栈溢出

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏洞，类似的还有堆溢出，bss段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。此外，我们也不难发现，发生栈溢出的基本前提是

- 程序必须向栈上写入数据。
- 写入的数据大小没有被良好地控制。

溢出方式

- **覆盖函数返回地址**，这时候就是直接看 EBP 即可。
- **覆盖栈上某个变量的内容**。
- **覆盖 bss 段某个变量的内容**。
- 根据现实执行情况，覆盖特定的变量或地址的内容。

缓解机制

Stack Canary

汇编代码(x64)：

Assembly language

```
...
mov    rax,QWORD PTR fs:0x28
mov    QWORD PTR [rbp-0x8],rax
...
mov    rax,QWORD PTR [rbp-0x8]
xor    rax,QWORD PTR fs:0x28
je     0x4005f8 <main+66>
call   0x400480<__stack_chk_fail@plt>
...
```

`linux`中，`fs`寄存器用于存放TLS(Thread Local Storage)，`fs:0x28`即`canary`，存储在TLS结构体中。

汇编代码(x86)：

Assembly language

```
...
mov    eax,gs:0x14
mov    DWORD PTR [ebp-0xc],eax
...
mov    eax,DWORD PTR [ebp-0xc]
xor    eax, DWORD PTR gs:0x14
je     0x80484cd <main+66>
call   0x804835 <__stack_chk_fail@plt>
...
```

`gs:0x14` 即 `canary`。

无论是 `canary` 还是与之 `XOR` 的控制数据被篡改，都会转到 `__stack_chk_fail` 函数，抛出 `stack smashing detected` 的错误。因此，攻击 `canary` 的方法主要有两种，一是，泄露 `canary`，使其在栈溢出时不会变化；二是，同时修改 `canary` 和 `TLS`。

off-by-one

单字节溢出

C

```c
//vuln.c
#include <stdio.h>
#include <string.h>

void foo(char* arg);
void bar(char* arg);

void foo(char* arg) {
 bar(arg); /* [1] */
}

void bar(char* arg) {
 char buf[256];
 strcpy(buf, arg); /* [2] */
}

int main(int argc, char *argv[]) {
 if(strlen(argv[1])>256) { /* [3] */
  printf("Attempted Buffer Overflow\n");
  fflush(stdout);
  return -1;
 }
 foo(argv[1]); /* [4] */
 return 0;
}
//gcc -fno-stack-protector -z execstack -no-pie -m32 -o vuln vuln.c
```

首先看一下strcpy和strlen函数原型

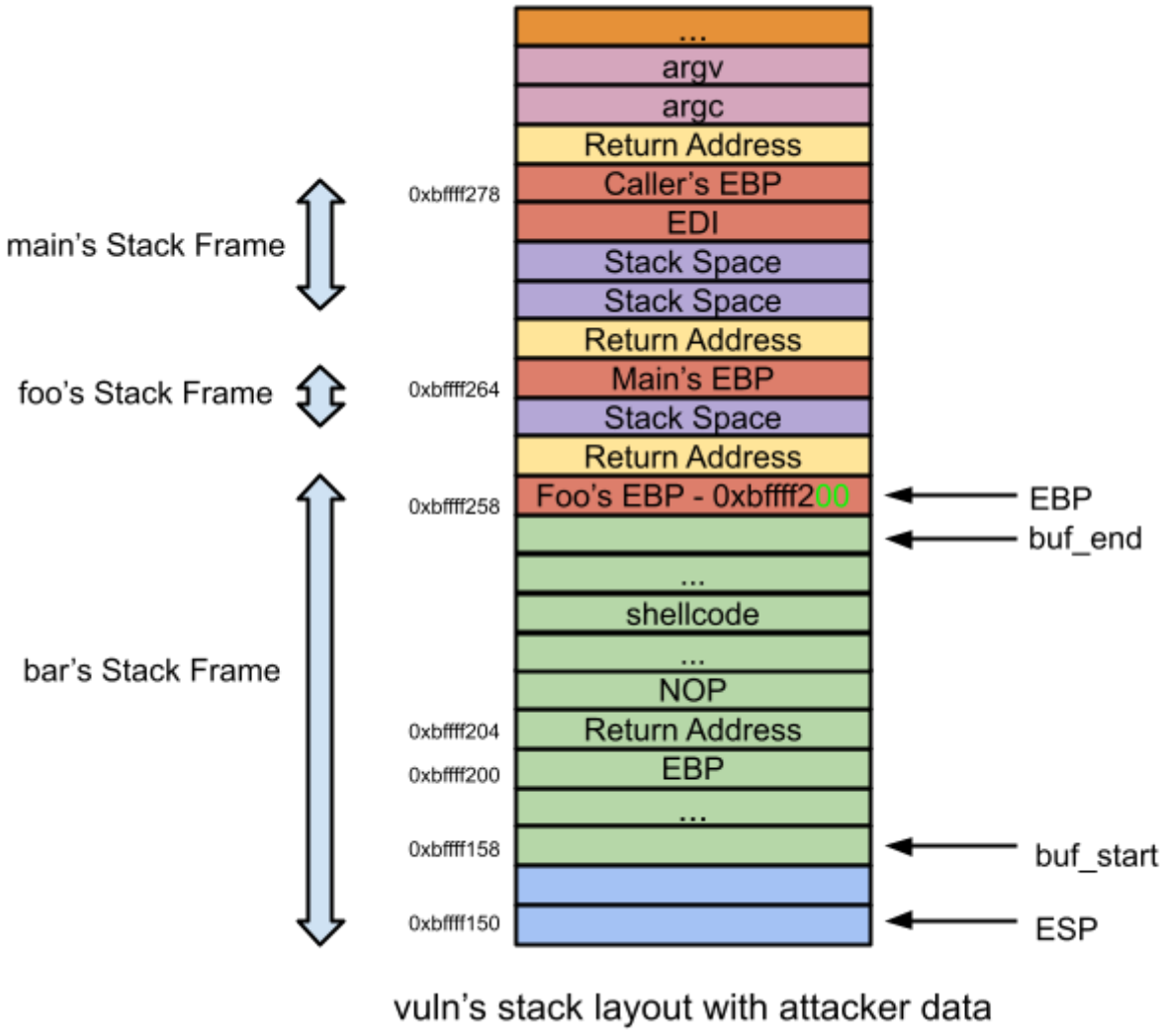
C

```c
strcpy(char *to, const char *from)
{
        char *save = to;

        for (; (*to = *from) != '\0'; ++from, ++to);    //'\0'也被复制
        return(save);
}
```

C

```c
strlen(const char *str)
{
        const char *s;

        for (s = str; *s; ++s);     //不计算字符串末尾的'\0'
        return (s - str);
}
```

那么，`argv[1]`可以在256个字符的基础上，溢出一个`'\0'`字符，即\x00。



vuln's stack layout with attacker data

从堆栈布局可以看到，bar函数的栈帧中，ebp原本是`0xbffff258`，由于`buf`溢出一个字节变成了`0xffff200`。

bar函数结束时：`mov ebp, esp;` $esp = 0xbffff264

`pop ebp;` $ebp = 0xbffff200

foo函数结束时： `mov ebp, esp;` $esp = 0xbffff200

`pop ebp;` $ebp = XXX

此时通过构造`esp(0xbffff200)` 之后的内存空间，我们可以返回到任何要实现任意代码执行的位置。