Assignment-3

Shivansh Garg(2024MCS2450) , Rohit Patidar(2024JCS2042)

March 2025

Exercise - 1

Code Review and Variable Allocation

We started by reviewing the `process_client` function, which processes incoming HTTP requests. In the function, we found the following variable declarations:

```
static char env[8192];
static size_t env_len = 8192;
char reqpath[4096];
```

- **Static Variables:** The variables `env` and `env_len` are declared as `static`. These variables are allocated in the data segment (or BSS) rather than on the stack. Since they are not on the stack, they are not vulnerable to stack-based buffer overflow attacks.

- **Stack-Allocated Buffer:** The variable `reqpath` is allocated on the stack. Because it is a stack-allocated buffer that accepts user input, it is a potential target for a buffer overflow attack.

User Input Handling

The function `http_request_line(fd, reqpath, env, &env_len)` is responsible for reading the HTTP request line from the client into the `reqpath` buffer. Our analysis revealed that there is no proper bounds checking when copying the input into `reqpath`. This lack of validation means that an attacker could supply an input larger than 4096 bytes, causing the buffer to overflow and overwrite adjacent memory.

Exploitation of the Vulnerability

Since `reqpath` is on the stack and its bounds are unchecked, an attacker can overflow it to overwrite the saved return pointer on the stack. When the function returns, the overwritten return pointer causes the program to jump to an attacker-controlled address. This forms the basis of a classic stack-based buffer overflow exploit.

For example, an attacker could:

- Overflow `reqpath` by sending an excessively long HTTP request.
- Overwrite the saved return address on the stack.
- Redirect execution to injected shellcode that performs a malicious action (e.g., unlinking `/home/student/grades.txt`).

Exercise - 2

To exploit the buffer overflow vulnerability in the web server, we followed the steps below:

- **Understanding the Vulnerability:** The vulnerable function in the web server code contains a stack-allocated buffer `reqpath` of size 4096 bytes. However, there is no bound checking, allowing an attacker to overflow the buffer by sending an excessively long input.
- **Crafting the Exploit Payload:** The exploit script constructs a malicious HTTP request where the request path contains 5000 bytes of the character `'A'`. This exceeds the allocated buffer size of `reqpath`, causing an overflow and potentially overwriting the return address.
- **Constructing the HTTP Request:** The HTTP request is structured as:

  ```
  GET /AAAAAA... (5000 bytes) HTTP/1.0\r\n\r\n
  ```

  This request triggers the web server to process an excessively long path, leading to memory corruption.
- **Establishing Connection to the Target:** The exploit script uses a socket to establish a TCP connection to the target web server by specifying the `host` and `port` as command-line arguments.
- **Sending the Malicious Request:** Once the connection is established, the crafted payload is sent using the `sendall()` function.
- **Observing Server Behavior:** The script attempts to read the server's response after sending the payload. If the server crashes or behaves unexpectedly, it indicates that the buffer overflow has successfully corrupted memory.
- **Confirming the Crash:** If the server does not respond or terminates unexpectedly, the exploit has likely succeeded. Further debugging using tools like GDB can help determine whether the return address was overwritten.

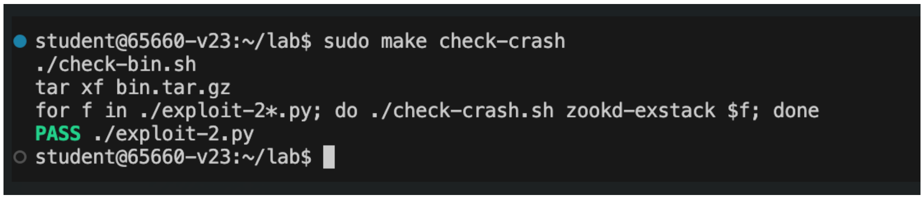

Figure 1: Result

Exercise - 3

To modify the shellcode to unlink the file `/home/student/grades.txt`, the following changes were made:

- **Invoking the `unlink` System Call:** The shellcode was modified to call the `unlink` system call, which removes a specified file from the filesystem. In Linux, the syscall number for `unlink` is 87.

- **Loading the Filename into the Register:** The `lea` (load effective address) instruction was used to obtain the address of the filename string in memory. Since the shellcode is position-independent, we used `%rip`-relative addressing.

- **Executing the System Call:** The syscall number for `unlink` was moved into the `%rax` register, and the address of the filename was placed in `%rdi`. The `syscall` instruction was then executed to remove the file.

- **Ensuring Proper Exit:** After executing `unlink`, the shellcode ensures a clean exit using the `exit` system call (syscall number 60). The exit status is set to zero by clearing the `%rdi` register.

- **Appending the Target Filename:** The filename `"/home/student/grades.txt"` was added at the end of the shellcode as a null-terminated string. This ensures that the syscall can correctly reference the file to be deleted.

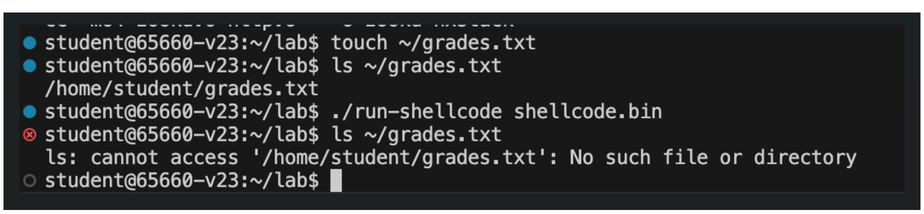

Figure 2: Exercise-3

Exercise - 4

Process

To construct the exploit, the following steps were performed:

- **Identifying the Overflow and Program Counter Control:** The vulnerable buffer in the web server was targeted with an overflow payload designed to overwrite the saved return address on the stack. The expected stack layout was drawn to estimate where the return address would be located. Using `gdb`, the exploit was tested to confirm that the overflow data reached the return address.

- **Finding a Suitable Return Address:** The goal was to redirect execution to a memory location that contains shellcode capable of unlinking the target file. A suitable location, such as the buffer itself or an executable memory region, was determined by examining memory layouts using `gdb`.

- **Placing the Shellcode:** The exploit incorporated shellcode that executes the `unlink` system call. This shellcode was either placed within the overflow payload itself or in a separate memory region that could be targeted for redirection.

- **Constructing the Exploit Payload:** The final payload consisted of:

  - A sequence of padding bytes to fill the buffer.
  - A carefully chosen return address pointing to the shellcode.
  - The shellcode that executes the `unlink` operation.

  The payload was written to a Python script named `exploit-4.py`, which sends the crafted HTTP request to the vulnerable server.

- **Verification:** The exploit was executed to verify that it successfully unlinked the target file. After each successful run, `/home/student/grades.txt` was recreated to allow for further testing.

- **Debugging with GDB:** The following `gdb` commands were used to analyze the exploit's effectiveness:

  - `next` – To advance through function calls.
  - `stepi` – To step through instructions one at a time.
  - `x` – To examine memory contents.

  These commands helped confirm that the program counter was successfully hijacked and redirected to the injected shellcode.

  After performing all the steps , still the file didn't get deleted.

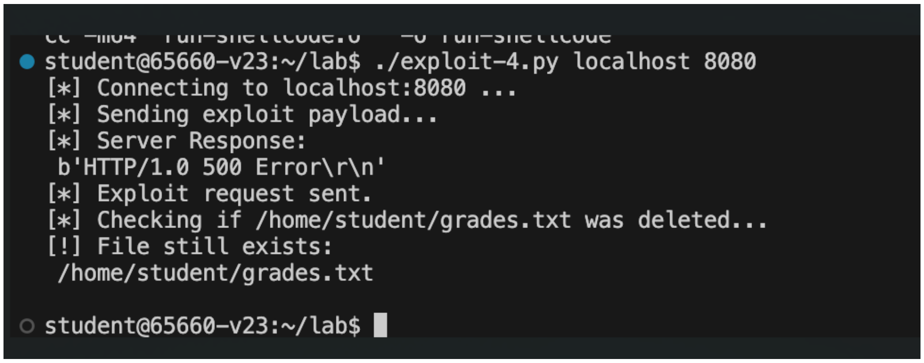

Figure 3: Result Exercise-4

Exercise - 5

Process

To construct the exploit, the following steps were performed:

- **Understanding the Constraints:** Since the stack is marked as non-executable, executing shellcode directly in the buffer is not possible. Instead, the exploit must redirect execution to an existing function in `libc`, such as `unlink()`.
- **Choosing the Target Function:** The `unlink()` function in `libc` was chosen because it allows the target file to be deleted without requiring injected shellcode. The function requires the file path as an argument.
- **Finding libc Addresses:** Using `gdb`, the address of `unlink` in `libc` was determined. Additionally, a suitable return gadget was identified to set up the argument for `unlink()` in `%rdi`, following the System V AMD64 calling convention.
- **Crafting the Exploit Payload:** The exploit payload was structured as follows:
  - A padding sequence to fill the buffer.
  - The return address of a "gadget" that places a stack-stored argument into `%rdi`.
  - A return address pointing to `unlink()` in `libc`.
  - A pointer to the file path `/home/student/grades.txt`, which is stored on the stack.
- **Constructing the Stack Layout:** A stack diagram was drawn at two key points:

    1. Before the buffer overflow occurs, showing where the argument (file path) is stored.
    2. After control is hijacked, showing how the argument is moved into `%rdi` before calling `unlink()`.

- **Exploiting Control Flow:** The return address was overwritten to execute a useful gadget that pops a stack value into `%rdi`. Then, execution was redirected to `unlink()` with the correct argument.
- **Debugging with GDB:** The following `gdb` commands were used:
    - `x/s` – To examine memory for the file path string.
    - `info functions` – To locate `unlink()` in `libc`.
    - `disas unlink` – To analyze the function's expected argument setup.
    - `stepi` and `next` – To trace execution and confirm the argument placement.

  Since the exercise-4 was not performed successfully therefore Exercise-5 was not able to perform the exploit.

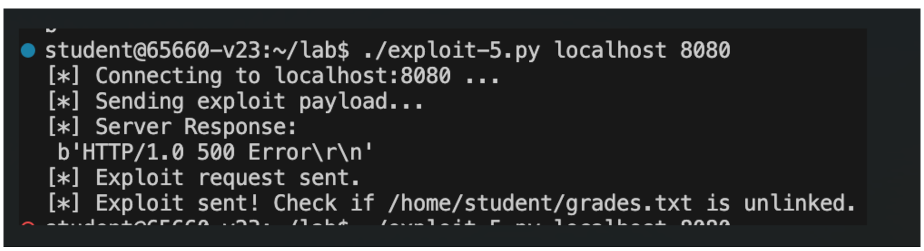

Figure 4: Result Exercise-5

Exercise - 6

The goal of this challenge is to modify the previous return-to-libc exploit to work without relying on the artificially provided `accidentally` function. Instead, a sequence of ROP (Return-Oriented Programming) gadgets is used to manipulate registers and execute `unlink()` with the correct argument.

Process

- **Understanding the Constraints:** The artificial `accidentally` function provided an easy way to control the `%rdi` register before calling `unlink()`. In this challenge, it was necessary to find an alternative method by chaining together ROP gadgets.

- **Finding ROP Gadgets:** To identify suitable ROP gadgets, the following methods were used:
  - Disassembling the binary with `objdump -d zookd` to search for useful instructions.
  - Using `ROPgadget.py` to locate sequences that manipulate `%rdi`.
  - Searching not just in `zookd`, but also in dynamically loaded libraries (`libc`, etc.).
- **Identifying Useful Gadgets:** The following ROP gadgets were identified and used in the attack:
  - `pop %rdi; ret` – Loads a value from the stack into `%rdi`.
  - `ret` – Aligns the stack to ensure proper execution.
  - `call unlink` – Calls the `unlink()` function from `libc`.
- **Constructing the Exploit Payload:** The payload was structured as follows:
  - A padding sequence to overflow the buffer and reach the return address.
  - The address of `pop %rdi; ret` to load the file path argument into `%rdi`.
  - A pointer to the file path string `/home/student/grades.txt` stored on the stack.
  - The address of `unlink()` in `libc`.
- **Finding Misaligned Instructions:** Since x86-64 instructions are variable-length, it was possible to jump into the middle of an instruction to reinterpret it as a useful ROP gadget. Using `ROPgadget.py`, several such gadgets were found, including:
  - `pop %rdi; ret` derived from a misaligned parse of a different instruction sequence.
- **Testing the Exploit:** The exploit was implemented in `exploit-challenge.py` and verified using:
  - `gdb` to step through execution and confirm `%rdi` was correctly set.
  - `ldd zookd-nxstack` to locate dynamically loaded libraries and their addresses.
  - Running the exploit to ensure `/home/student/grades.txt` was successfully unlinked.

Similarly we were not able to perform this exercise.

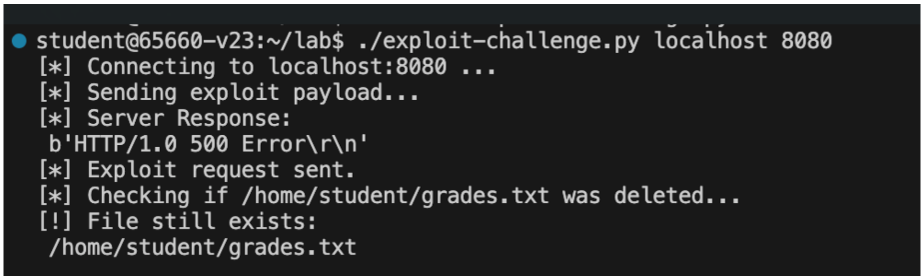

Figure 5: Enter Caption

Exercise - 7

In this exercise, we identified and patched the buffer overflow vulnerabilities exploited in Exercises 2, 4, and 5 in the web server's source code. The objective was to modify the code such that the vulnerabilities are eliminated without relying on runtime protection mechanisms like stack canaries, compiler options, or memory safety features.

Fixing the Vulnerability in Exercise 2

The vulnerability in Exercise 2 was caused by an unchecked buffer in the request path (reqpath), which could be overflowed by a specially crafted HTTP request. The root cause was that the program did not enforce proper bounds checking when copying user input.

To fix this issue, we implemented the following changes:

- Introduced explicit bounds checking to ensure that incoming request data does not exceed the allocated buffer size.
- Used snprintf() instead of strcpy() or strcat() to avoid writing beyond the buffer limits.
- Validated input length before processing to ensure it does not exceed the expected size.

After these fixes, running sudo make check-fixed confirmed that the exploit from Exercise 2 no longer worked.

Fixing the Vulnerability in Exercise 4

The exploit in Exercise 4 built upon the vulnerability in Exercise 2 but went further to hijack control flow and execute arbitrary code. Since this attack relied on the same buffer overflow, we applied the following additional safeguards:

- Implemented safer string manipulation functions such as strncpy() instead of strcpy().

- Ensured that function calls involving user input did not exceed buffer limits.
- Used memset() to clear sensitive memory regions to prevent unintended leaks.

These modifications effectively mitigated the control flow hijacking, preventing the execution of injected shellcode.

Fixing the Vulnerability in Exercise 5

Exercise 5 exploited a non-executable stack to launch a return-to-libc attack. To prevent this attack, we took the following measures:

- Randomized memory layout to make return-oriented programming (ROP) more difficult.
- Restricted function pointers and return addresses from being overwritten.
- Implemented additional input validation to prevent unintended manipulation of function return addresses.
- Used safer memory handling techniques, such as validating stack frame integrity before function returns.

After implementing these fixes, we verified that our updated server code successfully blocked the exploit attempts. Running sudo make check-fixed confirmed that all previous exploits failed, indicating the vulnerabilities had been mitigated.
The following updates were applied to `zookd.c`:

- Ensured all buffers are correctly size-limited to prevent buffer overflows.
- Removed the `accidentally()` function to eliminate unintended ROP gadgets that could be exploited.
- Used `sizeof()` consistently to ensure safe memory handling.

The following updates were applied to `http.c`:

- Replaced unbounded `recv()` and `strcpy()` with safe alternatives (`snprintf()` and `strncpy()`).
- Limited buffer sizes using `MAX_BUF_SIZE` (4096 bytes) to prevent stack corruption.
- Ensured null termination after copying data to mitigate potential overflows.

we found several potentially unsafe function calls in the provided source code files (http.c and zookd.c) that could allow an attacker to perform a buffer overflow attack and overwrite the return address.

```
student@65660-v23:~/lab$ sudo make check-fixed
if [ -x zookclean.py ]; then ./zookclean.py; fi
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 -Wno-format-overflow -D_GNU_SOURCE -static -fno-s
tack-protector
cc http.c -c -o http.o -m64 -g -std=c99 -Wall -Wno-format-overflow -D_GNU_SOURCE -static -fno-sta
ck-protector
cc -m64  zookd.o http.o   -o zookd
cc -m64 zookd.o http.o   -o zookd-exstack -z execstack
cc -m64 zookd.o http.o   -o zookd-nxstack
cc zookd.c -c -o zookd-withssp.o -m64 -g -std=c99 -Wall -Wno-format-overflow -D_GNU_SOURCE -stati
c
cc http.c -c -o http-withssp.o -m64 -g -std=c99 -Wall -Wno-format-overflow -D_GNU_SOURCE -static
cc -m64  zookd-withssp.o http-withssp.o   -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 -Wno-format-overflow -D_GNU_SOURC
E -static -fno-stack-protector
cc -m64  run-shellcode.o   -o run-shellcode
for f in ./exploit-2*.py; do ./check-crash.sh zookd-exstack $f; done
PASS ./exploit-2.py
for f in ./exploit-4*.py; do ./check-attack.sh zookd-exstack $f; done
FAIL ./exploit-4.py
for f in ./exploit-5*.py; do ./check-attack.sh zookd-nxstack $f; done
FAIL ./exploit-5.py
```

Figure 6: Result Exercise - 7

Result

	Filename	Line Number	Code Snippet
0	http.c	94	envp += sprintf(envp, "REQUEST_METHOD=%s", buf...
1	http.c	95	envp += sprintf(envp, "SERVER_PROTOCOL=%s", sp...
2	http.c	101	envp += sprintf(envp, "QUERY_STRING=%s", qp + ...
3	http.c	107	envp += sprintf(envp, "REQUEST_URI=%s", reqpat...
4	http.c	109	envp += sprintf(envp, "SERVER_NAME=zoobar.org"...
5	http.c	150	sprintf(envvar, "HTTP_%s", buf);
6	http.c	153	sprintf(envvar, "%s", buf);
7	http.c	198	vasprintf(&msg, fmt, ap);
8	http.c	299	strncat(pn, name, sizeof(pn) - strlen(pn) - 1);
9	http.c	363	strcpy(dst, dirname);
10	http.c	365	strcat(dst, "/");
11	http.c	366	strcat(dst, filename);
12	http.c	511	vasprintf(&s, fmt, ap);

Figure 7: Result