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。

第五章 测试及分析

第六章 总结及展望