

# 《黑客防线》3 期文章目录

总第 147 期 2013 年

## 漏洞攻防

栈溢出攻击学习与实践 (张少飞) .....	2
漏洞解析之 ExploitMe (woosheep) .....	4

## 编程解析

在 Win64 上实现驱动级解锁文件 (胡文亮) .....	16
在 Win64 上无 HOOK 实现监控驱动加载 (胡文亮) .....	22
MD5 算法设计与编程实现 (张少飞) .....	27
Python 黑客编程: 网站后台暴力破解 (blackcool) .....	30
共享内存的奥秘 (王晓松) .....	37

## Android 远程监控技术

Android 木马揭秘之用户定位技术的实现 (爱无言) .....	42
------------------------------------	----

2013 年第 4 期杂志特约选题征稿 .....	48
---------------------------	----

2013 年稿启示 .....	48
-----------------	----

# 栈溢出攻击学习与实践

文/图 张少飞

## 栈结构及形成过程

一个进程可能被加载到内存中不同的区域执行。进程运行所使用的内存空间按照功能，大致都能分成以下 4 个部分：

数据区：用来存储全局变量等。

栈区：用来存储函数之间的调用关系，以保证被调用函数在返回时恢复到母函数中继续执行。

堆区：动态分配与回收是堆区的最大特点，进程能够动态的申请一定大小的缓冲，并在用完之后归还给堆区。

代码区：存储 CPU 所执行的机器码，CPU 会到这个区域来读取指令并执行。

其中栈区由系统自动维护，它实现了高级语言中的函数调用。对于 C 语言等高级语言，栈区的 PUSH、POP 等平衡堆栈细节是透明的。请看如下代码：

```
int function_b(int argument_B1, int argument_B2)
{
    int variable_b1, variable_b2;
    variable_b1=argument_B1+argument_B2;
    variable_b2=argument_B1-argument_B2;
    return variable_b1*variable_b2;
}

int function_a(int argument_A1, int argument_A2)
{
    int variable_a;
    variable_a = function_b(argument_A1,argument_A2) + argument_A1 ;
    return variable_a;
}

int main(int argumentc, char **argumentv, char **envp)
{
    int variable_main;
    variable_main=function_a(4,3);
    return variable_main;
}
```

同一文件不同函数的代码，在内存代码区中的分布可能先后有序也可能无序，相邻也可能相离甚远。

当 CPU 执行调用 function\_a 函数时，会从代码区中 main 方法对应的二进制代码的区域跳转到 function\_a 函数对应的二进制代码区域，在那里获取指令并执行；当 function\_a 函数执行完闭，需要返回时，又会跳回到 main 方法对应的指令区域，紧接着调用 function\_a 后

面的指令继续执行 main 方法的代码。

这些代码区中精确的跳转都是通过与栈区巧妙的配合完成的。当函数调用发生时，栈区会为这个函数开辟一个新的栈区单元，并将它压入栈中。这个栈区单元中的内存空间被它所属的函数独占，正常情况下是不会和别的函数共享的。当函数返回时，栈区会弹出该函数所对应的栈区单元。

在函数调用的过程中，伴随的栈区中的操作如下：

在 main 方法调用 function\_a 时，先在自己的栈区单元中压入函数返回地址，而后为 function\_a 创建新栈区单元压入栈区。

在 function\_a 调用 function\_b 时，同样先在自己的栈区单元中压入函数返回地址，然后为 function\_b 创建新栈区单元并压入栈区。

在 function\_b 返回时，function\_b 的栈区单元被弹出栈区，function\_a 栈区单元中的返回地址“露”出栈顶，此时处理器按照这个返回地址重新跳到 function\_a 代码区中执行。

在 function\_a 返回时，function\_a 的栈区单元被弹出栈区，main 方法栈区单元中的返回地址“露”出栈顶，此时处理器按照这个返回地址跳到 main 方法代码区中执行。

每一个函数独占自己的栈区单元空间，当前正在运行的函数的栈区单元总是在栈顶。Win32 系统提供两个特殊的寄存器用来标识位于栈区栈顶的栈区单元。

ESP：栈指针寄存器，其内存放着指向栈区最上面一个栈区单元的栈顶的指针。

EBP：基址指针寄存器，其内存放着指向栈区最上面一个栈区单元的底部的指针。

函数栈区单元：ESP 和 EBP 之间的内存空间为当前栈区单元，EBP 标识了当前栈区单元的底部，ESP 标识了当前栈区单元的顶部。

在函数栈区单元中一般包含以下几类重要信息：

局部变量：为函数局部变量开辟内存空间。

栈区单元状态值：保存前栈区单元的顶部和底部（实际上只保存前栈区单元的底部，前栈区单元的顶部能够通过平衡堆栈计算得到），用来在本帧被弹出后，恢复上一个栈区单元。

函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

函数调用发生时用到的指令大致如下：

```

;调用前
push 参数 C ;
push 参数 B
push 参数 A
call 函数地址 ;call 指令完成两项工作：向栈中压入返回地址；跳转
    
```

```

;函数开始处代码形式
push ebp ;保存旧栈区单元的底部
mov ebp, esp ;栈区单元切换
sub esp, xxx ;抬高栈顶，开辟新栈区单元空间
    
```

函数调用大约包括以下几个步骤：

- 1) 参数入栈：将参数从右向左依次压入栈区中。
- 2) 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- 3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

4) 栈区单元调整: 具体包括保存当前栈区单元状态值, EBP 入栈; 将当前栈区单元切换到新栈区单元, 将 ESP 值装入 EBP, 更新栈区单元底部; 给新栈区单元分配空间, 将 ESP 减去所需空间的大小, 抬高栈顶。

类似的, 函数返回时的汇编指令序列大致如下:

```
add xxx, esp ;回收当前的栈区单元
pop ebp ;恢复上一个栈区单元底部位置
ret n ;有两个功能: 即弹出栈区单元中的返回地址, 让处理器恢复调用前的代码区
```

函数返回的步骤如下:

1) 通常将返回值保存在 EAX 中。

2) 弹出当前栈区单元, 恢复上一个栈区单元。具体包括 平衡堆栈的基础上, 给 ESP 加上栈区单元的大小, 回收当前栈区单元的空间; 将保存的前栈区单元 EBP 值弹入 EBP 寄存器, 恢复出上一个栈区单元; 将函数返回地址弹给 EIP 寄存器; 跳转: 按照函数返回地址继续执行母函数。

栈区结构就是按照这样的函数调用约定组织起来的。

### 栈溢出攻击实践

本实践是我自己手写了一个简单的 C 语言程序 (VC6.0 编译), 然后通过溢出栈区, 覆盖函数的返回地址, 从而改变程序的执行流程, 以达到攻击效果。

程序代码如下:

```
#include <stdio.h>
#define PWD "1234567"
int verify_pwd (char *pwd)
{
    int right;
    char buf[8];
    right=strcmp(pwd,PWD);
    strcpy(buf,pwd);//over flowed here!
    return right;
}
main()
{
    int flag_valid=0;
    char pwd[1024];
    FILE * fp;
    if(!(fp=fopen("pwd.txt","rw+")))
    {
        exit(0);
    }
    fscanf(fp,"%s",pwd);
    flag_valid = verify_pwd(pwd);
    if(flag_valid)
```

```

{
    printf("incorrect pwd!\n");
}
Else
{
    printf("Good Job! Verification passed!\n");
}
fclose(fp);
}
    
```

首先用 OD 加载得到的可执行 PE 文件，如图 1 所示。

Address	Hex dump	Disassembly	Comment
004010D0	- 83C4 04	ADD ESP,4	
004010E0	> 8D85 FCFBFFFF	LEA EAX,DWORD PTR SS:[EBP-404]	
004010E6	- 50	PUSH EAX	
004010E7	- 68 84304200	PUSH OFFSET 4_3_over.??_CG_02DILL0?CFS	Arg3 Format = "%s"
004010EC	- 8B8D F8FBFFFF	MOV ECX,DWORD PTR SS:[EBP-400]	
004010F2	- 51	PUSH ECX	stream fscanf
004010F3	- E8 B8030000	CALL 4_3_over.fscanf	
004010F8	- 83C4 0C	ADD ESP,0C	
004010FB	- 8D95 FCFBFFFF	LEA EDX,DWORD PTR SS:[EBP-404]	
00401101	- 52	PUSH EDX	
00401102	- E8 FEFEFFFF	CALL 4_3_over.00401005	
00401107	- 83C4 04	ADD ESP,4	
0040110A	- 8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
0040110D	- 837D FC 00	CMP DWORD PTR SS:[EBP-4],0	
00401111	- 74 0F	JE SHORT 4_3_over.00401122	
00401113	- 68 68304200	PUSH OFFSET 4_3_over.??_CG_0DF011FMainC	format = "incorrect"
00401118	- E8 13030000	CALL 4_3_over.printf	printf
0040111D	- 83C4 04	ADD ESP,4	
00401120	- EB 0D	JMP SHORT 4_3_over.0040112F	
00401122	> 68 28304200	PUSH OFFSET 4_3_over.??_CG_0DD0F1PBB0Com	Format = "Congrat"
00401127	- E8 04030000	CALL 4_3_over.printf	printf
0040112C	- 83C4 04	ADD ESP,4	
0040112F	> 8B85 F8FBFFFF	MOV EAX,DWORD PTR SS:[EBP-400]	
00401135	- 50	PUSH EAX	stream fclose
00401136	- E8 15020000	CALL 4_3_over.fclose	
0040113B	- 83C4 04	ADD ESP,4	
0040113E	- 5F	POP EDI	
0040113F	- 5E	POP ESI	
00401140	- 5B	POP EBX	

图 1

阅读反汇编代码，能够知道通过验证的程序分支的指令地址为 0x00401122。

0x00401102 处的函数调用就是 verify\_pwd 函数，之后在 0x0040110A 处将 EAX 中的函数返回值取出，在 0x0040110D 处与 0 比较，然后决定跳转到提示验证错误的分支或提示通过验证的分支。提示通过验证的分支，从 0x00401122 处的参数压栈开始。

通过用 OD 调试，发现栈区单元中的变量分布情况基本没变，这样就能够按照如下方法构造 pwd.txt 中的数据了。

为了字节对齐并且方便辨认，将“4321”作为一个串块。buf[8]共需要 2 个这样的单元，第 3 个串块将 right 覆盖，第 4 个串块将前栈区单元 EBP 值覆盖，第 5 个串块将函数返回地址覆盖。

为了将第 5 个串块的 ASCII 码值（0x34333231）改为通过验证分支指令的地址（0x00401122），借助十六进制编辑工具来完成（我用的 UltraEdit），因为部分 ASCII 码所对应符号无法用键盘输入。

Step 1: 新建一个名称为 pwd.txt 的文件，并使用记事本程序打开，输入 5 个“4321”，

如图 2 所示。

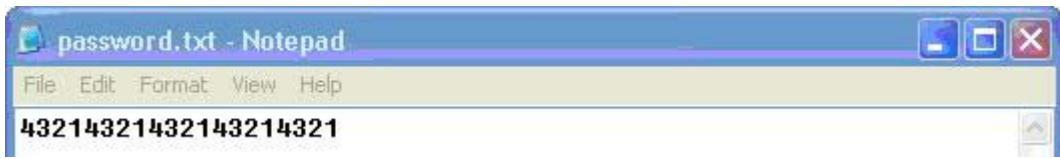


图 2

Step 2: 保存，关闭记事本并用 UltraEdit 打开，如图 3 所示。

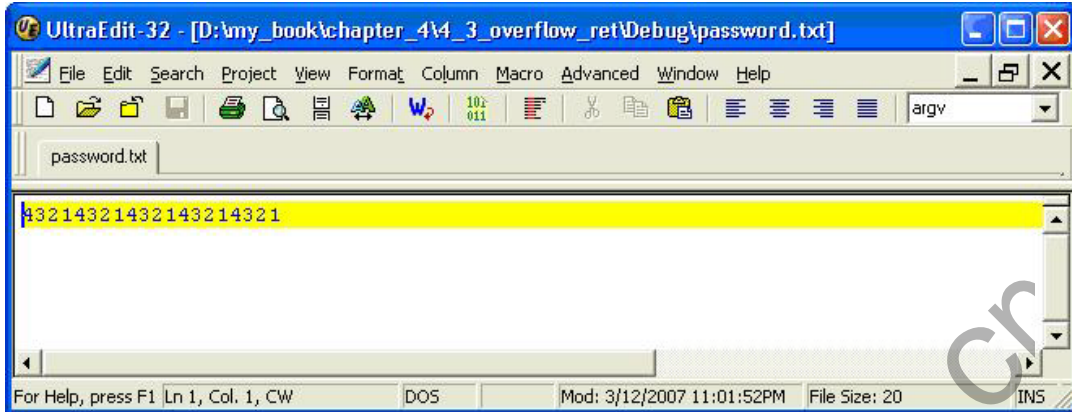


图 3

Step 3: 将 UltraEdit 的编辑模式切换到十六进制，如图 4 所示。

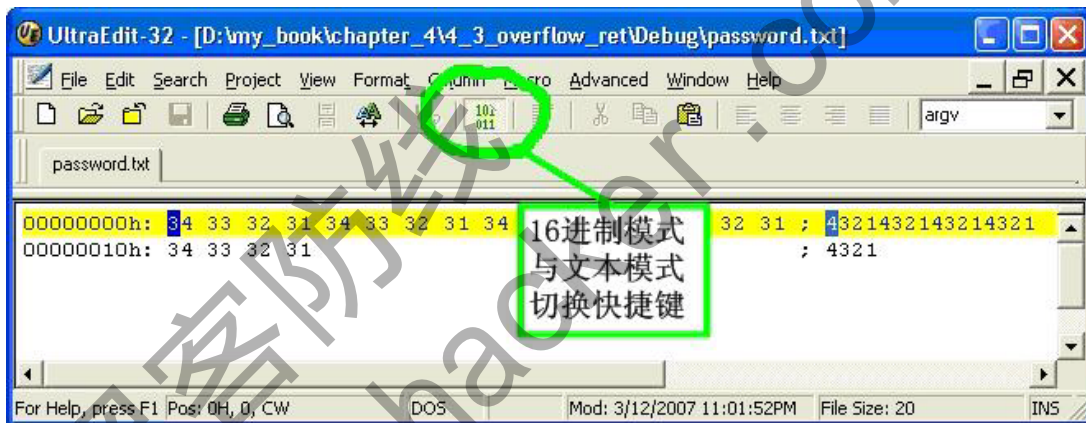


图 4

Step 4: 将最后 4 个字节改为新的函数返回地址，如图 5 所示。

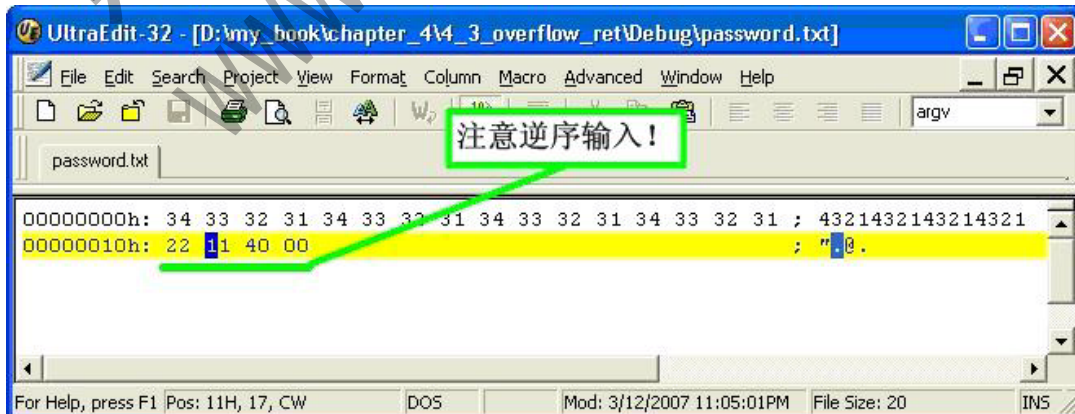


图 5

Step 5: 此时再切换回文本编辑模式，最后的 4 个字节的对应字符显示结果为乱码，如

图 6 所示。

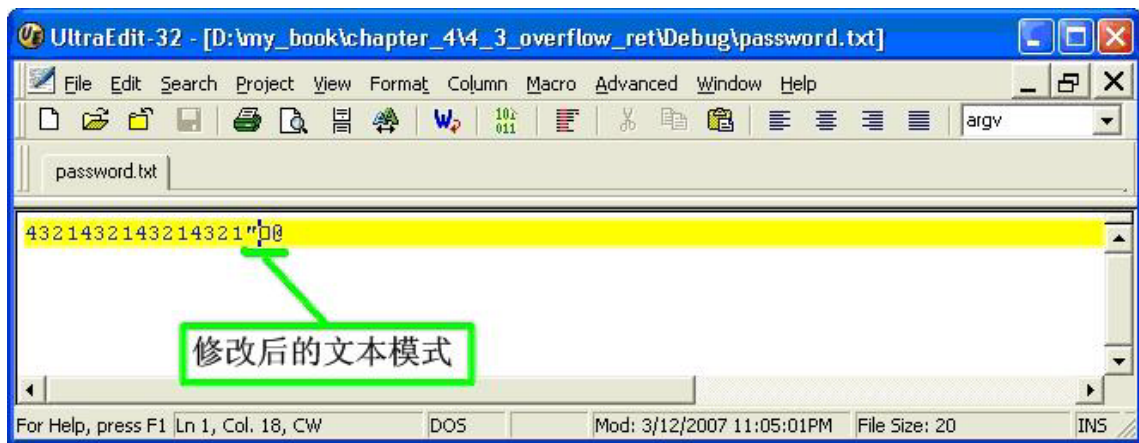


图 6

将 pwd.txt 保存后，用 OD 加载程序并调试，程序运行结果如图 7 所示。

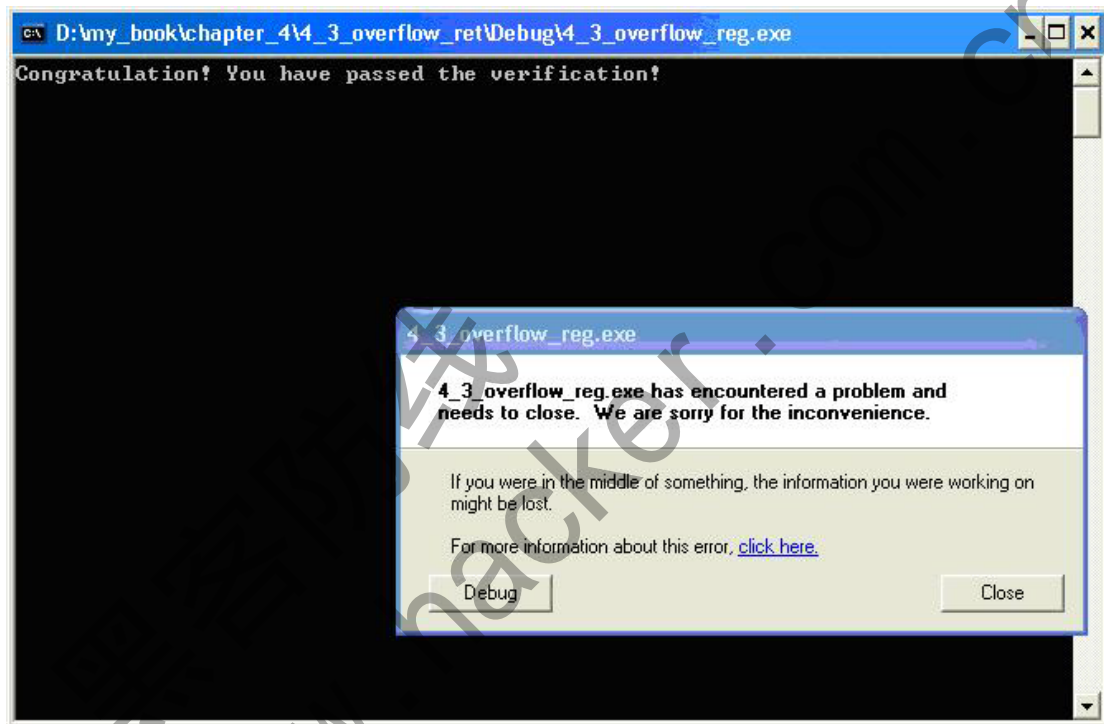


图 7

### 学习心得

能看懂二进制是研究安全技术所必需的技能。信息安全技术不仅需要计算机理论基础很扎实，更需要优秀的动手、实践能力，是一个对技术性要求很高的领域。

缓冲区溢出攻击的理论我很早就已经学习了，以为只是修改返回地址将 CPU 指到缓冲区中的恶意代码而已，但当自己动手实践时，才发现实际情形原来比原理要复杂很多。信息安全需要有强烈的兴趣做动力，还需要有能够为了梦想持之以恒的坚定意志。

# 漏洞解析之 ExploitMe

文/图 woosheep

说到漏洞，大家第一反应可能是 CVE 或者 KB 后带着一大串高深莫测的数字，看了就不敢碰了。本文为大家带来的是一个为考试而生的 ExploitMe 的解析，难度不难，很适合用来开题，消除 Exploit 的神秘感。

有人气吞山河地断定，凡是软件都有漏洞，这是一个哲学层面的问题，这里我们只关心什么样的漏洞可以溢出。漏洞可以粗分为 local 和 remote 两种类型，后者在入侵中更为常见，但要求也更苛刻，必须是允许输入才可能被溢出的。有朋友说，直接 patch 或者工具修改不行吗？行，不过仅限于 local，为什么呢？我们当然可以随意修改本机上的软件，但要利用漏洞入侵服务器，最开始只能通过 I/O 进行交互，无法从进程层面或者二进制码层面进行访问。不支持 I/O 的软件，就算存在漏洞，我们也无法直接利用。

选择正确的切入点很重要。一个可能被利用的 remote 型软件，要么开有端口，要么支持文件读取，本文的 ExploitMe 就属于后者，它要读取同目录下名为 exploit.dat 的文件。

怎样知道它读取什么文件呢？一是看软件说明，软件毕竟是供人使用的，或多或少都会有介绍，看看说明中是否提及软件会读取什么文件。在入侵某个系统时，熟知这个系统的输入环境往往会让工作如鱼得水。要是某位奇葩管理员装的全是剑走偏锋的玩意怎么办呢？那就只好花点时间找到相同的软件，照样配置好后逆向研究。以本 ExploitMe 为例，OD 载入，下断 CreateFileA，看一眼堆栈，入图 1 所示。

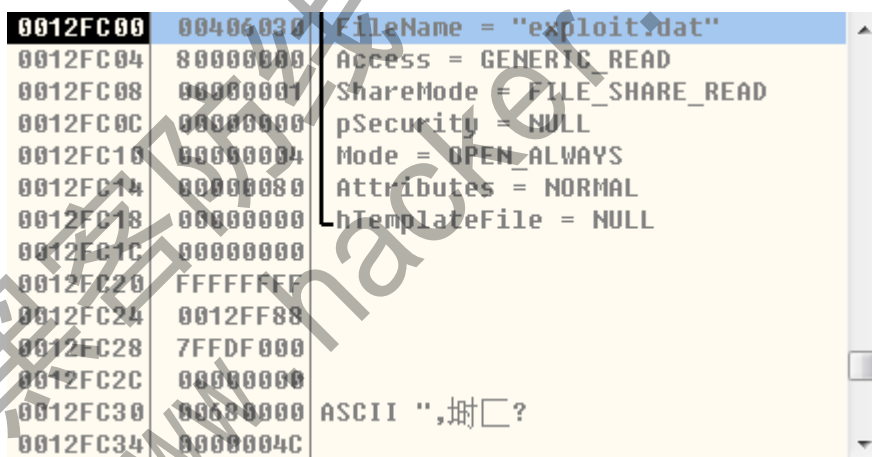


图 1

这时堆栈里保存着完整的输入参数，看到 FileName，嗯，懂了，建立一个名字为 exploit.dat 的文件。有了输入，接着是找溢出点。溢出点通常要同时满足两个条件，一是程序需要使用栈里的某一段数据作为执行地址，二是输入的数据能够覆盖到那一段栈地址。

首先找找有没有将栈数据作为地址执行的，找到了这里，如图 2 所示。



```

0040112D . 8BFD      mov     edi, ebp
0040112F . 81FB 84000000 cmp    ebx, 84
00401135 . F3:AB     rep    stos dword ptr es:[edi]
00401137 . 77 19     ja     short 00401152
00401139 . 8BCB     mov    ecx, ebx
0040113B . 8BF5     mov    esi, ebp
0040113D . 8BD1     mov    edx, ecx
0040113F . 8DBC24 A80000 lea    edi, dword ptr [esp+A8]
00401146 . C1E9 02   shr    ecx, 2
00401149 . F3:A5     rep    movs dword ptr es:[edi], dword ptr [esi]
0040114B . 8BCA     mov    ecx, edx
0040114D . 83E1 03   and    ecx, 3
00401150 . F3:A4     rep    movs byte ptr es:[edi], byte ptr [esi]
00401152 > 8B4424 20  mov    eax, dword ptr [esp+20]
00401156 . 8D4C24 20  lea    ecx, dword ptr [esp+20]
0040115A . FF10     call   dword ptr [eax]
0040115C . 8B9424 A40000 mov    edx, dword ptr [esp+A4]
00401163 . 8D8C24 A40000 lea    ecx, dword ptr [esp+A4]
0040116A . FF12     call   dword ptr [edx]
0040116C . 8B7C24 14  mov    edi, dword ptr [esp+14]
    
```

图 2

这里有两个 call，分别读取 eax 和 edx 作为地址，而且这两个寄存器的值都与栈有关！一个是 esp+0x20，一个是 esp+0xA4，先记好这两个值。

现在要验证这两处栈地址是否能被输入覆盖，我们看一下程序将文件读到哪里去了。重新载入，下断 ReadFile，来到图 3 所示的位置。

```

00401077 . 53       push   ebx
00401078 . 56       push   esi
00401079 . FF15 18504000 call   dword ptr [<KERNEL32.GetFileSize>]
0040107F . 8BD8     mov    ebx, eax
00401081 . 81FB 00020000 cmp    ebx, 200
00401087 . 0F87 EF000000 ja     0040117C
0040108D . 8D4424 1C lea    eax, dword ptr [esp+1C]
00401091 . 6A 00    push   0
00401093 . 50       push   eax
00401094 . 8D8C24 300100 lea    ecx, dword ptr [esp+130]
00401098 . 53       push   ebx
0040109C . 51       push   ecx
0040109D . 56       push   esi
0040109E . FF15 14504000 call   dword ptr [<KERNEL32.ReadFile>]
004010A4 . 8BCB     mov    ecx, ebx
004010A6 . 8D8424 280100 lea    esi, dword ptr [esp+128]
004010AD . 8BD1     mov    edx, ecx
004010AF . 8BFD     mov    edi, ebp
004010B1 . C1E9 02   shr    ecx, 2
004010B4 . F3:A5     rep    movs dword ptr es:[edi], dword ptr [esi]
    
```

图 3

看来不太妙。读入 Buffer 的地址是 esp+0x130，后面通过一个 rep 指令复制到 ebp，离我们想要的地址都挺远。能不能简单粗暴的增大文件直到覆盖特定栈地址呢？如果程序对读入文件大小没有限制是可以的，但这里我们发现了第二个不太妙的地方，如图 4 所示。

```

004010F5 . 81FB 84000000 cmp    ebx, 84
004010FB . A3 14854000 mov    dword ptr [408514], eax
00401100 . 77 16     ja     short 00401118
    
```

图 4

程序会比较文件大小，若超过 0x84 就 GameOver。现在有两个选择，一是继续寻找有没有溢出点，一是用全 0 填一个 0x84 大小的 exploit.dat 文件。第二种有点碰运气，不过取巧才是黑客的浪漫，所以这里选第二种。

运行，程序崩溃了！对于程序员，程序崩溃是一件遗憾的事，但对于黑客，崩溃如同听见胜利女神的召唤。找到崩溃日志，如图 5 所示。

0040116A 访问违规: 读取 [00000000]  
调试的程序无法处理异常

图 5

内存地址 0x0040116A 处发生读取异常，数值就是我们填入的 0（如果不确定，可以多次变化文件的填写内容作为验证）！0x0040116A 正好是上文提及的两个 Call 之一。

回溯一下这个 Call 的值是怎么来的。首先是 esp+0xA4，如图 6 所示。

0040115C	. 8B9424 A40000	mov	edx, dword ptr [esp+A4]
00401163	. 8D8C24 A40000	lea	ecx, dword ptr [esp+A4]
0040116A	. FF12	call	dword ptr [edx]

图 6

那么，esp+0xA4 是怎么来的呢？上下文中没有 esp+0xA4 的专门赋值，但是算一算，文件最大可达 0x84，在这个范围以内的栈地址都可覆盖，也就是只要在 esp+0x24 到 esp+0xA4 有可用的赋值即可。往上找到一处，如图 7 所示。

00401104	. 8BF5	mov	esi, ebp
00401106	. 8BC1	mov	eax, ecx
00401108	. 8D7C24 24	lea	edi, dword ptr [esp+24]
0040110C	. C1E9 02	shr	ecx, 2
0040110F	. F3:A5	rep	movs dword ptr es:[edi], dword ptr [esi]

图 7

这一段栈地址从 esp+0x24 开始，赋值的是 ebp 指向的值，而且长度覆盖了 esp+0xA4。上文说过，buffer 的数据通过一个 rep 指令复制到 ebp 里去，这样就对了。文件内容通过 ReadFile 读入 buffer，再被复制到 ebp 指向的内存地址，最后通过 rep 赋值给 esp+0x24 开始，共计 0x84 大小的栈地址。也就是说，文件的最后一个 DWORD，恰好可以控制 0x0040116A 处 Call 的调用地址，只要将栈首地址 esp+0x24 填入，就可以获得 0x80 大小的操作空间。

可以控制地址是好事，可是由于软件缺乏 jmp esp 等跳板指令，不得不把栈首地址硬编码，降低了 exploit 的适用性，如果操作系统加入 ASLR（内存地址随机布局）机制，理论上能很好地对抗硬编码的 exploit。

知道了怎么利用这个漏洞，那该怎么防呢？

这次漏洞之所以能成功溢出，归功于多次的内存复制，而这些多次内存复制没有功能上的需要，很可能是出题人为了方便解题故意“放水”，但在一个繁大复杂的系统中，出现这种低级错误并不奇怪。在调试过程中，我们发现这个程序至少做了两个防溢出的措施，最明显的就是文件不得大于 0x84，可惜少算了一个 DWORD，甚至可能只是少写了一个等号，只要修改为不得大于等于 0x84 即可堵上这个漏洞。

另一个措施做得比较隐蔽，如图 8 所示。程序会将使用过的 buffer 清 0，这对于保存栈数据绝对是坏消息，如果栈数据在溢出前被清 0，那么就算覆盖了返回地址，也只会导致程序崩溃，而无法为我所用。幸亏这个程序的清 0 措施有两大问题，一是最后一段，即直接造成溢出的那一段没有清 0；另一个是清 0 的范围有误，只设置了 0x80 大小，比合法的文件输入上限 0x84，刚好小一个 DWORD，黑客还是可以从远端控制 CALL 地址，虽然利用起来更为困难，但仍无法彻底消除溢出隐患。

```

004010B8 . 33C0      xor     eax, eax
004010BA . 83E1 03   and     ecx, 3
004010BD . 68 54604000 push   00406054
004010C2 . F3:A4    rep     movs byte ptr es:[edi], byte ptr [esi]
004010C4 . B9 80000000 mov     ecx, 80
004010C9 . 8DBC24 2C010 lea    edi, dword ptr [esp+12C]
004010D0 . F3:AB    rep     stos dword ptr es:[edi]
    
```

图 8

这样，一个 ExploitMe 即被我们成功解析了。学习溢出，对于漏洞既要知其然，也要之前所以然，才能不断地做到知识积累。

(完)

黑客防线  
www.hacker.com.cn



# 在 Win64 上实现驱动级解锁文件

文/图 胡文亮

相信大家在 Windows 系统上都遇到过想删除一个文件时却被提示“无法删除”的情况。我在初学电脑时遇到这种情况只能自认倒霉，重启之后再删除文件。学习了 Windows 后知道了在正常情况下（即不算文件被文件过滤驱动或者各种 API HOOK 保护的情况），遇到这个提示只有两种可能性：你没有删除这个文件的权限；有句柄在某个进程里被打开了。

解决第一种情况不需要编程，只要把以下代码保存成\*.reg 文件并添加到注册表即可。

Windows Registry Editor Version 5.00

```
[HKEY_CLASSES_ROOT\*\shell\takeownership]
```

```
@="Take ownership"
```

```
"HasLUAShield"=""
```

```
"NoWorkingDirectory"=""
```

```
[HKEY_CLASSES_ROOT\*\shell\takeownership\command]
```

```
@="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant administrators:F"
```

```
"IsolatedCommand"="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant  
administrators:F"
```

```
[HKEY_CLASSES_ROOT\exefile\shell\takeownership]
```

```
@="Take ownership"
```

```
"HasLUAShield"=""
```

```
"NoWorkingDirectory"=""
```

```
[HKEY_CLASSES_ROOT\exefile\shell\takeownership\command]
```

```
@="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant administrators:F"
```

```
"IsolatedCommand"="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant  
administrators:F"
```

```
[HKEY_CLASSES_ROOT\dllfile\shell\takeownership]
```

```
@="Take ownership"
```

```
"HasLUAShield"=""
```

```
"NoWorkingDirectory"=""
```

```
[HKEY_CLASSES_ROOT\dllfile\shell\takeownership\command]
```



```
@="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant administrators:F"
"IsolatedCommand"="cmd.exe /c takeown /f \"%1\" && icacls \"%1\" /grant
administrators:F"
```

```
[HKEY_CLASSES_ROOT\Directory\shell\takeownership]
```

```
@="Take ownership"
```

```
"HasLUAShield"=""
```

```
"NoWorkingDirectory"=""
```

```
[HKEY_CLASSES_ROOT\Directory\shell\takeownership\command]
```

```
@="cmd.exe /c takeown /f \"%1\" /r /d y && icacls \"%1\" /grant administrators:F
/t"
```

```
"IsolatedCommand"="cmd.exe /c takeown /f \"%1\" /r /d y && icacls \"%1\" /grant
administrators:F /t"
```

当要删除一个文件而遇到“无法删除需要权限”的提示时，只要对着文件按下右键，选择“Take ownership”再删除文件即可。

第二种情况就要通过编程解决了，这也就是本文的核心内容。要删除被打开的文件，比较好的方法是关闭此文件在其它进程里的句柄（直接解析文件系统也可以，不过这个难度太大，而且不通用）。总体来说，步骤分为以下两步：枚举系统句柄表；获得所有和此文件有关的句柄并关闭。具体到代码级的思想，又可以分为以下几步：

- 1) 调用 ZwQuerySystemInformation 的 16 功能号来枚举系统里的句柄；
  - 2) 打开拥有此句柄的进程，并把此句柄复制到自己的进程；
  - 3) 用 ZwQueryObject 查询句柄的类型和名称；
  - 4) 如果发现此句柄的类型是文件句柄，名称和被锁定的文件一致，就关闭此句柄。
- 重复 2、3、4 步，直到遍历完系统里所有的句柄。

实现代码如下：

```
VOID CloseFileHandle(char *szFileName)
{
    PVOID Buffer;
    ULONG BufferSize = 0x20000, rtl=0;
    NTSTATUS Status, qost=0;
    NTSTATUS ns = STATUS_SUCCESS;
    ULONG64 i=0;
    ULONG64 qwHandleCount;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO *p;
    OBJECT_BASIC_INFORMATION BasicInfo;
    POBJECT_NAME_INFORMATION pNameInfo;
```

```
ULONG ulProcessID;
HANDLE hProcess;
HANDLE hHandle;
HANDLE hDupObj;
CLIENT_ID cid;
OBJECT_ATTRIBUTES oa;
CHAR szFile[260]={0};
Buffer=kmalloc(BufferSize);
memset(Buffer,0,BufferSize);
Status = ZwQuerySystemInformation(16, Buffer, BufferSize, 0);
//SystemHandleInformation
while(Status == 0xC0000004) //STATUS_INFO_LENGTH_MISMATCH
{
    kfree(Buffer);
    BufferSize = BufferSize * 2;
    Buffer=kmalloc(BufferSize);
    memset(Buffer,0,BufferSize);
    Status = ZwQuerySystemInformation(16, Buffer, BufferSize, 0);
}
if (!NT_SUCCESS(Status)) return;
qwHandleCount=((SYSTEM_HANDLE_INFORMATION *)Buffer)->NumberOfHandles;
p=(SYSTEM_HANDLE_TABLE_ENTRY_INFO *)((SYSTEM_HANDLE_INFORMATION
*)Buffer)->Handles;
//clear array
memset(HandleInfo,0,1024*sizeof(HANDLE_INFO));
//ENUM HANDLE_PROC
for(i=0;i<qwHandleCount;i++)
{
    ulProcessID = (ULONG)p[i].UniqueProcessId;
    cid.UniqueProcess = (HANDLE)ulProcessID;
    cid.UniqueThread = (HANDLE)0;
    hHandle = (HANDLE)p[i].HandleValue;
    InitializeObjectAttributes( &oa ,NULL ,0 ,NULL ,NULL );
    ns = ZwOpenProcess( &hProcess ,PROCESS_DUP_HANDLE ,&oa ,&cid );
    if ( !NT_SUCCESS( ns ) )
    {
        KdPrint(( "ZwOpenProcess : Fail " ));
        continue;
    }
}
```



```

    ns = ZwDuplicateObject( hProcess , hHandle , NtCurrentProcess() , &hDupObj ,
PROCESS_ALL_ACCESS , 0 , DUPLICATE_SAME_ACCESS );
    if ( !NT_SUCCESS( ns ) )
    {
        KdPrint(( "ZwDuplicateObject : Fail " ));
        continue;
    }
    //get basic information
    ZwQueryObject( hDupObj , ObjectBasicInformation , &BasicInfo ,
sizeof( OBJECT_BASIC_INFORMATION ) , NULL );
    //get name information
    pNameInfo = ExAllocatePoolWithTag( PagedPool , 1024 , 'ONON' );
    RtlZeroMemory( pNameInfo , 1024 );
    qost=ZwQueryObject( hDupObj, ObjectNameInformation, pNameInfo, 1024,
&rtl );
    //get information and close handle
    UnicodeStringToCharArray (&(pNameInfo->Name), szFile);
    ExFreePool( pNameInfo );
    ZwClose( hDupObj );
    ZwClose( hProcess );
    if( !_stricmp( szFile, szFileName ))
    {
        PEPROCESS ep=LookupProcess( (HANDLE) (p[i].UniqueProcessId));
        ForceCloseHandle( ep, p[i].HandleValue );
        ObDereferenceObject( ep );
    }
}
}

```

接下来说说如何关闭其它进程里的句柄。

- 1) 用 KeStackAttachProcess “依附” 到目标进程;
- 2) 用 ObSetHandleAttributes 设置句柄为 “可以关闭”;
- 3) 用 ZwClose 关闭句柄;
- 4) 用 KeUnstackDetachProcess 脱离 “依附” 的目标进程。

实现代码如下:

```

VOID ForceCloseHandle(PEPROCESS Process, ULONG64 HandleValue)
{
    HANDLE h;

```

```

KAPC_STATE ks;
OBJECT_HANDLE_FLAG_INFORMATION ohfi;
if( Process==NULL )
    return;
if( !MmIsValid(Process) )
    return;
KeStackAttachProcess(Process, &ks);
h=(HANDLE)HandleValue;
ohfi.Inherit=0;
ohfi.ProtectFromClose=0;
ObSetHandleAttributes(h, &ohfi, KernelMode);
ZwClose(h);
KeUnstackDetachProcess(&ks);
}
    
```

要注意的是，要解锁的文件的路径不能写成常见的 DOS 格式，而要写成 NT 格式，比如“c:\lockfile.txt”的 NT 格式路径可能是“\\Device\\HarddiskVolume2\\LockFile.txt”。为什么说“可能是”呢？因为前半段“\\Device\\HarddiskVolumeX”中的 X 并不能确定是什么，要通过转换才知道。转换方法很简单，用 QueryDosDevice 就行了。以下是封装好的函数：

```

char *DosPathToNtPath(char *szFileName)
{
    char szDosDrive[3]={0};
    char *szNtDrive=NULL,*szFilePart=NULL,*szNtPath=NULL;
    szNtDrive=(char*)malloc(260);
    memset(szNtDrive,0,260);
    memcpy(szDosDrive,szFileName,2);
    QueryDosDeviceA(szDosDrive,szNtDrive,260);
    szFilePart=Mid(szFileName,3,0);
    szNtPath=cs(szNtDrive,szFilePart);
    free(szFilePart);
    free(szNtDrive);
    return szNtPath;
}
    
```

可能的输出结果如图 1 所示。





图 1

接下来说说测试步骤：

- 1) 新建文件 c:\lockfile.txt。
- 2) 用 lockfile.exe 锁定文件 c:\lockfile.txt，如图 2 所示。

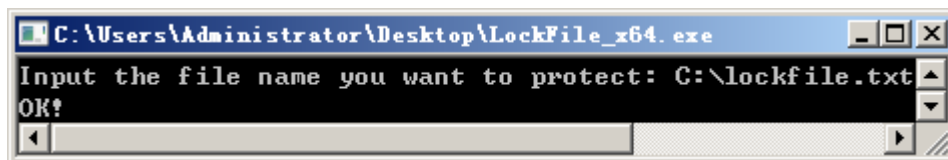


图 2

3) 双击 c:\lockfile.txt，会提示无法打开（如果此时用 Unlocker 查看，会发现系统里每个进程都有 c:\lockfile.txt 的句柄，如图 3 所示）。

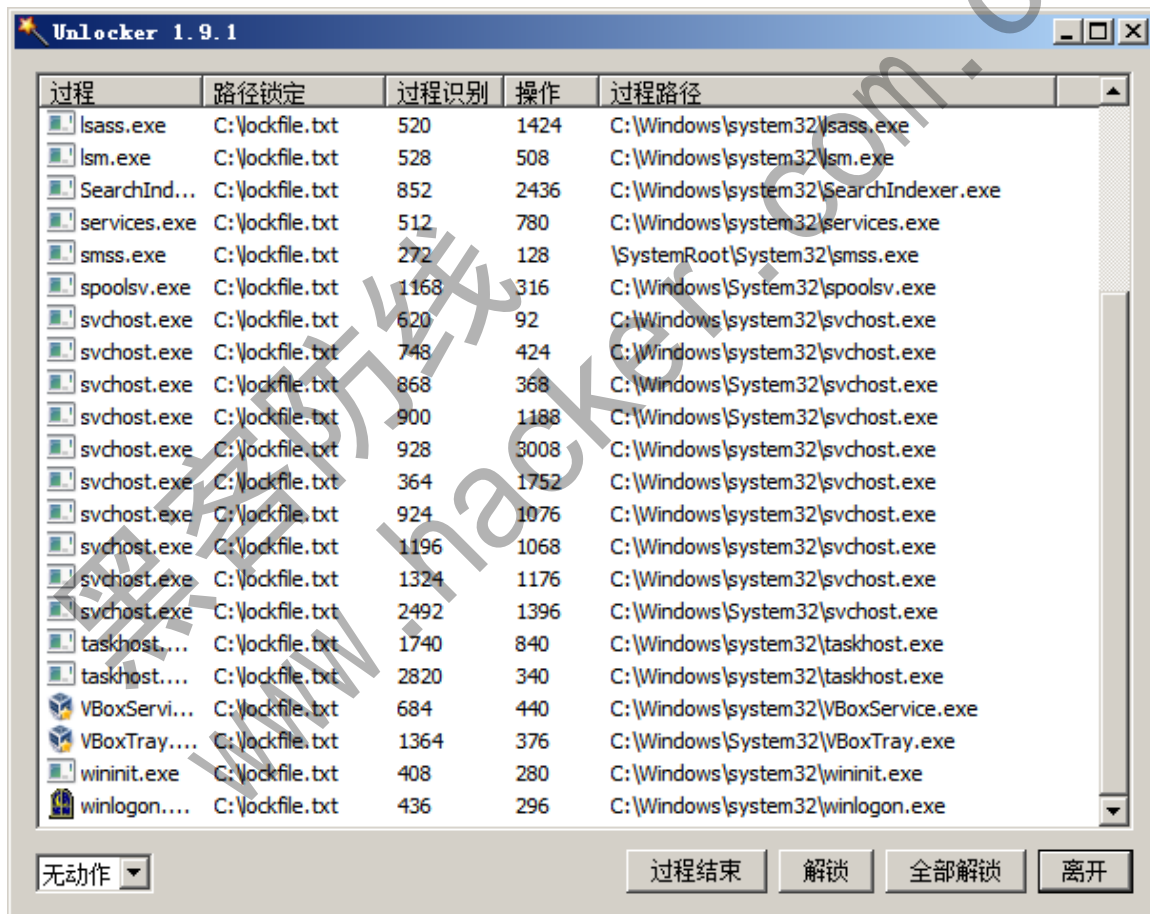


图 3

4) 加载 UnlockFile.sys，再次打开 c:\lockfile.txt，发现文件又可以打开了。

本文到此结束，代码在 Win7 X64 和 Win8 X64 上测试通过（运行任何程序时，都要以管理员权限运行）。在 Win32 上可以用相同的方法，但是结构体的定义并不相同，对应的结构体需要自己去寻找。



# 在 Win64 上无 HOOK 实现监控驱动加载

文/图 胡文亮

在 32 位系统上监控驱动加载，常用的手段是 Hook NtLoadDriver，不过有不少方法加载驱动无需经过 NtLoadDriver，比如用 ZwSetSystemInformation，或者使用一些未公开的 ODay 方法，可见 Hook NtLoadDriver 是极其表层并不可信的方法。后来黑防上刊登过一篇 Hook MmCheckSystemImage 来拦截驱动加载的方法，可惜此方法在 Vista 以后的操作系统上并不适用。其实大家都多虑了，微软早就帮我们想好了一种标准方法来监控驱动的加载，下面我就详细地介绍一下。

此函数名称为 PsSetLoadImageNotifyRoutine，可以设置一个“映像加载通告例程”，来通知你的驱动当前系统在加载什么 DLL 或者驱动。有人可能认为这个标准方法的检控非常表层，其实恰恰相反，这个方法非常底层，大部分加载驱动的方法都可以绕过 NtLoadDriver，但是无法绕过“映像加载通告例程”，所以用此方法监控驱动加载是最合适的了。

首先看看此函数的原型：

```
NTSTATUS PsSetLoadImageNotifyRoutine(PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine);
```

其中 NotifyRoutine 是一个函数指针，此回调函数的原型是：

```
VOID (*PLOAD_IMAGE_NOTIFY_ROUTINE)
(
    __in_opt PUNICODE_STRING FullImageName,
    __in HANDLE ProcessId,
    __in PIMAGE_INFO ImageInfo
);
```

回调函数的前两个参数显而易见，分别是映像的路径和加载此映像的进程 ID，第三个参数包含了更加详细的信息。

```
typedef struct _IMAGE_INFO {
    union {
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode : 8; //code addressing mode
            ULONG SystemModeImage : 1; //system mode image
            ULONG ImageMappedToAllPids : 1; //mapped in all processes
            ULONG Reserved : 22;
        };
    };
};

PVOID ImageBase;
ULONG ImageSelector;
ULONG ImageSize;
ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```



不过此结构体到了 Vista 之后，发生了一点变化。

```
typedef struct _IMAGE_INFO {
    union {
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode : 8; // Code addressing mode
            ULONG SystemModeImage      : 1; // System mode image
            ULONG ImageMappedToAllPids : 1; // Image mapped into all processes
            ULONG ExtendedInfoPresent  : 1; // IMAGE_INFO_EX available
            ULONG Reserved              : 21;
        };
    };
    PVOID ImageBase;
    ULONG ImageSelector;
    SIZE_T ImageSize;
    ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

当 ExtendedInfoPresent 标志非零时，IMAGE\_INFO 结构体被包含在了另外一个更大的结构体里。

```
typedef struct _IMAGE_INFO_EX {
    SIZE_T      Size;
    IMAGE_INFO  ImageInfo;
    struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

不过这个变动与实现监控驱动加载的关系不大，我们只需要 IMAGE\_INFO 的信息即可实现监控驱动加载。下面先讲解如何添加和删除“映像加载通告例程”。

```
//添加
PsSetLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)LoadImageNotifyRoutine);
//删除
PsRemoveLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)LoadImageNotifyRoutine);
```

接下来讲如何获得加载驱动的信息。之前说过，这个通告例程不仅负责处理加载驱动，连进程加载 DLL 也负责，那我们怎么判断到底是加载驱动还是加载 DLL 呢？根据后缀名判断，很明显是一个很不好的方法。我的方法是，根据回调函数 LoadImageNotifyRoutine 的第二个参数判断，如果 PID 是 0，则表示加载驱动，如果 PID 位非零，则表示加载 DLL。原因很简单，之前说过这个函数很底层，到了一定的深度之后，就无法判断到底是谁主动引发的行



为了，一切都是系统的行为。当然，也可以认为这是通过回调来监控驱动加载的缺点。判断是驱动后，就通过 ImageInfo->ImageBase 来获取驱动的映像基址。如果不想让这个驱动加载，就通过 ImageBase 来获得 DriverEntry 的地址，写入如下汇编的机器码：

```
Mov eax,c0000022h    B8 22 00 00 C0
Ret                  C3
```

实现代码如下：

```
PVOID GetDriverEntryByImageBase(PVOID ImageBase)
{
    PIMAGE_DOS_HEADER pDOSHeader;
    PIMAGE_NT_HEADERS64 pNTHHeader;
    PVOID pEntryPoint;
    pDOSHeader = (PIMAGE_DOS_HEADER)ImageBase;
    pNTHHeader = (PIMAGE_NT_HEADERS64)((ULONG64)ImageBase + pDOSHeader->e_lfanew);
    pEntryPoint = (PVOID)((ULONG64)ImageBase +
    pNTHHeader->OptionalHeader.AddressOfEntryPoint);
    return pEntryPoint;
}

void DenyLoadDriver(PVOID DriverEntry)
{
    UCHAR fuck[]="\xB8\x22\x00\x00\xC0\xC3";
    VxkCopyMemory(DriverEntry, fuck, sizeof(fuck));
}

VOID LoadImageNotifyRoutine
(
    __in_opt PUNICODE_STRING FullImageName,
    __in HANDLE ProcessId,
    __in PIMAGE_INFO ImageInfo
)
{
    PVOID pDrvEntry;
    char szFullImageName[260]={0};
    if(FullImageName!=NULL && MmIsAddressValid(FullImageName))
    {
        if(ProcessId==0)
        {
            DbgPrint("[LoadImageNotifyX64]%wZ\n", FullImageName);
            pDrvEntry=GetDriverEntryByImageBase(ImageInfo->ImageBase);
            DbgPrint("[LoadImageNotifyX64]DriverEntry: %p\n", pDrvEntry);
            UnicodeToChar(FullImageName, szFullImageName);
            if(strstr(_strlwr(szFullImageName), "win64ast.sys"))
            {
```

```
DbgPrint("Deny load [WIN64AST.SYS]");  
//禁止加载 win64ast.sys  
DenyLoadDriver(pDrvEntry);  
}  
}  
}  
}
```

有些读者心中可能想问，为什么拒绝加载驱动处仍然是“mov eax, c000022h”而不“mov rax, c000022h”？这是因为 NTSTATUS 其实就是 long 的马甲，而 long 的长度在 Win64 系统下依然是 4 字节而不是 8 字节，所以用“mov eax”足矣。如果对通过 ImageBase 获得 DriverEntry 不理解，可以参考我以前的拙文《初步探索 PE32+格式文件》。最后实现的效果如图 1 所示，效果为监视所有的驱动加载并拒绝名为 win64ast.sys 的驱动加载。

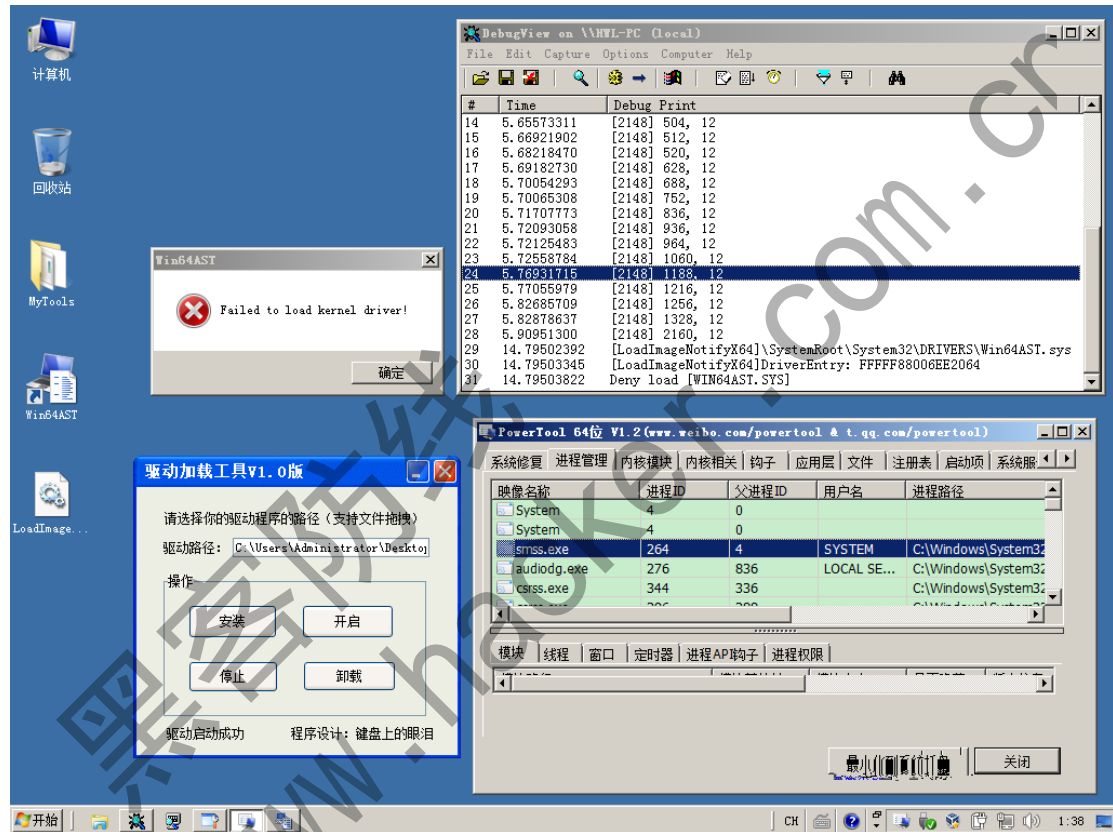


图 1

## MD5 算法设计与编程实现

文/图 张少飞

加密技术与我们的日常生活息息相关，在信息社会更是凸显重要。本文将主要就 MD5 算法及密码学算法实现做一些相关探讨。MD5 的全称是 message-digest algorithm 5（信息摘要算法）。由于其使用不需要支付任何版权费用，安全性好，所以 MD5 成为了当今非常流行的优秀的典型 Hash 加密技术。为了使自己能够对 MD5 加深理解，本文将利用 VC++ 编程



实现 MD5 加密过程，在软件实现上，提供了友好的界面，能够实现字符串和文件加密。

### 算法描述

算法输入一个字节串，每个字节 8 个 bit，算法的执行分为以下几个步骤：

第一步，补位。

MD5 算法先对输入的数据进行补位，使得数据的长度（以 byte 为单位）对 64 求余的结果是 56。即数据扩展至  $LEN=K*64+56$  个字节，K 为整数。

补位方法：补一个 1，然后补 0 至满足上述要求。相当于补一个 0x80 的字节，再补值为 0 的字节。这一步总共补充的字节数为 0~63 个。

第二步，附加数据长度。

用一个 64 位的整数表示数据的原始长度（以 bit 为单位），将这个数字的 8 个字节按低位在前高位在后的顺序附加在补位后的数据后面。此时，数据被填补后的总长度为： $LEN = K*64+56+8=(K+1)*64$ Bytes。注意，64 位整数是输入数据的原始长度，而不是填充字节后的长度。

第三步，初始化 MD5 参数。

有 4 个 32 位整数变量（A、B、C、D）用来计算信息摘要，每一个变量被初始化成以下以十六进制数表示的数值，低位字节在前面。

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

注意，低位字节在前面指的是 Little Endian 平台上内存中字节的排列方式，而在程序中书写时，要写成。

A=0x67452301

B=0xefcdab89

C=0x98badcfe

D=0x10325476

第四步，定义 4 个 MD5 基本的按位操作函数。

X、Y、Z 为 32 位整数。

$F(X,Y,Z) = (X \text{ and } Y) \text{ or } (\text{not}(X) \text{ and } Z)$

$G(X,Y,Z) = (X \text{ and } Z) \text{ or } (Y \text{ and } \text{not}(Z))$

$H(X,Y,Z) = X \text{ xor } Y \text{ xor } Z$

$I(X,Y,Z) = Y \text{ xor } (X \text{ or } \text{not}(Z))$

再定义 4 个分别用于四轮变换的函数。

设  $M_j$  表示消息的第 j 个子分组（从 0 到 15）， $\lll s$  表示循环左移 s 位，则四种操作为：

FF(a,b,c,d,Mj,s,ti)表示  $a=b+((a+(F(b,c,d)+Mj+ti)\lll s)$

GG(a,b,c,d,Mj,s,ti)表示  $a=b+((a+(G(b,c,d)+Mj+ti)\lll s)$

HH(a,b,c,d,Mj,s,ti)表示  $a=b+((a+(H(b,c,d)+Mj+ti)\lll s)$

II(a,b,c,d,Mj,s,ti)表示  $a=b+((a+(I(b,c,d)+Mj+ti)\lll s)$

第五步，对输入数据作变换。

处理数据，N 是总的字节数，以 64 个字节为一组，每组作一次循环，每次循环进行四轮操作。要变换的 64 个字节用 16 个 32 位的整数数组  $M[0...15]$  表示，而数组  $T[1...64]$  表示一组常数， $T[i]$  为  $4294967296*\text{abs}(\sin(i))$  的 32 位整数部分，i 的单位是弧度，i 的取值从 1 到 64。

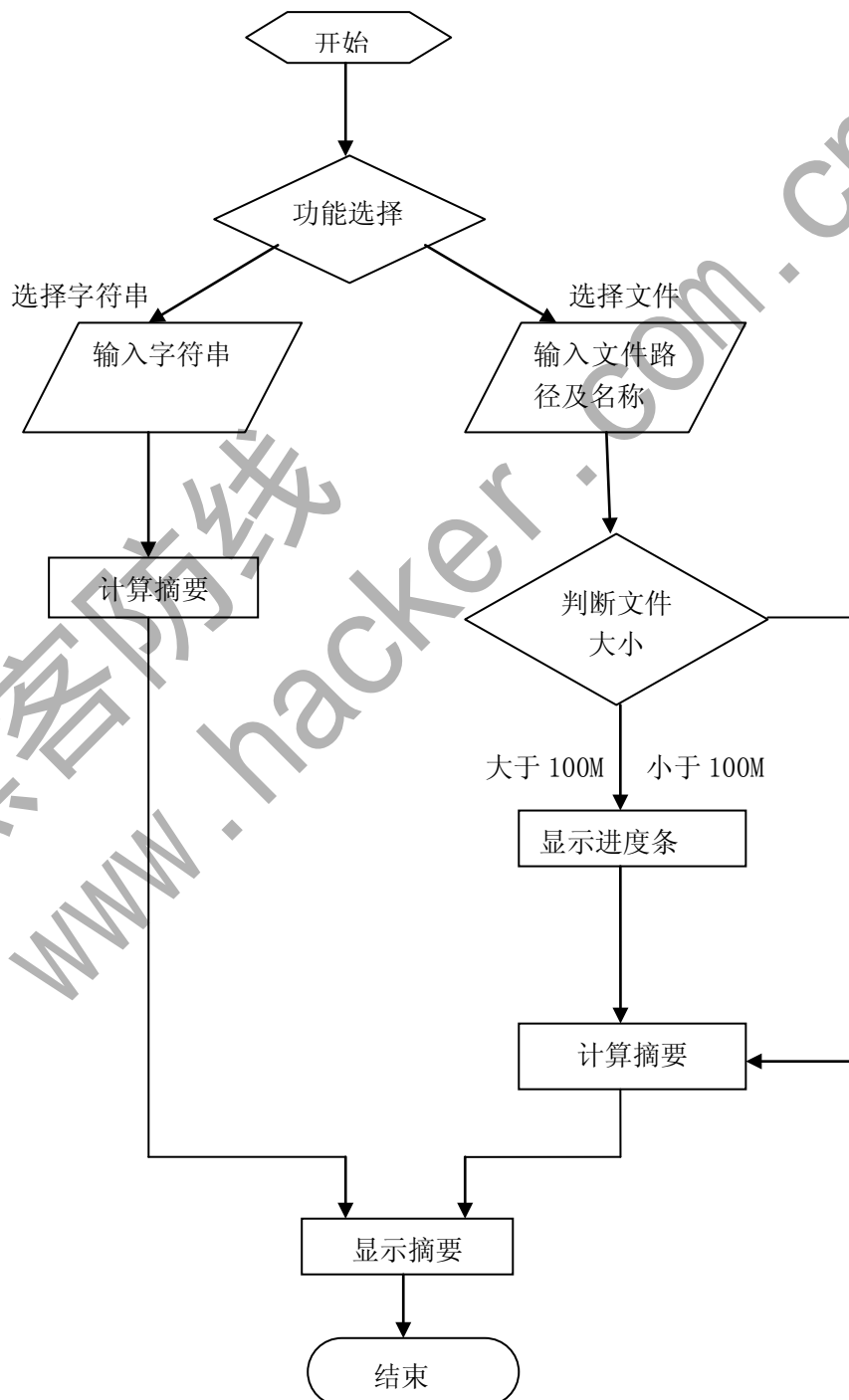


### 设计任务

在设计任务中，加密软件具有以下功能：

- A: 加密软件具有良好的用户界面；
- B: 能够对字符串进行加密；
- C: 能够对文件进行加密；
- D: 对于文件加密大于 10M 的文件能够显示进度条；
- F: 在没有选择加密类型时，必须先选择才能进行加密；
- H: 界面要简练，输出输入表示要明确，尽量做到通俗易懂。

### 系统流程





## 部分代码说明

本程序主要由 ZMD5.cpp 和 MyMD5Dlg.cpp 文件构成,前者是 MD5 算法的实现,后者用于界面的控制和文件选择。以下是这两个文件的关键代码。

### 1. ZMD5.cpp 关键代码

```
unsigned int ZMD5::ROTATE_LEFT(unsigned int x,unsigned int n)
{
    return (((x) << (n)) | ((x) >> (32-(n))));
}
unsigned int ZMD5::F(unsigned int x,unsigned int y,unsigned int z)
{
    return ((x & y) | ((~x) & z));
}
unsigned int ZMD5::G(unsigned int x,unsigned int y,unsigned int z)
{
    return ((x & z) | (y & (~z)));
}
unsigned int ZMD5::H(unsigned int x,unsigned int y,unsigned int z)
{
    return x ^ y ^ z;
}
unsigned int ZMD5::I(unsigned int x,unsigned int y,unsigned int z)
{
    return (y ^ (x | (~z)));
}

void ZMD5::FF(unsigned int& a,unsigned int b,unsigned int c,unsigned int
d,unsigned int x,int s,unsigned int ac)
{
    (a) += F ((b), (c), (d)) + (x) + (ac);
    (a) = ROTATE_LEFT ((a), (s));
    (a) += (b);
}

void ZMD5::GG(unsigned int& a,unsigned int b,unsigned int c,unsigned int
d,unsigned int x,int s,unsigned int ac)
{
    (a) += G ((b), (c), (d)) + (x) + (ac);
    (a) = ROTATE_LEFT ((a), (s));
    (a) += (b);
}

void ZMD5::HH(unsigned int& a,unsigned int b,unsigned int c,unsigned int
d,unsigned int x,int s,unsigned int ac)
```





```
{
    (a) += H ((b), (c), (d)) + (x) + (ac);
    (a) = ROTATE_LEFT ((a), (s));
    (a) += (b);
}

void ZMD5::II(unsigned int& a,unsigned int b,unsigned int c,unsigned int
d,unsigned int x,int s,unsigned int ac)
{
    (a) += I ((b), (c), (d)) + (x) + (ac);
    (a) = ROTATE_LEFT ((a), (s));
    (a) += (b);
}

void ZMD5::Init()
{
    S11 = 7;   S21 = 5;   S31 = 4;   S41 = 6;
    S12 = 12;  S22 = 9;   S32 = 11;  S42 = 10;
    S13 = 17;  S23 = 14;  S33 = 16;  S43 = 15;
    S14 = 22;  S24 = 20;  S34 = 23;  S44 = 21;
    A = 0x67452301; // in memory, this is 0x01234567
    B = 0xEFCDA89; // in memory, this is 0x89ABCDEF
    C = 0x98BADCFE; // in memory, this is 0xFEDCBA98
    D = 0x10325476; // in memory, this is 0x76543210
}

void ZMD5::Append(unsigned int MsgLen)
{
    //计算要补位的字节数
    int m = MsgLen % 64;
    if(m==0)
        m_AppendByte=56;
    else if(m<56)
        m_AppendByte=56-m;
    else
        m_AppendByte=64-m+56;
    //截取传入长度的高十六位和低十六位
    int hWord=(MsgLen & 0xFFFF0000) >> 16;
    int lWord=MsgLen & 0x0000FFFF;
    //将低十六位和高十六位分别乘以八(1byte=8bit)
    int hDiv=hWord*8;
    int lDiv=lWord*8;
    m_MsgLen[0] = lDiv & 0xFF ;
    m_MsgLen[1] = (lDiv >> 8) & 0xFF ;
    m_MsgLen[2] = ((lDiv >> 16) & 0xFF) | (hDiv & 0xFF);
    m_MsgLen[3] = (hDiv >> 8) & 0xFF ;
    m_MsgLen[4] = (hDiv >> 16) & 0xFF ;
}
```



```
m_MsgLen[5] = (hDiv >> 24) & 0xFF ;
m_MsgLen[6] = 0;
m_MsgLen[7] = 0;
}
void ZMD5::Transform(unsigned char Block[64])
{
    //将 64 字节位转换为 16 个字节
    unsigned long x[16];
    for (int i=0, j=0; j<64; i++, j+=4)
x[i]=Block[j] | Block[j+1] <<8 | Block[j+2] <<16 | Block[j+3] <<24 ;
    //初始化临时寄存器变量
    unsigned int a, b, c, d;
    a=A; b=B; c=C; d=D;
    //第一轮计算
    FF (a, b, c, d, x[ 0], S11, 0xD76AA478); // 1
    FF (d, a, b, c, x[ 1], S12, 0xE8C7B756); // 2
    FF (c, d, a, b, x[ 2], S13, 0x242070DB); // 3
    FF (b, c, d, a, x[ 3], S14, 0xC1BDCEEE); // 4
    FF (a, b, c, d, x[ 4], S11, 0xF57C0FAF); // 5
    FF (d, a, b, c, x[ 5], S12, 0x4787C62A); // 6
    FF (c, d, a, b, x[ 6], S13, 0xA8304613); // 7
    FF (b, c, d, a, x[ 7], S14, 0xFD469501); // 8
    FF (a, b, c, d, x[ 8], S11, 0x698098D8); // 9
    FF (d, a, b, c, x[ 9], S12, 0x8B44F7AF); // 10
    FF (c, d, a, b, x[10], S13, 0xFFFF5BB1); // 11
    FF (b, c, d, a, x[11], S14, 0x895CD7BE); // 12
    FF (a, b, c, d, x[12], S11, 0x6B901122); // 13
    FF (d, a, b, c, x[13], S12, 0xFD987193); // 14
    FF (c, d, a, b, x[14], S13, 0xA679438E); // 15
    FF (b, c, d, a, x[15], S14, 0x49B40821); // 16
    //第二轮计算
    GG (a, b, c, d, x[ 1], S21, 0xF61E2562); // 17
    GG (d, a, b, c, x[ 6], S22, 0xC040B340); // 18
    GG (c, d, a, b, x[11], S23, 0x265E5A51); // 19
    GG (b, c, d, a, x[ 0], S24, 0xE9B6C7AA); // 20
    GG (a, b, c, d, x[ 5], S21, 0xD62F105D); // 21
    GG (d, a, b, c, x[10], S22, 0x2441453); // 22
    GG (c, d, a, b, x[15], S23, 0xD8A1E681); // 23
    GG (b, c, d, a, x[ 4], S24, 0xE7D3FBC8); // 24
    GG (a, b, c, d, x[ 9], S21, 0x21E1CDE6); // 25
    GG (d, a, b, c, x[14], S22, 0xC33707D6); // 26
    GG (c, d, a, b, x[ 3], S23, 0xF4D50D87); // 27
    GG (b, c, d, a, x[ 8], S24, 0x455A14ED); // 28
    GG (a, b, c, d, x[13], S21, 0xA9E3E905); // 29
```



```
GG (d, a, b, c, x[ 2], S22, 0xFCEFA3F8); // 30
GG (c, d, a, b, x[ 7], S23, 0x676F02D9); // 31
GG (b, c, d, a, x[12], S24, 0x8D2A4C8A); // 32
//第三轮计算
HH (a, b, c, d, x[ 5], S31, 0xFFFFA3942); // 33
HH (d, a, b, c, x[ 8], S32, 0x8771F681); // 34
HH (c, d, a, b, x[11], S33, 0x6D9D6122); // 35
HH (b, c, d, a, x[14], S34, 0xFDE5380C); // 36
HH (a, b, c, d, x[ 1], S31, 0xA4BEEA44); // 37
HH (d, a, b, c, x[ 4], S32, 0x4BDECF A9); // 38
HH (c, d, a, b, x[ 7], S33, 0xF6BB4B60); // 39
HH (b, c, d, a, x[10], S34, 0xBEBFBC70); // 40
HH (a, b, c, d, x[13], S31, 0x289B7EC6); // 41
HH (d, a, b, c, x[ 0], S32, 0xEAA127FA); // 42
HH (c, d, a, b, x[ 3], S33, 0xD4EF3085); // 43
HH (b, c, d, a, x[ 6], S34, 0x4881D05); // 44
HH (a, b, c, d, x[ 9], S31, 0xD9D4D039); // 45
HH (d, a, b, c, x[12], S32, 0xE6DB99E5); // 46
HH (c, d, a, b, x[15], S33, 0x1FA27CF8); // 47
HH (b, c, d, a, x[ 2], S34, 0xC4AC5665); // 48
//第四轮计算
II (a, b, c, d, x[ 0], S41, 0xF4292244); // 49
II (d, a, b, c, x[ 7], S42, 0x432AFF97); // 50
II (c, d, a, b, x[14], S43, 0xAB9423A7); // 51
II (b, c, d, a, x[ 5], S44, 0xFC93A039); // 52
II (a, b, c, d, x[12], S41, 0x655B59C3); // 53
II (d, a, b, c, x[ 3], S42, 0x8F0CCC92); // 54
II (c, d, a, b, x[10], S43, 0xFPEFF47D); // 55
II (b, c, d, a, x[ 1], S44, 0x85845DD1); // 56
II (a, b, c, d, x[ 8], S41, 0x6FA87E4F); // 57
II (d, a, b, c, x[15], S42, 0xFE2CE6E0); // 58
II (c, d, a, b, x[ 6], S43, 0xA3014314); // 59
II (b, c, d, a, x[13], S44, 0x4E0811A1); // 60
II (a, b, c, d, x[ 4], S41, 0xF7537E82); // 61
II (d, a, b, c, x[11], S42, 0xBD3AF235); // 62
II (c, d, a, b, x[ 2], S43, 0x2AD7D2BB); // 63
II (b, c, d, a, x[ 9], S44, 0xEB86D391); // 64
//保存当前寄存器结果
A+=a; B+=b; C+=c; D+=d;
}

string ZMD5::ToHex(bool UpperCase)
{
    string strResult;
```



```
int ResultArray[4]={A, B, C, D};
char Buf[33]={0};
for(int i=0;i<4;i++)
{
    memset(Buf, 0, 3);
    sprintf(Buf, "%02x", ResultArray[i] & 0x00FF);
    strResult+=Buf;
    memset(Buf, 0, 3);
    sprintf(Buf, "%02x", (ResultArray[i] >> 8) & 0x00FF);
    strResult+=Buf;
    memset(Buf, 0, 3);
    sprintf(Buf, "%02x", (ResultArray[i] >> 16) & 0x00FF);
    strResult+=Buf;
    memset(Buf, 0, 3);
    sprintf(Buf, "%02x", (ResultArray[i] >> 24) & 0x00FF);
    strResult+=Buf;
}
if(UpperCase) CharUpper((char *)strResult.c_str());
return strResult;
}

string ZMD5::GetMD5OfString(string InputMessage, bool UpperCase)
{
    //初始化 MD5 所需常量
    Init();
    //计算追加长度
    Append(InputMessage.length());
    //对原始信息进行补位
    for(int i=0;i<m_AppendByte;i++)
    {
        if(i==0) InputMessage+=(unsigned char)0x80;
        else InputMessage+=(unsigned char)0x0;
    }
    //将原始信息长度附加在补位后的数据后面
    for(int i=0;i<8;i++) InputMessage+=m_MsgLen[i];
    //位块数组
    unsigned char x[64]={0};
    //循环, 将原始信息以 64 字节为一组拆分进行处理
    for(int i=0, Index=-1;i<InputMessage.length();i++)
    {
        x[++Index]=InputMessage[i];
        if(Index==63)
        {
            Index=-1;

```



```

        Transform(x);
    }
}
return ToHex(UpperCase);
}
string ZMD5::GetMD5OfFile(const string FileName, bool UpperCase)
{
    //定义读取文件的缓冲区
    char* ReadBuf =new char[FILE_BUFFER_READ+1];
    memset(ReadBuf, 0, FILE_BUFFER_READ);
    try
    {
        //检查文件是否存在
        if((_access(FileName.c_str(),0 )) == -1) return "";
        //二进制方式读取文件
        if(m_pFile=fopen(FileName.c_str(),"rb"),m_pFile==NULL) return "";
        m_FileOpen=true;
        //获取文件大小
        unsigned long FileSize=0xFFFF;
        WIN32_FIND_DATA win32_find_data;
        HANDLE hFile;
        if((hFile=FindFirstFile(FileName.c_str(),&win32_find_data))!=INVALID_HANDLE
        _VALUE)
            if(hFile==NULL) return "";
            if(FileSize=win32_find_data.nFileSizeLow,FileSize==0xFFFF ||
        FileSize==0) return "";
            FindClose(hFile);
            //初始化 MD5 所需常量
            Init();
            //通过文件长度计算追加长度
            Append(FileSize);
            //位块数组
            unsigned char x[64]={0};
            //本次读取字节数
            int ReadSize=fread(ReadBuf, 1, FILE_BUFFER_READ, m_pFile);
            //读取次数
            int ReadCount=0;
            while(ReadSize==FILE_BUFFER_READ)
            {
                /*
                如果用户开启了另一个线程调用此函数，则允许用户从外部结束此函数。
                为安全起见，没有在这个类的内部开启线程，可以最大限度的保证文件安全关闭。
                */
                if(!m_FileOpen)
    
```

```
{
    fclose(m_pFile);
    return "";
}
//将处理进度返回给用户
ReadCount++;
OnProcessing((int)(FILE_BUFFER_READ * ReadCount / (FileSize /
100)));

//将原始信息以 64 字节为一组拆分进行处理
for(int i=0, Index=-1; i<FILE_BUFFER_READ; i++)
{
    x[++Index]=ReadBuf[i];
    if(Index==63)
    {
        Index=-1;
        Transform(x);
    }
}
memset(ReadBuf, 0, FILE_BUFFER_READ); // 重置缓冲区
ReadSize=fread(ReadBuf, 1, FILE_BUFFER_READ, m_pFile);
} // end while
/*处理不能被整除的剩余部分数据,此时要对剩余部分数据进行补位及长原始
信息长度追加。如果最后一次读取数据的长度为零,说明文件已被读完,则直接将补位数据
及原信息长度送入 Transform 处理。*/
if(ReadSize==0)
{
    string strData;
    for(int i=0; i<m_AppendByte; i++)
    {
        if(i==0) strData+=(unsigned char)0x80;
        else strData+=(unsigned char)0x0;
    }
    for(int i=0; i<8; i++) strData+=m_MsgLen[i];
    for(int i=0, Index=-1; i<strData.length(); i++)
    {
        x[++Index]=strData[i];
        if(Index==63)
        {
            Index=-1;
            Transform(x);
        }
    }
}
else //将剩余数据处理完再补位
```

```
{
    for(int i=0, Index=-1; i<ReadSize+m_AppendByte+8; i++)
    {
        //将原始信息以 64 字节为一组，进行拆分处理
        if(i<ReadSize)
            x[++Index]=ReadBuf[i];
        else if(i==ReadSize)
            x[++Index]=(unsigned char)0x80;
        else if(i<ReadSize+m_AppendByte)
            x[++Index]=(unsigned char)0x0;
        else if(i==ReadSize+m_AppendByte)
            x[++Index]=m_MsgLen[0];
        else if(i==ReadSize+m_AppendByte+1)
            x[++Index]=m_MsgLen[1];
        else if(i==ReadSize+m_AppendByte+2)
            x[++Index]=m_MsgLen[2];
        else if(i==ReadSize+m_AppendByte+3)
            x[++Index]=m_MsgLen[3];
        else if(i==ReadSize+m_AppendByte+4)
            x[++Index]=m_MsgLen[4];
        else if(i==ReadSize+m_AppendByte+5)
            x[++Index]=m_MsgLen[5];
        else if(i==ReadSize+m_AppendByte+6)
            x[++Index]=m_MsgLen[6];
        else if(i==ReadSize+m_AppendByte+7)
            x[++Index]=m_MsgLen[7];
        if(Index==63)
        {
            Index=-1;
            Transform(x);
        }
    }
    OnProcessing(100); //处理进度百分之百
    fclose(m_pFile); //关闭文件
    m_FileOpen=false; //文件打开状态为 false
    delete[] ReadBuf; //释放动态申请的内存
}
catch(...)
{
    if(m_FileOpen)
    fclose(m_pFile); //关闭文件
    m_FileOpen=false; //文件打开状态为 false
    delete[] ReadBuf; //释放动态申请的内存
```



```
        return "";\n    }\n    return ToHex(UpperCase);\n}\nvoid ZMD5::GetMD5OfFile_Terminate()\n{\n    if(m_FileOpen) m_FileOpen=false;\n}\n
```

## 2. MyMD5Dlg.cpp 关键代码

```
void CMyMD5Dlg::OnBtnCalculation()\n{\n    //取“计算”按钮文字，如果上面的文字为“停止”，则停止当前信息的MD5计算\n    CString strButton;\n    int selected;\n    selected=GetCheckedRadioButton(IDC_RADIO_1, IDC_RADIO_2);\n    GetDlgItemText(IDC_BTN_CALCULATION, strButton);\n    if(strButton=="停止")\n    {\n        SetDlgItemText(IDC_EDIT_RESULT, "");\n        md5.GetMD5OfFile_Terminate();\n        SetDlgItemText(IDC_BTN_CALCULATION, "计算");\n        ((CProgressCtrl*)GetDlgItem(IDC_PROGRESS))->ShowWindow(SW_HIDE);\n        return;\n    }\n    //清空结果文本框\n    SetDlgItemText(IDC_EDIT_RESULT, "");\n    //获取待计算MD5的文件数据\n    CString strInput;\n    GetDlgItemText(IDC_EDIT_SOURCE, strInput);\n    //获取待计算MD5的字符串数据\n    CString strInput2;\n    GetDlgItemText(IDC_EDIT_SOURCE2, strInput2);\n    //如果原信息类型为字符串\n    if(selected==IDC_RADIO_1)\n    {\n        SetDlgItemText(IDC_EDIT_RESULT,\n            md5.GetMD5OfString((LPSTR) (LPCTSTR) strInput2, true).c_str());\n    }\n    //如果原信息类型为文件\n    else if(selected==IDC_RADIO_2)\n    {\n        //如果文件为空，显示选择文件对话框\n    }\n}
```





```
if(m_File.empty())
{
    OnBtnSelect();
    return ;
}
//检查文件是否存在
else if( _access((char*)(LPCTSTR)strInput, 0 )) == -1 )
{
    AfxMessageBox("文件不存在!");
    return ;
}
else
{
    //计算文件大小
    unsigned long FileSize=0xFFFFFFFF;
    WIN32_FIND_DATA win32_find_data;
    HANDLE hFile;

    if((hFile=FindFirstFile((char*)(LPCTSTR)strInput,&win32_find_data))!=INVALID_HANDLE_VALUE)
    {
        FindClose(hFile);

        if(FileSize=win32_find_data.nFileSizeLow,FileSize==0xFFFFFFFF||FileSize==0)
            return ;
    }
    //文件大于等于 10M, 显示处理进度条
    if(FileSize>=10485760)
    {
        //显示处理进度条
        m_Processing.SetPos(0);

        ((CProgressCtrl*)GetDlgItem(IDC_PROGRESS))->ShowWindow(SW_SHOW);

        //将“计算”按钮上的文字变成“停止”
        SetDlgItemText(IDC_BTN_CALCULATION,"停止");
        //开启一个线程用于计算 MD5, 以免造成假死现象
        hThread= CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)m_thunk.CallBack(this,
            &CMyMD5Dlg::ProcessingThread, ZThunk::THISCALL),
            0, 0, &dwThreadId);
    }
    //文件小于 10M, 直接进行 MD5 计算, 不开启线程, 因为计算时间很短
    else
```



```
        {
            SetDlgItemText(IDC_EDIT_RESULT,
                md5.GetMD5OfFile((LPCTSTR) (LPCTSTR) strInput, true).c_str());
        }
    }
}
else
    AfxMessageBox("请选择输入类型,“字符串”或“文件”!");
}

void CMyMD5Dlg::OnBtnAbout()
{
    CAboutDlg dlg;
    dlg.DoModal();
}
//处理 WM_RETURN (回车) 和 WM_VK_ESCAPE (取消)
BOOL CMyMD5Dlg::PreTranslateMessage(MSG* pMsg)
{
    if(pMsg->message==WM_KEYDOWN)
    {
        switch(pMsg->wParam)
        {
            case VK_ESCAPE:
                return TRUE;
            case VK_RETURN:
                return TRUE;
            default:
                break;
        }
    }
    return CDialog::PreTranslateMessage(pMsg);
}
DWORD CMyMD5Dlg::ProcessingThread(LPVOID lpParameter)
{
    CString strInput;
    GetDlgItemText(IDC_EDIT_SOURCE, strInput);
    SetDlgItemText(IDC_EDIT_RESULT,
        md5.GetMD5OfFile((LPCTSTR) (LPCTSTR) strInput, true).c_str());
    return 1;
}
LRESULT CMyMD5Dlg::OnProcessing(WPARAM wParam, LPARAM lParam)
{
    m_Processing.SetPos((int)wParam);
    if((int)wParam==100)
```



```

    {
        //隐藏 CProcessCtrl 控件
        ((CProgressCtrl*)GetDlgItem(IDC_PROGRESS))->ShowWindow(SW_HIDE);
        //将“停止”按钮上的文字变成“计算”
        SetDlgItemText(IDC_BTN_CALCULATION,“计算”);
    }
    return 1;
}
//选择字符串加密功能
1 void CMyMD5Dlg::OnRadio_1()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    GetDlgItem(IDC_OPEN)->EnableWindow(false);
    GetDlgItem(IDC_EDIT_SOURCE)->EnableWindow(false);
    GetDlgItem(IDC_EDIT_SOURCE2)->EnableWindow(true);
    SetDlgItemText(IDC_EDIT_SOURCE2,“”);
}
//选择文件加密功能
void CMyMD5Dlg::OnRadio_2()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    GetDlgItem(IDC_OPEN)->EnableWindow(true);
    GetDlgItem(IDC_EDIT_SOURCE)->EnableWindow(true);
    GetDlgItem(IDC_EDIT_SOURCE2)->EnableWindow(false);
    SetDlgItemText(IDC_EDIT_SOURCE,“”);
}
//文件打开按钮
void CMyMD5Dlg::OnOpen()
{
    // TODO: Add your control notification handler code here
    ZFileDialog fileDlg;
    deque<string> dqSelectFile=fileDlg.GetOpenFileName(FALSE,“所有文件
    (*.*)\0*.*\0\0”);
    if(dqSelectFile.size() !=0)
    {
        m_File=dqSelectFile[0];
        SetDlgItemText(IDC_EDIT_SOURCE,m_File.c_str());
    }
}
}

```

### 系统运行说明

本文件不需要安装，直接运行文件夹里面的 MyMD5.exe 文件，就会打开如图 1 所示的界面，

新手可以先看说明，如果对本程序已经了解，可以直接运行程序。接下来要做的就是选择输入的类型，当选择文件加密并且大小大于 10M 的时候，会出现如图 2 所示的界面（其他两种：如字符串和文件小于 10M 的请自己体验），最终的计算结果如图 3 所示。

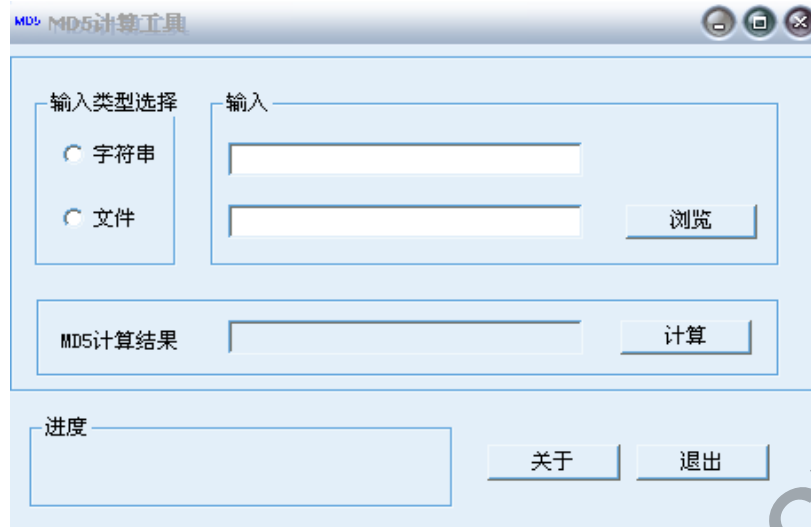


图 1

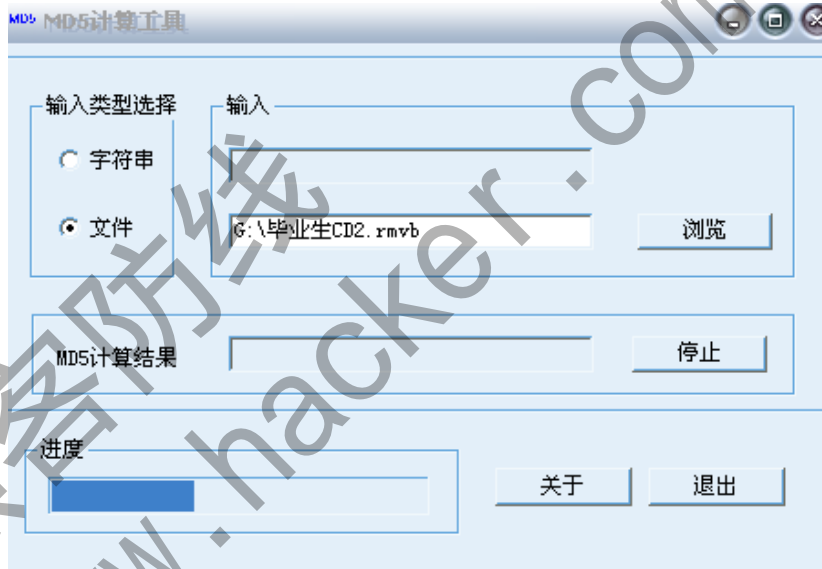


图 2

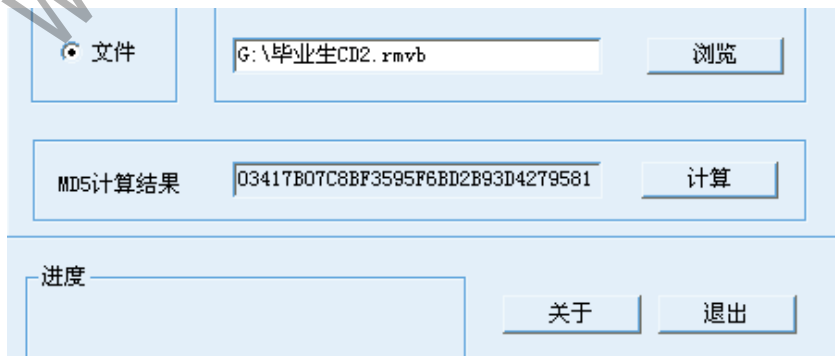


图 3



## 系统测试

### 1. 正确性测试

#### 1) 字符串计算

输入字符串: 123456789

计算结果为: 25F9E794323B453885F5181F1B624D0B

解密结果: 123456789

证明本程序能够正确的计算字符串的摘要。

#### 2) 文件计算

在记事本中输入字符串“123456789”，保存为：“1.txt”。

计算该文件结果为: 25F9E794323B453885F5181F1B624D0B。

### 2. 性能测试

本程序能较快的完成摘要计算功能，对 100M 大小的文件，计算时间大约为 3 秒钟，对系统资源占用较小。

---

# Python 黑客编程：网站后台暴力破解

文/图 blackcool

在 Web 安全测试中，敏感路径及文件的探测是经常用到的技术，一般的扫描器如 AWVA、APPSCAN、JSKY 等都具有这个功能，但是各款扫描器中的路径及敏感文件字典各有不同，而扫描成功的主要因素就是字典的匹配度及工具的灵活性。成型的产品功能比较全面，运行起来也比较耗时，因此笔者就有了一个简单想法，用 Python 实现一个轻量级的网站后台暴力破解工具，这样我们就可以加载自己的字典来对网站后台进行猜解了。

扫描原理比较简单：首先组装敏感 URL，然后调用 urllib2.Request 发送请求，最后判断请求结果，如果结果为 urllib2.URLError 则退出，否则即为命中，并将命中结果输出。下面结合代码来看下实现过程。

主要函数是 urlcheck，用来完成对 URL 有效性的检测。

首先是构造敏感 URL，完整的路径主要有三部分：用户输入的域名+敏感路径字典+文件后缀。先将这三部分进行组装，实现代码如下：

```
def urlcheck(url,path,ext):
    ext='.'+ext
    req = urllib2.Request(url+path+ext)
```

得到完整 URL 后就可以进行请求测试了。先使用 urllib2.urlopen 发送 http 请求，当返回结果为 urllib2.URLError 时，返回结果为假，否则为真。

```
try:
    fp = urllib2.urlopen(req)
except urllib2.URLError:
    pass
    return (False,0)
else:
```

```
return (True,url+path+ext)
```

通过 `urlcheck` 函数就可以完成敏感路径的有效性验证了。

其他部分与之前《FTP暴力破解》的框架基本相似，就不多说了，完整实现代码如下：

```
#coding=utf-8

import urllib2
import sys
import getopt

def urlcheck(url,path,ext):
    ext='.'+ext
    req = urllib2.Request(url+path+ext)
    try:
        fp = urllib2.urlopen(req)
    except urllib2.URLError:
        pass
        return (False,0)
    else:
        return (True,url+path+ext)

def usage():
    print "[!] Usage : %s -u <url> -e <extend> " % sys.argv[0]
    print "[!] Example : %s -u http://www.google.com/ -e asp " % sys.argv[0]
    raise SystemExit

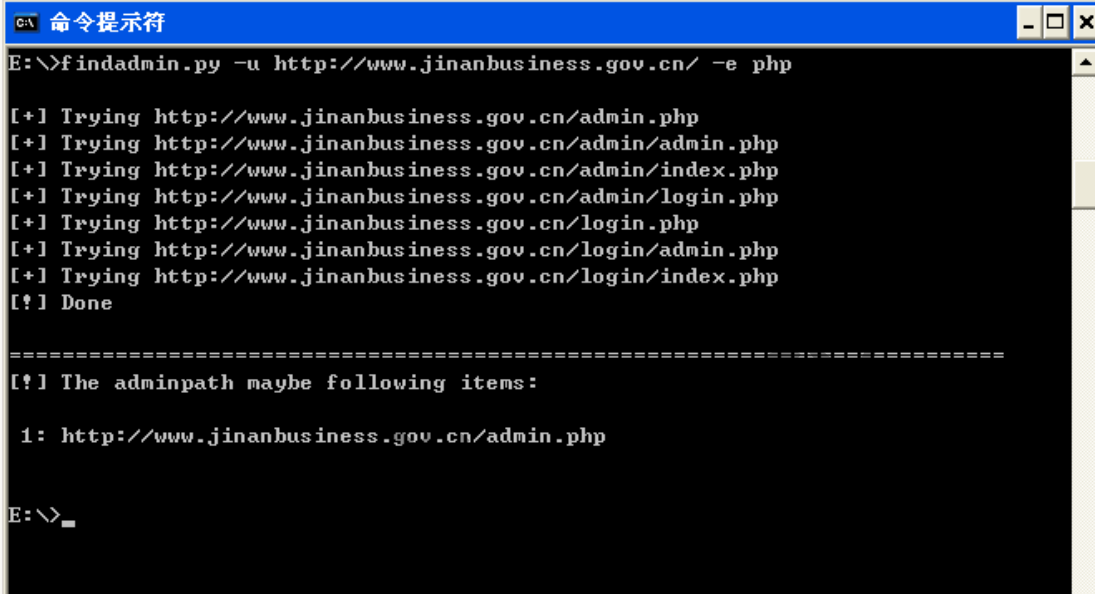
def main():
    _path_list=("admin","admin/admin","admin/index","admin/login","login","login/admin",
    "login/index",)
    opts,args = getopt.getopt(sys.argv[1:], 'e:u:')
    for o,a in opts:
        if o == '-u':
            url = a
        if o == '-e':
            ext = a
    _result = []
    print
    for path in _path_list:
        ck = urlcheck(url,path,ext)
        print "[+] Trying %s" % (url+path+'.'+ext)
        if ck[0]:
            _result.append(ck[1])
```

```
print "[!] Done\n"
print "=====
print "[!] The adminpath maybe following items:\n"
for index,result in enumerate(_result):
    print "%2d: %s"%(index+1,result)

print

if __name__ == "__main__":
    main()
```

下面我们测试下这段代码是否好用，结果如图 1 所示。



```
命令提示符
E:\>python findadmin.py -u http://www.jinanbusiness.gov.cn/ -e php

[+] Trying http://www.jinanbusiness.gov.cn/admin.php
[+] Trying http://www.jinanbusiness.gov.cn/admin/admin.php
[+] Trying http://www.jinanbusiness.gov.cn/admin/index.php
[+] Trying http://www.jinanbusiness.gov.cn/admin/login.php
[+] Trying http://www.jinanbusiness.gov.cn/login.php
[+] Trying http://www.jinanbusiness.gov.cn/login/admin.php
[+] Trying http://www.jinanbusiness.gov.cn/login/index.php
[!] Done

=====
[!] The adminpath maybe following items:

1: http://www.jinanbusiness.gov.cn/admin.php

E:\>
```

图 1

这样，利用 Python 编写一款简单的后台暴力破解器就完成了，希望以后有机会和大家交流分享更多更有趣的 python hacking 代码。

## 共享内存的奥秘

文/图 王晓松

在互联网兴盛之初，BBS 曾火爆一时，大家都可以在这块共享的领地分享自己的故事和心情。在计算机的世界里，如果把每个进程比作一个人，那么它们之间交流的手段又是什么呢？就是我们下面要介绍的 Windows 中的共享内存机制。

### 在应用层面共享内存的使用

我们知道，在 Windows 中每个进程的地址空间都是相互独立的。换句话说，对于两个进程同样的虚拟地址，背后映射的物理内存并不相同（内核地址空间例外）。但有时会有这样的需求，要将两个进程中不同或者相同的地址映射为同一块物理内存，这样做的目的可以是：

1) 这段内存是不可修改的可执行程序，两个进程使用同样的代码。简单的例子就是打

开两个 Word 进程，其处理的文档可能不同，但是 Word 本身的代码是可以共用的，代码段的物理内存可以共用。

2) 将一块相同的物理内存映射到不同的进程后，这两个进程可以通过这段物理内存实现进程之间的通信。

以上是共享内存标准的用法，但在实际应用中，共享内存还可以用于文件的映射，即将一个文件的一块区域映射到某一进程的地址空间内，该进程对文件的操作就转换为对内存（编程中表现为对数组）的操作，这样可以有效的减少 I/O 操作的数量，提高效率。

为了让大家对共享内存有个初步的概念，我们先看看编程世界里对共享内存的使用方法，标准步骤如下：

```
①hFile=CreateFile(pszFileName, ...); //创建或者打开文件对象
②hFileMap=CreateFileMapping(hFile, ...); //创建一个文件映射内核对象
③pbFile=MapViewOfFile(hFileMap, ...);
//将文件映射对象的部分或者全部映射到进程的地址空间
④pbFile[0]=1; //共享内存的使用
⑤UnMapViewOfFile(pbFile, ...); //取消上一步骤的映射
⑥CloseHandle(hFileMap); //关闭文件映射内核对象
⑦CloseHandle(hFile); //关闭文件内核对象
```

共七个步骤，如果细看，可以发现步骤①与步骤⑦相对应，步骤②与步骤⑥相对应，步骤③与步骤⑤相对应，都是互相对应的反向操作。

在第②步中，CreateFileMapping() 函数创建了一个文件映射内核对象，该函数使用一个由步骤①中 CreateFile() 函数打开并返回的文件句柄，但若是为了两个进程之间共享内存，此句柄可以设置为 0xFFFFFFFF，表示使用系统页面文件，则步骤①可以省略。该函数最后一个参数可以为内存映射对象指定名字，通过调用 CreateFileMapping 函数和 OpenFileMapping 函数，其他进程可用这个名字来访问相同的文件映像。

CreateFileMapping 函数创建成功文件映射内核对象后，步骤③调用 MapViewOfFile 函数，把文件的一块区域映射到进程地址空间上，调用这个函数需要指定文件映射对象、目标文件的起始地址、操作的数量等参数。

完成步骤③后，操作数组 pbFile 就等同于操作目标文件本身。当不再需要把文件的数据映射到进程空间时，调用步骤⑤中的 UnMapViewOfFile 函数解除映射，同时会将一些映射数据写入文件。最后在步骤⑥、⑦中，释放文件映射对象和文件对象。

如果上面的步骤通过编程，亲自实践，能够顺利完成的话，那么就是一个合格的 Windows 编程人员，但作为一名研究 Windows 内核的同仁，还要想的更多，在 CreateFileMapping 函数和 MapViewOfFile 函数背后，共享内存实现的奥秘是什么呢？

## 内核层面共享内存的实现

在最初学习这段内容的时候，看着 N 多个数据结构，N 多的连接关系，密密麻麻如八爪鱼般邪恶。为了使读者少一些痛苦，轻松一些，下面我将按照前面介绍共享内存使用的步骤①~③进行分步讲解。

### 1. CreateFile 的工作



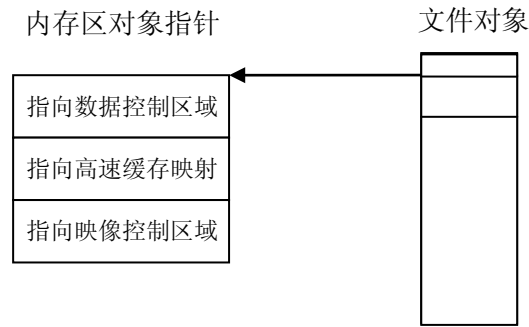


图 1 CreateFile 的工作

因为本文并不专门针对文件的操作，所以只涉及共享内存有关的内容。当我们使用 CreateFile 创建或者打开一个文件时，系统会创建一个文件对象，如果该文件再次被打开，会有一个新的文件对象被创建，打开 N 次，生成 N 个文件对象。在每个文件对象中，有一个指针指向一个叫做“内存区对象指针”的结构。

“内存区对象指针”由三个 32 位的指针组成：指向共享的高速缓存映射的指针、指向数据控制区域的指针和指向映像控制区域的指针。指向共享的高速缓存映射的指针用于文件的缓存管理，是一个很重要的概念，以后的文章会有详细的描述，而后两者都指向控制区域结构，分别用来映射数据 (Data) 文件和可执行 (Image) 文件。

## 2. CreateFileMapping 的工作

前面提到，CreateFileMapping 函数会创建文件映射对象 (File Mapping Object)，该对象还有一个别名，叫做内存区对象 (Section Object)。内存区对象有两种：一种是需要具体文件支撑的，这个文件可以是可执行 (Image) 文件 (Image)，也可以是数据 (Data) 文件，通常这种内存区对象用于文件的映射操作，因此可以说这种内存区对象背靠的是映射文件。还有一种内存区对象是不需要具体文件，而使用页面文件支撑的，这种内存区对象通常用于两个或者多个进程共享内存，进行进程间通信，所以可以说这种内存区对象背靠的是页面文件。

注意，所谓的页面文件是操作系统创立的文件，专门用于将使用次数少的内存进行存储，从而空出内存，所以可以认为页面文件是内存的一个背靠文件。

CreateFileMapping 函数通过间接调用 MmCreateSection 函数完成内存区对象的创建。MmCreateSection 函数实现的逻辑如下：

1) 如果 FileHandle 非空，代表该内存区对象由映射文件支撑，那么新建一个 Control\_Area 对象，并用文件对象中的信息填充，利用 MiCreateImageFileMap 或者 MiCreateDataFileMap 创建一个新的 segment 对象；

2) 如果 FileHandle 为空，代表该内存区对象由页面文件支撑，则调用 MiCreatePagingFileMap 创建一个控制区对象和 segment 对象，而并不涉及到文件对象；

3) 利用 ObCreateObject 函数创建 section 对象，并填充相应信息，返回该 section 对象。

好了，一会功夫，蹦出三个对象：控制区域 (Control\_Area) 对象，内存区 (Section) 对象和段 (Segment) 对象，对象多了好头疼！

内存区对象：内存区对象属于内核对象之一，有标准的对象头，也有对象体，对象头由对象管理器负责，而对对象体的内容由内存管理器管理，其对象体结构如下：

```
typedef struct _SECTION {
    MMADDRESS_NODE Address;
```



```

//当这个 section 是可执行程序时，放于专门存放可执行程序的 VAD 树中
PSEGMENT Segment;           //指向段对象
LARGE_INTEGER SizeOfSection; // 内存区的大小
union {
    ULONG LongFlags;
    MMSECTION_FLAGS Flags;    //内存区的一组标志
} u;
MM_PROTECTION_MASK InitialPageProtection; //页面保护模式
} SECTION, *PSECTION;

```

可以看到，内存区对象包含了描述的内存区的大小，一些标志和页面的保护模式，而最重要的内存内容则隐藏在内存区对象 Segment 成员指向的段对象中。

控制区域对象的结构体如下：

```

typedef struct _CONTROL_AREA {
    PSEGMENT Segment;           //指向段对象
    .....
    ULONG NumberOfMappedViews; //反映了与之关联的内存区对象被映射了多少次
    .....
    PFILE_OBJECT FilePointer;   //指向文件对象
    PEVENT_COUNTER WaitingForDeletion;
    USHORT ModifiedWriteCount;
    USHORT FlushInProgressCount;
    ULONG WritableUserReferences;
    ULONG QuadwordPad;
} CONTROL_AREA, *PCONTROL_AREA;

```

这个数据结构是 Control\_Area 对象结构的主体，一个完整的 Control\_Area 对象之后还紧跟着 N 个 SUBSECTION 结构，每个 SUBSECTION 对应着文件中的一个 SECTION，用于描述文件中每节映射信息（只读、读写、写时复制等）。例如，我们知道一个 PE 文件有 N 个节（section，这个 section 和前面提到的不一样，它是 PE 文件的组成部分，可以翻译为“节”），那么 PE 文件中有几个节，在 Control\_Area 对象中就有多少个 Subsection。所有的 SUBSECTION 结构构成一个单链表，每个 SUBSECTION 结构有一个指针指回到 Control\_Area 对象结构。

段对象的结构体如下：

```

typedef struct _SEGMENT {
    struct _CONTROL_AREA *ControlArea;
    ULONG TotalNumberOfPtes;
    ULONG NonExtendedPtes;

    UINT64 SizeOfSegment;
    MMPTE SegmentPteTemplate;
    .....
    SEGMENT_FLAGS SegmentFlags;

```

```

PVOID BasedAddress;
.....
union {
PSECTION_IMAGE_INFORMATION ImageInformation;
PVOID FirstMappedVa;
} u2;
PMMPTYPE PrototypePte;
MMPTE ThePtes[MM_PROTO_PTE_ALIGNMENT / PAGE_SIZE];
} SEGMENT, *PSEGMENT;
    
```

Segment 段对象在分页缓冲池中分配，用来描述和存放内存区数据。大家会注意到该结构中有很多 PTE 的字样，一个完整的 Segment 段对象除了上述的结构，紧接着还会有一个 PTE 数组，形成一个原型 PTE 阵列，用于完成将内存区对象实际映射到物理内存上。

列出以上三个对象的数据结构，我们先看看它们之间的连接关系，如图 2 所示。

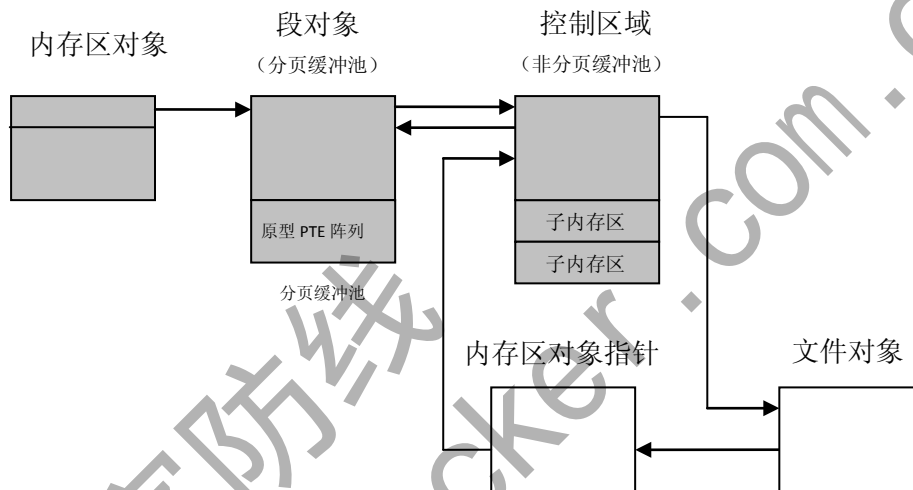


图 2 CreateFileMapping 函数所做的工作

当控制区域对象被创建后，如果该内存区对象为可执行文件，则内存区对象指针中的映像控制指针指向该控制区域对象，如果该文件为数据文件，则内存区对象指针中的数据控制指针指向该控制区域对象。

### 3. MapViewOfFile 的工作

好的，现在对象以及各个对象之间的关系都已经创建了，应用程序获得了内存区对象的句柄，但还不能访问内存区对象中的数据。为了使用内存区对象中的数据，应用程序必须映射一个视图，将内存区对象描述的地址映射到进程的地址空间，这个步骤由 Windows 内核的系统服务例程 NtMapViewOfSection 函数完成，对应于内存管理器中的函数是 MmMapViewOfSection 函数。

MmMapViewOfSection 函数的大致逻辑是这样的：由内存区对象→段对象→控制区对象中的标志信息确定内存区的类型：

- ① 若 PhysicalMemory 位为 1，则映射的类型为物理内存，使用 MiMapViewOfSpecialSection 函数来映射内存区；
- ② 若 Image 位为 1，表明内存区对象是个镜像文件，则调用 MiMapViewOfImageSection 函数来映射内存区；

③若非以上两种情况，那么内存区对象为数据文件或者页面文件，则调用 MiMapViewOfDataSection 函数来映射。

下面我们以 MiMapViewOfDataSection 函数的逻辑为例，该函数大体完成两个任务：①在进程地址空间中找到内存区对象声明大小的空闲地址范围，建立一个与该地址范围对应的 VAD 对象之后，将该 VAD 对象中的 ControlArea 指向在 CreateSection 函数中创建的控制区对象，并将该 VAD 节点插入到进程的 VAD 树中；②针对该地址范围，设置相应页表中的内容为段对象中的 SegmentPteTemplate 值。如果该内存区对象是使用页面文件支撑的，则仅设置保护属性。

在调用过 MiMapViewOfDataSection 函数后，系统就已经完成了从虚拟地址到物理内存再到文件的映射，应用程序就可以通过访问内存的方式来访问文件或者共享内存，其结构如图 3 所示。

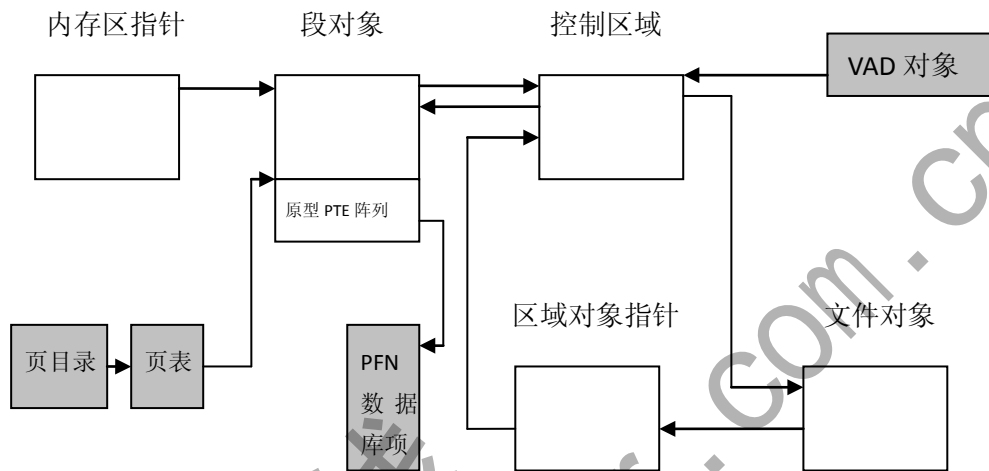


图 3 MapViewOfFile 函数所做的工作

#### 4. 原型 PTE 的使用

在前面的内容我们经常提到原型 PTE，下面我们看看原型 PTE 的庐山真面目！可以说原型 PTE 是实现共享内存最根本的机制。一个原型 PTE 可以描述 6 种状态的页面：

- ①有效。对应的页面位于物理内存中，此时原型 PTE 已经是一个有效的 PTE。
- ②位于页面文件中，对应的页面位于页面文件中。
- ③位于映射文件中，对应的页面位于映射文件中。

还有三种状态的页面分别是：要求零页面，转移页面，已修改但不写出页面。前面提到，在段对象中包含了内存区对应页面的原型 PTE 阵列，当进程访问该内存区对象中的页面时，内存管理器将页面对应的原型 PTE 中的内容填充到对应的页表 PTE 中，下面不妨用例子来说明。

阶段 1：

为了表达更清楚，我们将进程 A 中 P1 页面的 PTE 定义为 PTE1，进程 B 中 P1 页面的 PTE 定义为 PTE2。

假设进程 A 和进程 B 共享一个内存区对象，而该内存区包含一个页面 P1，目前该页面还没有被访问过，所以进程 A 和进程 B 中对应页面的 PTE 是无效的，并且都指向段对象中的 P1 的原型 PTE，而该原型 PTE 指向页面文件中的页面 P1，如图 4 所示。

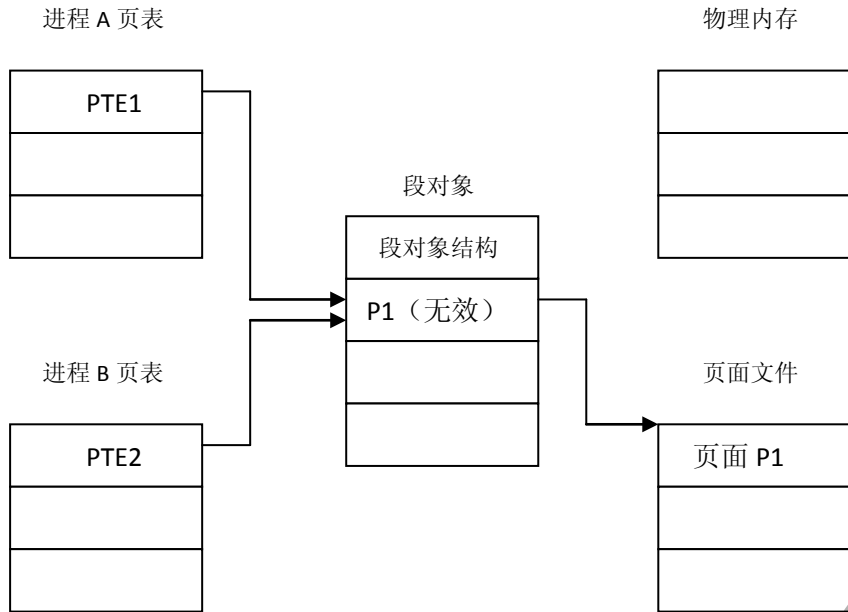


图 4 原型 PTE 使用的例子，阶段 1

当一个共享页面无效时，进程页表中的页表项由一个特殊的页表项来填充。这个特殊的页表项指向描述该页面的原型页表项，此时 PTE1 和 PTE2 中的格式如图 5 所示，其中有效位为 0 表示这是一个无效的 PTE，原型位为 1，表示这是一个指向原型 PTE 的 PTE，两段原型 PTE 地址组成 0~27 共 28 位的地址，因为每个 PTE 是四个字节，所以 28 位的地址可以用来描述 30 位的空间。因为段对象在系统换页池中分配，那么原型 PTE 都在换页内存池中，因此图 5 中原型 PTE 的地址指的是该原型 PTE 相对于系统换页内存池起始位置的偏移。

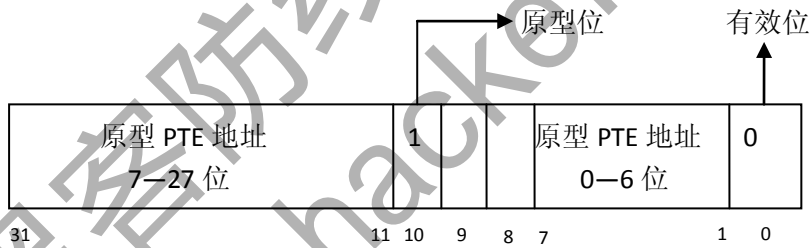


图 5 指向原型 PTE 的无效 PTE

为了将图 5 所示结构中的原型 PTE 地址转换为虚拟内存地址，需要将其进行转换，转换方法如图 6 所示。

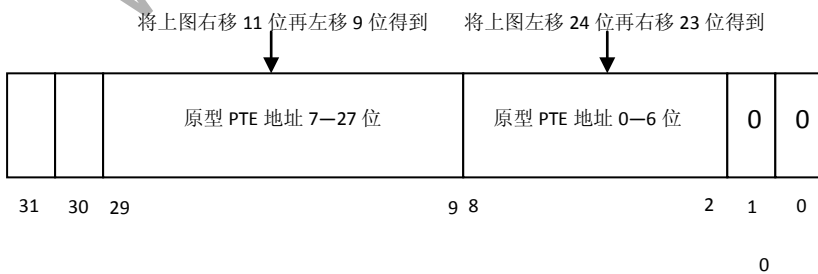


图 6 PTE 转换为虚拟地址（偏移）

因此，我们可以看到 WRK 中会有以下转换宏：

```
#define
MiPteToProto(lpte) (PMMPTe) ((PMMPTe) (((lpte)->u.loong)>>11<<9)+\(((lpte)->u.lo
ng))<<24)>>23)+MmProtopte_Base))
```

其中 MmProtopte\_Base 定义为:

```
#define MmProtopte_Base ((ULONG)MmPagedPoolStart)
```

实际上也就是换页内存池的开始位置。

阶段 2:

当进程 A 访问该页面时, 发生访问违例, 系统将页面 P1 倒入内存, 并将其页面号赋予段对象中的指向 P1 的原型 PTE, 同时将此原型 PTE 赋予 PTE1, 如图 7 所示; 此时 PTE1 和段对象中的原型 PTE 的内容是一致的, 都是有效的 PTE, 而 PTE2 则仍然是一个如图 5 所示的结构。

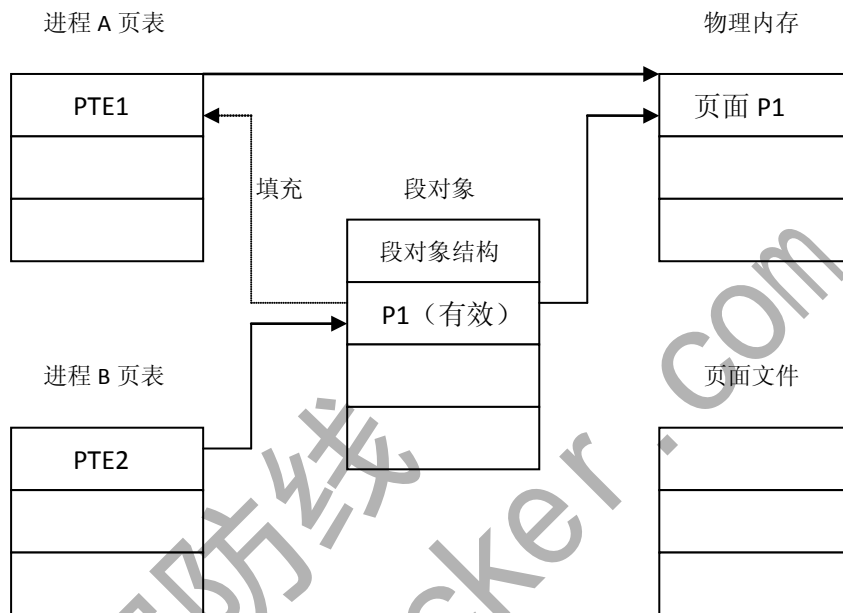


图 7 原型 PTE 使用的例子, 阶段 2

阶段 3:

当进程 B 访问此页面时, 依然会发生访问违例, 发现指向此页面的 PTE 是个原型 PTE (由原型位为 1, valid 位为 0), 并且页面已经导入内存, 那么就直接将段对象中的对应的原型 PTE 赋予 PTE2, 从而快速的实现页面共享。为了跟踪每个共享页面的使用情况, 在物理页面对应的帧号数据库中记录了该页面被几个进程共享, 当一个共享页面已经不再被任何页表引用, 内存管理器会将这个页面标记为无效, 并将其移到转换链表或写回外存。如图 8 所示。

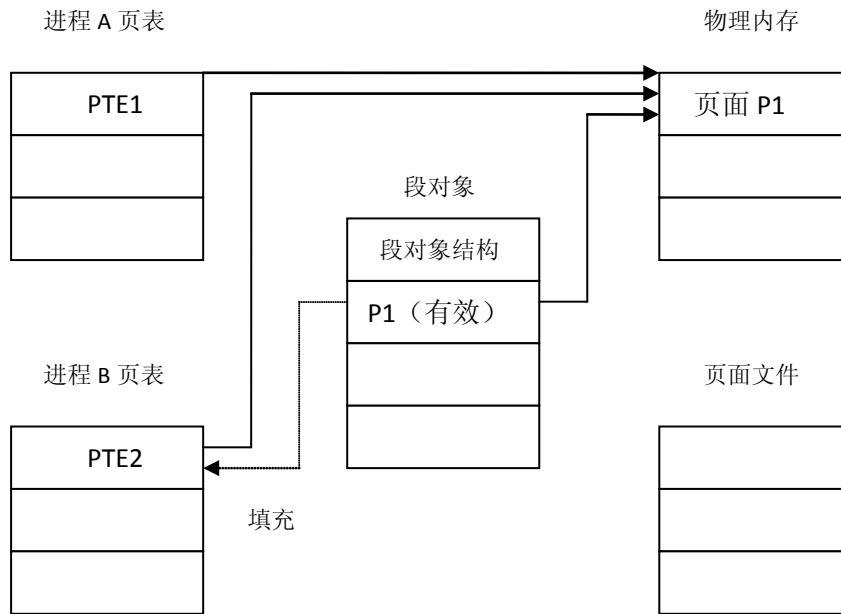


图 8 原型 PTE 使用的例子，阶段 3

另外如在图 7 的状态下，内存区对象中的一个页面从有效变成无效时，它的硬件 PTE 将直接指向原型 PTE，恢复图 4 中的状态。

### 小结

共享内存不仅应用于进程间内存共享，也用于将文件映射到进程的地址空间，从而实现文件的快速访问。本文讲解了共享内存的实现机制，重点是要理解 CreateFileMapping 函数和 MapViewOfFile 函数所做的工作以及原型 PTE 的原理。

(完)

# Android 木马揭秘之用户定位技术的实现

文/图 爱无言

这几年安卓系统的普及速度可谓迅猛，一时间各式各样的设备都承载着安卓系统，手机、平板、机顶盒等都忠实地成为了安卓系统的用户。由于安卓系统在移动设备上的使用率最高，而移动设备上存储的数据往往涉及到个人隐私，如手机通讯录、短信内容、拍摄照片、阅读书目、保存文档等，有时更会涉及到经济利益，这诱惑着一些利益集团开始制作基于安卓系统的远程控制程序，即安卓系统木马。首个安卓系统木马应属 2010 年出现的“Trojan-SMS.AndroidOS.FakePlayer.a”，这是一个以扣取用户手机话费为目的的盈利性安卓系统木马。随着需要的发展，单纯的盈利性木马已经不是重点，用户的隐私数据才是核心，尤其是具有用户行为监视性的木马最受关注。所谓“用户行为监视性的木马”就是指该类安卓木马能够监视用户的所在、所说、所做。“所在”即用户所处位置在哪里；“所说”即通话内容，聊天内容；“所做”即在操作什么程序，在干什么活动。这一类的木马由于涉及到用户核心利益，往往经济价值较大，多用于私人侦探、商业窃密等领域，平时很难见到，更不要说了解其核心代码、实现机制。为此，本文将逐步向读者揭秘这些高级安卓木马的核心实现技术，帮助大家更好地了解这些木马实现技术，从而做好对个人隐私的保护，防范该类木马的入侵。本文旨在讨论技术，凡利用本文技术进行违法活动的作者与杂志概不负责。

这里首先要向大家揭秘的是高级安卓木马是如何实现对用户定位的，即如何知道用户的“所在”。如果你利用百度搜索安卓定位原理，会发现百度给出的解释不外乎是利用 GPS 或者手机基站定位，甚至结合 Wi-Fi 信号。原理不错，但这只是原理，要想具体实现定位可是有一定难度的。以手机基站定位为例，现在传统的实现方式是利用 Android SDK 中的 API（TelephonyManager）获得 MCC、MNC、LAC、CID 等信息，然后通过 Google 的 API 获得所在位置的经纬度，最后再通过 Google Map 的 API 获得实际的地理位置。这其中，MCC 即 Mobile Country Code，移动国家代码（中国的为 460）；MNC，Mobile Network Code，移动网络号码（中国移动为 00，中国联通为 01）；LAC，Location Area Code，位置区域码；CID，Cell Identity，基站编号，是个 16 位的数据（范围是 0 到 65535）。由于谷歌存储了 MCC、MNC、LAC、CID 等信息，一旦我们能够获取当前移动设备所在基站的这些数据，就可以通过向谷歌的“<http://www.google.com/loc/json>”网址发送查询数据获取基站所在经纬度。得到经纬度后，我们将其转换为实际地址，这需要向谷歌的“<http://maps.google.cn/maps/geo?key=abcdefg&q=>”发送经纬度数据，最终获得移动设备所在实际地址。这样的实现代码在网上很多，你会发现它们都不好使了，为什么呢？因为“<http://www.google.com/loc/json>”这个网址现在已经不能访问了。这个可悲的消息使得我们意识到必须采用一种相对稳妥的方法来实现移动设备定位。在对某个安卓木马程序做逆向分析时，发现一种新的基于手机基站定位实现技术。当然在这之前，细心的读者会发现为什么我们一直在详细讲解基于手机基站的定位实现，而不采用最为常用的 GPS。因为手机这样的移动设备一旦进入到房屋内等封闭场所，GPS 信号就衰减为 0，不足以实现定位，而手机信号多半都是存在的，所以基于手机基



站的定位方式更为稳妥，这就是为什么很多高级安卓木马会采用该方式实现定位的原因。言归正传，我们发现的这个安卓木马采用了基于百度提供的定位 SDK。根据百度官方的解释：百度 Android 定位 SDK 支持 Android1.5 及以上设备，提供定位功能，通过 GPS、网络定位（WIFI、基站）混合定位模式，返回当前所处的位置信息；反地理编码功能：解析当前所处的位置坐标，获得详细的地址描述信息。如此丰富的技术支持，难怪该安卓木马会采用这个 SDK。

百度 Android 定位 SDK 的使用非常简单，首先在百度的官网下载最新的库文件，将 liblocSDK.so 文件拷贝到 libs/armeabi 目录下，将 locSDK.jar 文件拷贝到工程根目录下，并在工程属性->Java Build Path->Libraries 中选择 Add JARs，选定 locSDK.jar，确定后返回，就可以在程序中使用百度 Android 定位 SDK 了。在代码实现时，首先需要初始化 LocationClient 类，其代码如下：

```
public LocationClient mLocationClient = null;
public BDLocationListener myListener = new MyLocationListener();
public void onCreate() {
    mLocationClient = new LocationClient(this);    //声明 LocationClient 类
    mLocationClient.registerLocationListener( myListener );    //注册监听函数
}
```

接着实现 BDLocationListener 接口。BDLocationListener 接口有一个方法，作用是接收异步返回的定位结果，参数是 BDLocation 类型参数。其代码如下：

```
public class MyLocationListener implements BDLocationListener {
    @Override
    public void onReceiveLocation(BDLocation location) {
        if (location == null)
            return ;
        StringBuffer sb = new StringBuffer(256);
        sb.append("time : ");
        sb.append(location.getTime());
        sb.append("\nerror code : ");
        sb.append(location.getLocType());
        sb.append("\nlatitude : ");
        sb.append(location.getLatitude());
        sb.append("\nlontitude : ");
        sb.append(location.getLongitude());
        sb.append("\nradius : ");
        sb.append(location.getRadius());
    }
}
```

```
        if (location.getLocType() == BDLocation.TypeGpsLocation){
            sb.append("\nspeed : ");
            sb.append(location.getSpeed());
            sb.append("\nsatellite : ");
            sb.append(location.getSatelliteNumber());
        } else if (location.getLocType() == BDLocation.TypeNetWorkLocation){
            sb.append("\naddr : ");
            sb.append(location.getAddrStr());
        }
        logMsg(sb.toString());
    }
}
```

接着设置参数。设置定位参数包括定位模式（单次定位，定时定位），返回坐标类型，是否打开 GPS 等。实现代码如下：

```
LocationClientOption option = new LocationClientOption();
option.setOpenGps(true);
option.setAddrType("detail");
option.setCoorType("gcj02");
option.setScanSpan(5000);
mLocClient.setLocOption(option);
```

最后，发起定位请求。请求过程是异步的，定位结果在上面的监听函数中获取，代码如下：

```
if (mLocClient != null && mLocClient.isStarted())
    mLocClient.requestLocation();
else
    Log.d("LocSDK_2.0_Demo1", "locClient is null or not started");
```

实际测试效果如图 1 所示。从图中可以看出，演示程序准确定位到了我此刻手机所在的位置，定位精度在百米内。木马程序一旦使用了这样的技术，完全可以实现对用户所在的监视，你此刻是不是有一种毛骨悚然的感觉呢？

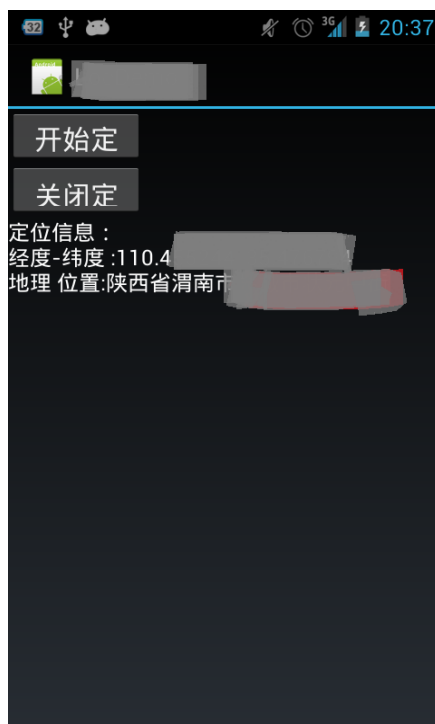


图 1

定位代码实现后，就可以利用移动网络将用户数据时时上传至控制端网站，由控制端用户查看。结合以往数据，就可以勾画出一个人在一段时间内的行踪，对被控制者来说，此刻的手机或者其它移动设备就成为了出卖自己的第一元凶。

定位代码的成功实现，只是高级安卓木马程序的一部分功能，随后的工作还有很多，我会陆续将这些核心技术向读者一一揭秘。

(完)

# 2013 年第 4 期杂志特约选题征稿

黑客防线于 2013 年推出新的约稿机制，每期均会推出编辑部特选的选题，涵盖信息安全领域的各个方面。对这些选题有兴趣的读者与作者，可联系信箱：[du\\_xing\\_zhe@yahoo.com.cn](mailto:du_xing_zhe@yahoo.com.cn)，或者 QQ: 675122680，确定有意的选题。按照要求如期完成稿件者，稿酬从优！第 3 期的选题如下：

## 1) 编写下载者

说明：

编写一个下载者程序，每次开机启动后，都能从指定网站获取数据，下载并执行。

要求：

- 1) 将该程序上传到邮箱，然后下载到机器上运行，不会有 SmartScreen 提示。
- 2) Windows UAC 安全设置最高，程序自身运行时无提示，能获取系统管理员权限。
- 3) 执行其他程序也能无提示获取系统管理员权限。
- 4) 能绕过 360 安全卫士监控，加载驱动保护自身，隐藏和保护指定的文件，隐藏连接，使用 XueTr、PowerTool 工具无法查看到文件和连接。
- 5) 免杀过 360 安全卫士、360 杀毒等主流杀软。
- 6) 程序没有签名。
- 7) 运行的系统补丁打到最新。
- 8) 支持操作系统 Windows xp/2003/ista/7/2008/8。
- 9) 以上所有功能，能在 32 位和 64 位系统上通用稳定。

## 2) 自动下载邮箱内容

说明：

支持 IE 系列、Firefox、Chrome 等常用浏览器，支持 Gmail、Hotmail、Yahoo 三个邮箱。需要实现的功能是当用户登录上述邮箱时，程序自动将用户邮箱中的邮件自动保存到本地指定目录，要求保存为 eml（包括附件内容）。

要求：

初步要求将当前用户查看的邮件内容及附件保存为 eml 到本地；进一步的要求是将当前用户的所有邮件保存为 eml 到本地。

## 3) Ring3 下实现 PE Loader

要求：

- 1) 支持 C、C++ 的 Run time 运行库以及 tls；支持 mfc；支持 .net 运行库；
- 2) 支持 Windows NT5/NT6 32 位以及 64 位的操作系统；
- 3) 使用 VC 实现，不要封装成类的形式；
- 4) 实现成功的标志：成功将测试 DLL 程序加载入进程的内存空间，并且进程模块中不显示被加载的 DLL 模块。

## 4) 邮箱自动取信程序

说明：

编写一个程序，当用户使用浏览器登录邮箱时，自动把信箱里的所有邮件下载到本地。

要求:

- 1) 支持 Gmail、hotmail、yahoo 新版旧版。
- 2) 支持 IE 浏览器 6/7/8/9/10, 或支持火狐浏览器, 或谷歌浏览器。
- 3) 获取收件箱、发件箱与通讯录内容。

建议编写成 BHO 插件或其他浏览器插件, 分析邮箱网页上的元素, 将邮件内容、主题、发件人等相关信息分析出来。

### 5) 邮箱附件劫持

说明:

编写一个程序, 当用户在浏览器上登录邮箱 (本地权限), 发送邮件时, 自动将附件里的文件替换为另外一个文件。

要求:

- 1) 支持 Gmail、hotmail、yahoo 新版旧版、163、126。
- 2) 支持 IE 浏览器 6/7/8/9/10, 或支持火狐浏览器, 或谷歌浏览器。

### 6) 突破 Windows7 UAC

说明:

编写一个程序, 绕过 Windows7 UAC 提示, 启动另外一个程序, 并使这个程序获取到管理员权限。

要求:

- 1) Windows UAC 安全设置为最高级别;
- 2) 系统补丁打到最新;
- 3) 支持 32 位和 64 位系统。

### 7) Android Wifi Tether 数据转储

说明:

Wifi Tether (开源项目) 可以在 ROOT 过的 Android 设备上共享移动网络 (也就是我们常说的 Wi-Fi 热点), 请参照 Wifi Tether 实现一个程序, 对流经本机的所有网络数据进行分析存储。

要求:

- 1) 开启 Wifi 热点后, 对流经本机的所有网络数据进行存储;
- 2) 不同的网络协议存储为不同的文件, 比如 HTTP 协议存储为 HTTP.DAT。

### 8) Android 系统中暴力破解以 WEP 加密的 WIFI 密码

说明:

- 1) Android 系统为 root 过的;
- 2) 针对 WEP 加密方式进行破解;
- 3) 可以使用跑密码字典的方式实现;
- 4) 可以编写新程序或者移植。

要求:

- 1) 代码必须以后台方式运行;
- 2) 代码从 sd 卡 `wifi.txt` 文件读取要破解的目标;
- 3) 代码从 sd 卡 `pass.txt` 文件读取要尝试的密码;
- 4) 破解结果写入 sd 卡 `password.txt` 文件。

黑客防线  
www.hacker.com.cn

## 2013 征稿启示

《黑客防线》作为一本技术月刊，已经 13 年了。这十多年以来基本上形成了一个网络安全技术坎坷发展的主线，陪伴着无数热爱技术、钻研技术、热衷网络安全技术创新的同仁们实现了诸多技术突破。再次感谢所有的读者和作者，希望这份技术杂志可以永远陪你一起走下去。

投稿栏目：

### 首发漏洞

要求原创必须首发，杜绝一切二手资料。主要内容集中在各种 0Day 公布、讨论，欢迎第一手溢出类文章，特别欢迎主流操作系统和网络设备的底层 0Day，稿费从优，可以洽谈深度合作。有深度合作意向者，直接联系总编辑 binsun20000@hotmail.com

### Android 技术研究

黑防重点栏目，对 android 系统的攻击、破解、控制等技术的研究。研究方向包括 android 源代码解析、android 虚拟机，重点欢迎针对 android 下杀毒软件机制和系统底层机理研究的技术和成果。

### 本月焦点

针对时下的热点网络安全技术问题展开讨论，或发表自己的技术观点、研究成果，或针对某一技术事件做分析、评测。

### 漏洞攻防

利用系统漏洞、网络协议漏洞进行的渗透、入侵、反渗透，反入侵，包括比较流行的第三方软件和网络设备 0Day 的触发机理，对于国际国内发布的 poc 进行分析研究，编写并提供优化的 exploit 的思路和过程；同时可针对最新爆发的漏洞进行底层触发、shellcode 分析以及对各种平台的安全机制的研究。

### 脚本攻防

利用脚本系统漏洞进行的注入、提权、渗透；国内外使用率高的脚本系统的 0Day 以及相关防护代码。重点欢迎利用脚本语言缺陷和数据库漏洞配合的注入以及补丁建议；重点欢迎 PHP、JSP 以及 html 边界注入的研究和代码实现。

### 工具与免杀

巧妙的免杀技术讨论；针对最新 Anti 杀毒软件、HIPS 等安全防护软件技术的讨论。特别欢迎突破安全防护软件主动防御的技术讨论，以及针对主流杀毒软件文件监控和扫描技术的新型思路对抗，并且欢迎在源代码基础上免杀和专杀的技术论证！最新工具，包括安全工具和黑客工具的新技术分析，以及新的使用技巧的实力讲解。

### 渗透与提权

黑防重点栏目。欢迎非 windows 系统、非 SQL 数据库以外的主流操作系统地渗透、提权技术讨论，特别欢迎内网渗透、摆渡、提权的技术突破。一切独特的渗透、提权实际例子均在此栏目发表，杜绝任何无亮点技术文章！

### 溢出研究

对各种系统包括应用软件漏洞的详细分析，以及底层触发、shellcode 编写、漏洞模式等。

### 外文精粹

选取国外优秀的网络安全技术文章，进行翻译、讨论。

### 网络安全顾问

我们关注局域网和广域网整体网络防/杀病毒、防渗透体系的建立；ARP 系统的整体防护；较有效的不损失网络资源的防范 DDos 攻击技术等相关方面的技术文章。

### 搜索引擎优化

主要针对特定关键词在各搜索引擎的综合排名、针对主流搜索引擎的多关键词排名的优化技术。

### 密界寻踪

关于算法、完全破解、硬件级加解密的技术讨论和病毒分析、虚拟机设计、外壳开发、调试及逆向分析技术的深入研究。

### 编程解析

各种安全软件和黑客软件的编程技术探讨；底层驱动、网络协议、进程的加载与控制技术探讨和 virus 高级应用技术编写；以及漏洞利用的关键代码解析和测试。重点欢迎 C/C++/ASM 自主开发独特工具的开源讨论。

### 投稿格式要求：

1) 技术分析来稿一律使用 Word 编排，将图片插入文章中适当的位置，并明确标注“图 1”、“图 2”；

2) 在稿件末尾请注明您的账户名、银行账号、以及开户地，包括你的真实姓名、准确的邮寄地址和邮编、QQ 或者 MSN、邮箱、常用的笔名等，方便我们发放稿费。

3) 投稿方式和和周期：

采用 E-Mail 方式投稿，投稿 mail: du\_xing\_zhe@yahoo.com.cn QQ675122680 投稿后，稿件录用情况将于 1-3 个工作日内回复，请作者留意查看。每月 10 日前投稿将有机会发表在下月杂志上，10 日后将放到下下月杂志，请作者朋友注意，确认在下一期也没使用者，可以另投他处。限于人力，未采用的恕不退稿，请自留底稿。

**重点提示：**严禁一稿两投。无论什么原因，如果出现重稿——与别的杂志重复——与别的网站重复，将会扣发稿费，从此不再录用该作者稿件。

4) 稿费发放周期：

稿费当月发放，稿费从优。欢迎更多的专业技术人员加入到这个行列。

5) 根据稿件质量，分为一等、二等、三等稿件，稿费标准如下：

一等稿件 900 元/篇

二等稿件 600 元/篇

三等稿件 300 元/篇

6) 稿费发放办法：

银行卡发放，支持境内各大银行借记卡，不支持信用卡。

7) 投稿信箱及编辑联系

投稿信箱 du\_xing\_zhe@yahoo.com.cn

编辑 QQ 675122680