ASS96] Harold Abelson、Gerald Jay Sussman 和 Julie Sussman 撰写的《Structure and Interpretation of Computer Programs》，1996。

[Bro] Ralf Brown《The x86 Interrupt List》。笔者将之收录为 http://go.yurichev.com/17292。

[Bur] Mike Burrell《Writing Effcient Itanium 2 Assembly Code》。笔者将之收录为 http://go.yurichev.com/17265。

[Cli] Marshall Cline《C++ FAQ》。笔者将之收录为: http://go.yurichev.com/17291。

[Cor+09] Thomas H. Cormen 等人编写的《Introduction to Algorithms, Third Edition. 3rd》The MIT Press, 2009。ISBN: 0262033844，9780262033848。

[Dij68] Edsger W. Dijkstra《Letters to the editor: go to statement considered harmful"》，发表于《Commun. ACM 11.3（1968 年 3 月）》，pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. 笔者将之收录为 http://go.yurichev.com/ 17299。

[Dol13] Stephen Dolan《mov is Turing-complete》，2013。笔者将之收录为 http://go.yurichev.com/17269。

[Dre07] Ulrich Drepper《What Every Programmer Should Know About Memory》，2007。笔者将之收录为 http://go.yurichev.com/17341。

[Dre13] Ulrich Drepper《ELF Handling For Thread-Local Storage》，2013。笔者将之收录为 http://go.yurichev.com/17272。

[Eic11] Jens Eickhoff《Onboard Computers, Onboard Software and Satellite Operations: An Introduction》，2011。

[Fog13a] AgnerFog《Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms》，2013。 http://go. yurichev.com/17279。

[Fog13b] AgnerFog《The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers》，2013。笔者将之收录为:http://go.yurichev.com/17278。

[Fog14] Agner Fog《Calling conventions》，2014。http://go.yurichev.com/17280

[haq] papasutra of haquebright.《WRITING SHELLCODE FOR IA-64》。笔者将之收录为：http://go.yurichev.com/17340。

[IBM00] IBM《PowerPC (tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors》，2000。笔者将之收录为 http://go.yurichev.com/17281。

[Int13] Intel《Intel® 64 and IA-32 Architectures Software Developer's Manual 》，2013，重点参考了其中的 1，2A，2B，2C，3A，3B，3C 卷。笔者将之收录为: http://go.yurichev.com/17283。

[Int14] Intel《Intel® 64 and IA-32 Architectures Optimization Reference Manual》（2014 年 9 月）。笔者将之收录为 http://go.yurichev.com/17342

[ISO07] ISO《ISO/IEC 9899: TC3（C C99 standard）》（2007）。笔者将之收录为：http://go.yurichev.com/17274

[ISO13] ISO《ISO/IEC 14882:2011 (C++ 11 standard)》(2013)。笔者将之收录为：http://go.yurichev.com/17275

[Jav13] Java《The Java® Virtual Machine Specification Java SE 7 Edition》，2013 年 2 月。笔者将之收录为：http://go.yurichev.com/17345，以及 http://go.yurichev.com/17346

[Ker88] Brian W. Kernighan《The C Programming Language.Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference》, 1988. ISBN: 0131103709.

[Knu74] Donald E. Knuth《Structured Programming with go to Statements》，刊登于《ACM Comput. Surv. 6.4 (Dec. 1974)》. 笔者将之收录为:http://go.yurichev.com/17271, pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635. 356640. URL: http://go.yurichev.com/17300.

[Knu98] Donald E. Knuth《The Art of Computer Programming Volumes 1-3 Boxed Set》 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201485419.

[Loh10] Eugene Loh《The Ideal HPC Programming Language》，发表于《Queue 8.6 (June 2010)》, 30:30–30:38. ISSN: 1542-7730. DOI: 10.1145/1810226.1820518.。笔者将之收录为:http://go.yurichev.com/17298。

[Ltd94] Advanced RISC Machines Ltd《The ARM Cookbook》，1994。笔者将之收录为：http://go.yurichev.com/17273。

[Mit13] Michael Matz/Jan Hubicka/Andreas Jaeger/Mark Mitchell《System V Application Binary Interface. AMD64 Archi- tecture Processor Supplement》，2013。笔者将之收录为：http://go.yurichev.com/17295。

[Mor80] Stephen P. Morse《The 8086 Primer》，1980。笔者将之收录为：http://go.yurichev.com/17351。

[One96] Aleph One.《Smashing The Stack For Fun And Profit》，发表《Phrack（1996）》。笔者将之收录为:http://go.yurichev.com/17266。

[Pie] Matt Pietrek《A Crash Course on the Depths of Win32TM Structured Exception Handling》发表于 MSDN magazine。URL: http://go.yurichev.com/17293.

[Pie02] Matt Pietrek《An In-Depth Look into the Win32 Portable Executable File Format》发表于《MSDN magazine（2002）》. URL:http://go.yurichev.com/17318.

[Pre+07] William H. Press 等人《Numerical Recipes》，2007。

[RA09] Mark E. Russinovich and David A. Solomon with Alex Ionescu《Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition》，2009。

[Ray03] Eric S. Raymond《The Art of UNIX Programming》，Pearson Education, 2003. ISBN: 0131429019。笔者将之收录为:http://go.yurichev.com/17277。

[Rit79] Dennis M. Ritchie《The Evolution of the Unix Time-sharing System》，1979。

[Rit86] Dennis M. Ritchie《Where did ++ come from? (net.lang.c)》，1986。笔者将之收录为：http://go.yurichev.com/17296 [2013 年整理]。

[Rit93] Dennis M. Ritchie《The development of the C language》，发表于《SIGPLAN Not. 28.3（Mar. 1993）》。笔者将之收录为 http://go.yurichev.com/17264, pp. 201–208. ISSN: 0362-1340. DOI: 10.1145/155360.155580. URL: http://go.yurichev.com/17297。

[RT74] D. M. Ritchie and K. Thompson《The UNIX Time Sharing System》，1974。笔者将之收录为 http://go.yurichev.com/17270。

[Sch94] Bruce Schneier《Applied Cryptography: Protocols, Algorithms, and Source Code in C》，1994。

[SK95] SunSoft Steve Zucker and IBM Kari Karhi《SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement》，1995。笔者将之收录为 http://go.yurichev.com/17282。

[Sko12] Igor Skochinsky《Compiler Internals: Exceptions and RTTI》，2012。笔者将之收录为 http://go.yurichev.com/17294。

[Str13] BjarneStroustrup《The C++ Programming Language, 4th Edition》，2013。

[Swe10] Dominic Sweetman《See MIPS Run, Second Edition》，2010。

[War02] Henry S. Warren.《Hacker's Delight》，Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201914654。

[Yur12] Dennis Yurichev《Finding unknown algorithm using only input/output pairs and Z3 SMT solver》，发表于 2012 年。笔者将之收录为 http://go.yurichev.com/17268。

[Yur13] Dennis Yurichev《C/C++ programming language notes》，2013。笔者将之收录为 http://go.yurichev.com/17289。

逆向工程权威指南 下册

逆向工程是一种分析目标系统的过程。

本书专注于软件逆向工程，即研究编译后的可执行程序。本书是写给初学者的一本权威指南。全书共分为12个部分，共102章，涉及软件逆向工程相关的众多技术话题，堪称是逆向工程技术百科全书。全书讲解详细，附带丰富的代码示例，还给出了很多习题来帮助读者巩固所学的知识，附录部分给出了习题的解答。

本书适合对逆向工程技术、操作系统底层技术、程序分析技术感兴趣的读者阅读，也适合专业的程序开发人员参考。

“... 谨向这本出色的教程致以个人的敬意!”

—— Herbert Bos，阿姆斯特丹自由大学教授
《Modern Operating Systems (4th Edition)》作者

“... 引人入胜，值得一读!”

—— Michael Sikorski
《Practical Malware Analysis》的作者



作者简介

Dennis Yurichev，乌克兰程序员，安全技术专家。读者可以通过https://yurichev.com/联系他，并获取和本书相关的更多学习资料。

异步社区 人民邮电出版社 www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／软件开发／安全
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43445-6

ISBN 978-7-115-43445-6
定价：168.00 元（上、下册）