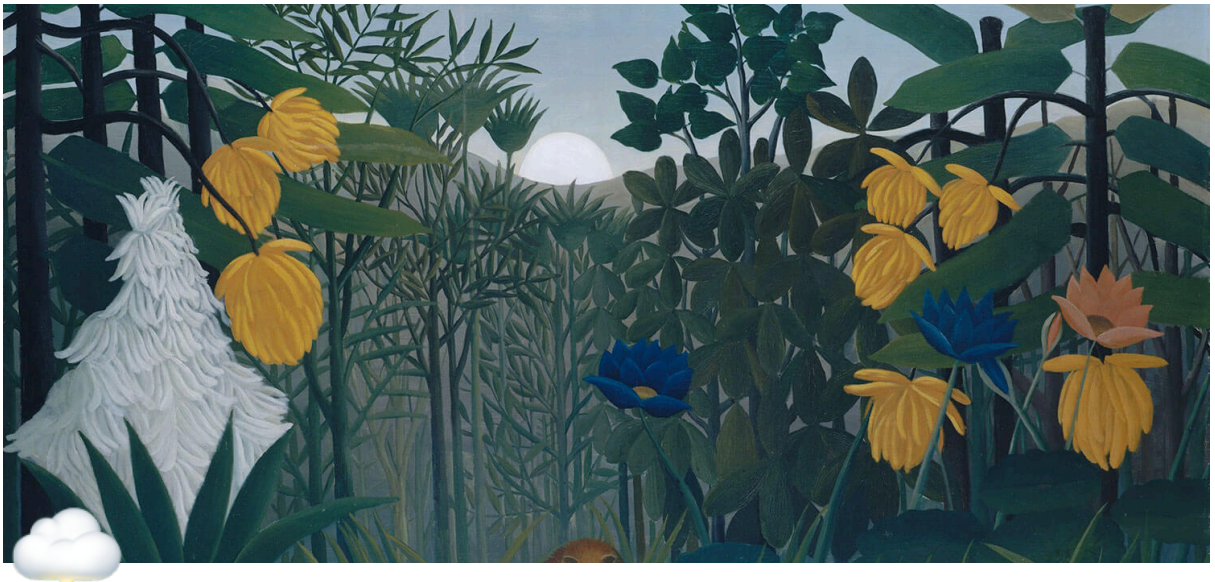


Writeup

> This report summarizes the vulnerabilities hidden in the Flask framework app.py code and several C codes, and how they were patched. I provide the vulnerable portions of the code, how each vulnerability was patched, and the corresponding code changes with a brief explanation of the patch method.

Python: app.py

- The list of hidden vulnerabilities is as follows.

> 1. SQL Injection
> 2. Cross Site Scripting
> 3. SSRF
> 4. Command Injection
> 5. Local File Inclusion (Directory traversal)

1. SQL Injection

A web application security vulnerability that occurs when input data from users is not properly validated. An attacker can exploit this to inject malicious SQL statements into the database to exfiltrate sensitive information or manipulate the database to induce abnormal behavior.

bad code

```
@app.route('/Login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
            ...
```

```
        # Vulnerable to SQL Injection
        # need patch
        # if username == admin and password == anything 'or'1'='1 → always
true
        # → above input makes query like:
        # SELECT * FROM users WHERE username = 'admin' AND password =
'anything' or'1'='1'
        # vulnerable code
        query = "SELECT * FROM users WHERE username = '" + username +
"' AND password = '" + password + "'"
        try:
            cur.execute(query)
            user = cur.fetchone()
            if user:
                session['user_id'] = user['id']
                session['username'] = user['username']
                return redirect(url_for('profile'))
        ...
```

In the code above, the web application trusts user input (for example, from a login form or search box) and uses it directly in the query when generating a database query.

If a user inputs the following values as input, `username ← admin` and `password ← anything ‘ or’1’=’1`, the following query is generated and executed.

Dangerous query: `SELECT * FROM users WHERE username = 'admin' AND password = 'anything' or'1'='1’`

Since the password is not 'anything', the result of password=anything is FALSE, but '1'='1' is TRUE, which returns TRUE. Finally, due to the OR operation, the value (FALSE or TRUE) becomes TRUE, resulting in successful authentication bypass.

Patched code

```
@app.route('/Login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
            ...
```

```
        # patched code
        query = "SELECT * FROM users WHERE username = ? AND password
= ?"
        try:
            cur.execute(query, (username, password))
            user = cur.fetchone()
        ...
```

To fix the problem in the bad code, I used a parameterized query (prepared statement).

Parameterized queries separate the query structure (syntax) from variables (data). Values bound to placeholders (?) are treated as data only by the SQL engine and are not interpreted as part of the query syntax regardless of the characters included. Therefore, even if an attacker inserts malicious SQL fragments (such as ' OR '1'='1) into input values, they are treated as plain string values, preventing SQL Injection.

---

2. Cross Site Scripting

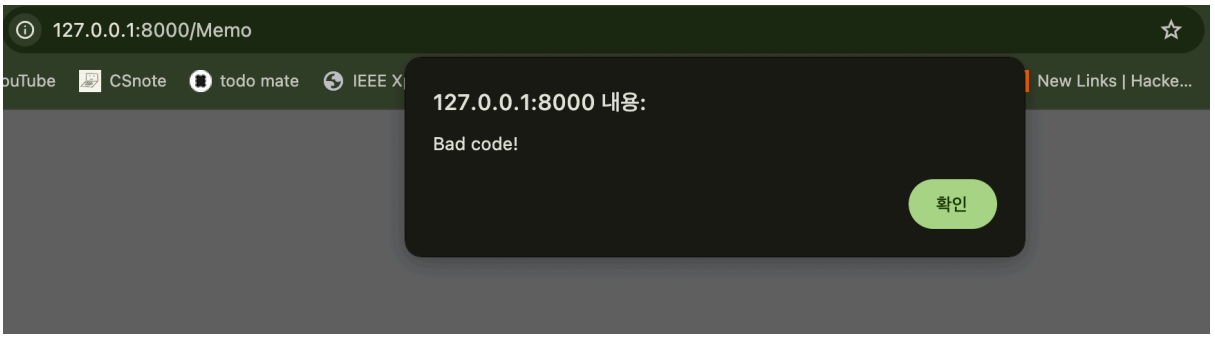
bad code

```
@app.route('/Memo', methods=['GET', 'POST'])
def memo():
        ...
    if request.method == 'POST':
        content = request.form.get('content', '')
        # Vulnerable to XSS
        # need patch
        # It blocks SQL Injection but allows XSS
        # Their is no sanitization or escaping of user input before storing in DB
        # So if user inputs <script>alert('XSS')</script>, it will be stored
        # patched code should sanitize or escape content before storing or re
ndering
        cur.execute('INSERT INTO memos (user_id, content) VALUES (?, ?)', (s
ession['user_id'], content))
        db.commit()
        return redirect(url_for('memo'))
```

```
    cur.execute('SELECT content FROM memos WHERE user_id = ?', (sessio
n['user_id'],))
    memos = cur.fetchall()
    return render_template('memo.html', memos=memos)
```

If a user enters JavaScript code such as <script>alert('XSS')</script> in the memo input field, the application stores it in the database without validation or sanitization (cur.execute('INSERT INTO memos ...')) and renders it directly in the template, which can lead to a stored XSS vulnerability. In this case, the script will execute in the browsers of other users viewing the page, enabling attacks such as session theft, malicious redirection, or UI manipulation.

In fact, when entering <script>alert('Bad code!')</script> and posting it, an alert appears on every visit.



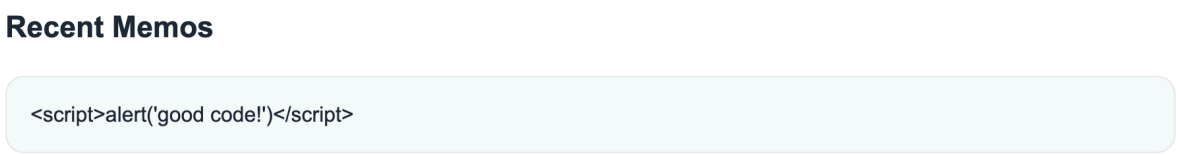
Patched code

```
@app.route('/Memo', methods=['GET', 'POST'])
def memo():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    db = get_db()
    cur = db.cursor()

    if request.method == 'POST':
        content = request.form.get('content', '')

        import html # patched line
        sanitized_content = html.escape(content) # patched line
        cur.execute('INSERT INTO memos (user_id, content) VALUES (?, ?)', (s
```

```
ession['user_id'], sanitized_content)) # patched line

        db.commit()
        return redirect(url_for('memo'))

    cur.execute('SELECT content FROM memos WHERE user_id = ?', (sessio
n['user_id'],))
    memos = cur.fetchall()
    return render_template('memo.html', memos=memos)
```

In the patched code, input values are immediately HTML-escaped on the server. `html.escape()` converts HTML special characters like `<, >, &, "` (and optionally ') into HTML entities (`<, >, &, &quot;`) so that the browser does not interpret them as HTML tags. Therefore, even if a malicious script is entered, it is displayed as plain text in the document and does not execute.

To verify, I saved the attack payload (<script>alert('good code!')</script>) as a memo, then opened the page containing that memo to check that the script was not executed and was displayed as text instead, and that no browser alert appeared.



---

3. SSRF

A SSRF allows an attacker to make the vulnerable server send requests to internal or external systems, or to the server itself. This occurs when server functionality can be manipulated to access or modify resources that would otherwise be inaccessible. The assignment code creates security risks for the same reasons.

bad code

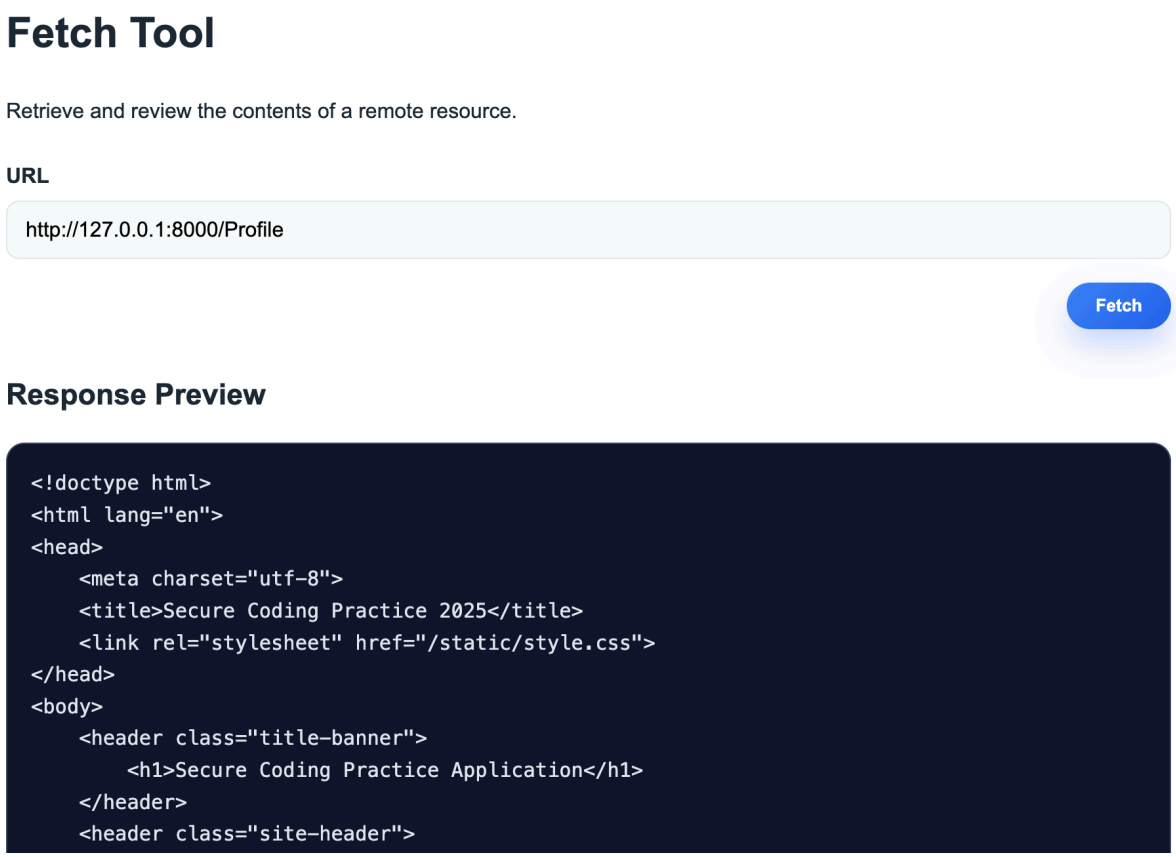
```
@app.route('/Fetch', methods=['GET', 'POST'])
def fetch():
    content = None
```

```
    url = ''
    if request.method == 'POST':
        url = request.form.get('url', '')
        try:
            r = requests.get(url, timeout=10)
            content = r.text[:4096]
        except Exception as e:
            content = f'Error: {e}'
    return render_template('fetch.html', url=url, content=content)
```

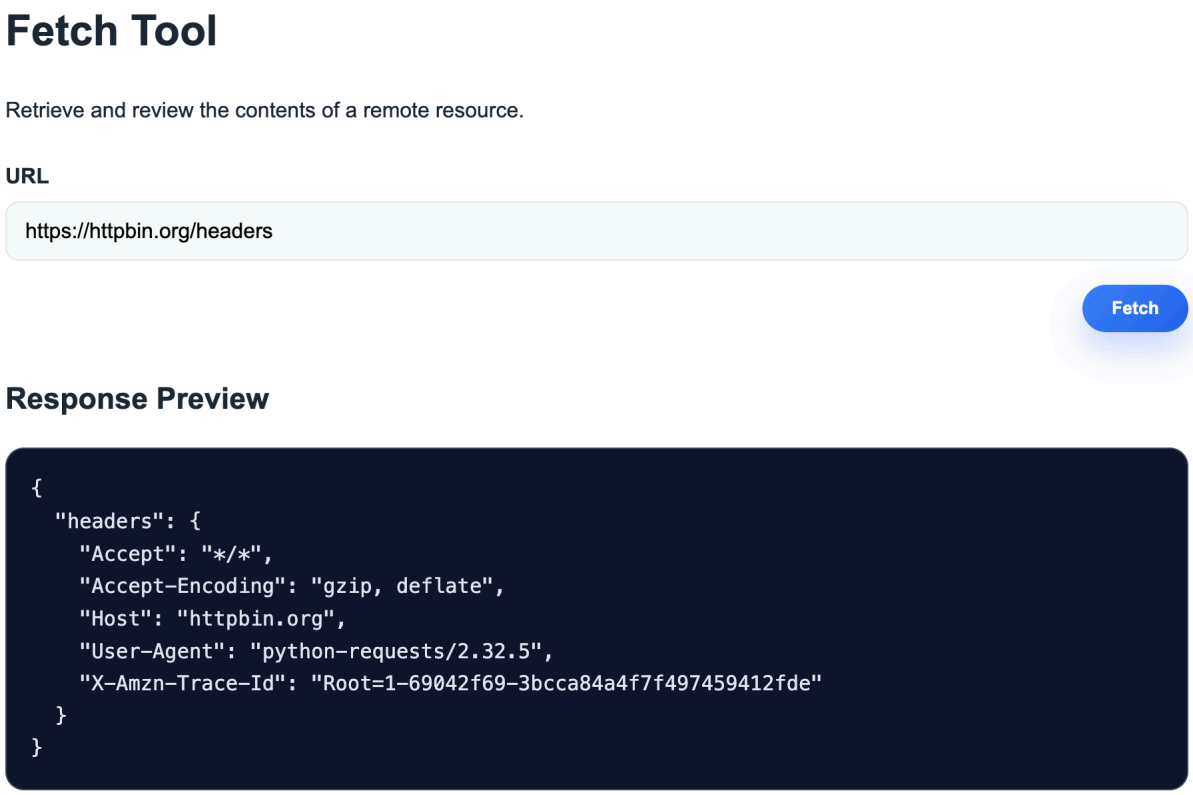
The original code has three major problems:

1. Lack of input validation: no verification of URL scheme, hostname, or IP address
2. Confused trust boundary: user input is treated as trusted data
3. Abuse of server privileges: users can abuse the web server’s network access

Using this, I confirmed that internal HTML files on the server can be leaked. This means that an attacker could bypass authentication and access internal pages, so it is a security vulnerability.



When the `/Fetch` endpoint — which is vulnerable to SSRF — was given the URL `https://httpbin.org/headers`, the server made an HTTP request to that URL and returned the response containing the server's own HTTP headers. An attacker can use this to discover important information about the server's HTTP client library version (e.g. `python-requests/2.32.5`), internal IP-related headers (`X-Forwarded-For`), `User-Agent` strings, and other details about the server's tech stack and network topology. Collected information can be used to find known vulnerabilities, map internal network ranges for follow-up internal attacks, and —worst case—if `Authorization` or `Cookie` headers are exposed, lead to immediate auth bypass or session hijacking. In our tests the endpoint successfully returned the remote response, and we confirmed that parts of the `User-Agent` header (i.e., information about the server or client) were leaked.



Patched code

```
# [Patched Code]
from urllib.parse import urlparse
import ipaddress
import socket
import logging
"""
# Logging setup (example)
logging.basicConfig(
    filename=os.path.join(LOG_DIR, 'ssrf_requests.log'),
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
"""
@app.route('/Fetch', methods=['GET', 'POST'])
def fetch():
    content = None
    url = ''
    if request.method == 'POST':
        url = request.form.get('url', '')
```

```
    # Request logging
    logging.info(f"User {session.get('user_id', 'anonymous')} requested U
RL: {url} from IP: {request.remote_addr}")

    try:
        parsed = urlparse(url)

        # 1. Scheme validation (allow http/https only)
        if parsed.scheme not in ['http', 'https']:
            content = 'Error: Only HTTP/HTTPS protocols are allowed'
            logging.warning(f"Blocked protocol: {parsed.scheme} for URL: {u
rl}")
            return render_template('fetch.html', url=url, content=content)

        # 2. Host presence check
        if not parsed.hostname:
            content = 'Error: Invalid URL format'
            logging.warning(f"Invalid URL format: {url}")
            return render_template('fetch.html', url=url, content=content)

        # 3. IP address validation function
        def is_safe_host(hostname):
            """
            Block internal/private IPs and metadata service
            """
            try:
                # If hostname is an IP literal
                ip = ipaddress.ip_address(hostname)

                if ip.is_private:
                    return False
                if ip.is_loopback:
                    return False
                if ip.is_link_local:
                    return False
                if ip.is_reserved:
                    return False
```

```
                if str(ip) == '169.254.169.254':
                    return False
                return True
            except ValueError:
                # Domain name: resolve and validate the resolved IP
                try:
                    resolved_ip = socket.gethostbyname(hostname)
                    ip = ipaddress.ip_address(resolved_ip)
                    if ip.is_private or ip.is_loopback or ip.is_link_local or ip.is_res
erved:
                        return False
                    if str(ip) == '169.254.169.254':
                        return False
                    return True
                except socket.gaierror:
                    return False

        # Host safety check
        if not is_safe_host(parsed.hostname):
            content = 'Error: Access to internal/private networks is forbidden'
            logging.warning(f"Blocked internal IP access: {parsed.hostname}
for URL: {url}")
            return render_template('fetch.html', url=url, content=content)

        # 4. Domain allowlist
        ALLOWED_DOMAINS = [
            'httpbin.org',
            'api.github.com',
            'jsonplaceholder.typicode.com',
            'www.example.com'
        ]

        if parsed.hostname not in ALLOWED_DOMAINS:
            content = f'Error: Domain "{parsed.hostname}" is not in the allow
ed list'
            logging.warning(f"Domain not whitelisted: {parsed.hostname}")
            return render_template('fetch.html', url=url, content=content)
```

```
            # 5. Port restriction (only 80, 443)
            port = parsed.port or (443 if parsed.scheme == 'https' else 80)
            if port not in [80, 443]:
                content = f'Error: Only ports 80 and 443 are allowed (requested:
{port})'
                logging.warning(f"Blocked port access: {port} for URL: {url}")
                return render_template('fetch.html', url=url, content=content)

            # 6. Harden request execution
            r = requests.get(
                url,
                timeout=5,
                allow_redirects=False,
                headers={
                    'User-Agent': 'SecureWebApp/1.0',
                    'Accept': 'text/html,text/plain'
                },
                verify=True
            )

            # 7. Response size limit
            if len(r.content) > 1048576:  # 1MB
                content = 'Error: Response size exceeds 1MB limit'
                logging.warning(f"Response too large for URL: {url}")
            else:
                content = r.text[:4096]
                logging.info(f"Successful fetch from URL: {url}")

        except requests.exceptions.SSLError as e:
            content = f'SSL Error: Invalid certificate'
            logging.error(f"SSL error for URL {url}: {e}")
        except requests.exceptions.Timeout:
            content = 'Error: Request timeout'
            logging.error(f"Timeout for URL: {url}")
        except requests.exceptions.RequestException as e:
            content = f'Request error: Connection failed'
            logging.error(f"Request exception for URL {url}: {e}")
        except Exception as e:
```

```
            content = f'Validation error: {str(e)}'
            logging.error(f"Unexpected error for URL {url}: {e}")

    return render_template('fetch.html', url=url, content=content)
```

To defend against SSRF, I applied a defense-in-depth strategy.

First, parse the input URL with urlparse(), then allow only HTTP and HTTPS with `parsed.scheme not in ['http', 'https']`, blocking dangerous schemes such as gopher://, dict://, ftp://, etc.

Second, when the host is an IP literal, use the ipaddress module to block RFC 1918 private ranges (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16), loopback (127.0.0.0/8, ::1), link-local (169.254.0.0/16), and reserved addresses via is_private, is_loopback, is_link_local, and is_reserved. In particular, explicitly block the AWS metadata service at 169.254.169.254 to prevent cloud metadata exposure.

Third, to mitigate DNS rebinding, resolve domain inputs with socket.gethostbyname() and apply the same blacklist checks to the resolved IP.

Fourth, complement blacklist limits by introducing a domain allowlist and explicitly allow only domains needed in the actual service (such as httpbin.org, api.github.com, jsonplaceholder.typicode.com, www.example.com).

Fifth, restrict ports so that when parsed.port is specified, only 80 (HTTP) and 443 (HTTPS) are allowed, blocking access to internal service ports such as 22 (SSH), 3306 (MySQL), and 6379 (Redis).

Sixth, execute requests using `requests.get(..., allow_redirects=False, timeout=5, verify=True)` to block redirect-based bypass, mitigate DoS risk with a short timeout, and enforce SSL certificate verification.

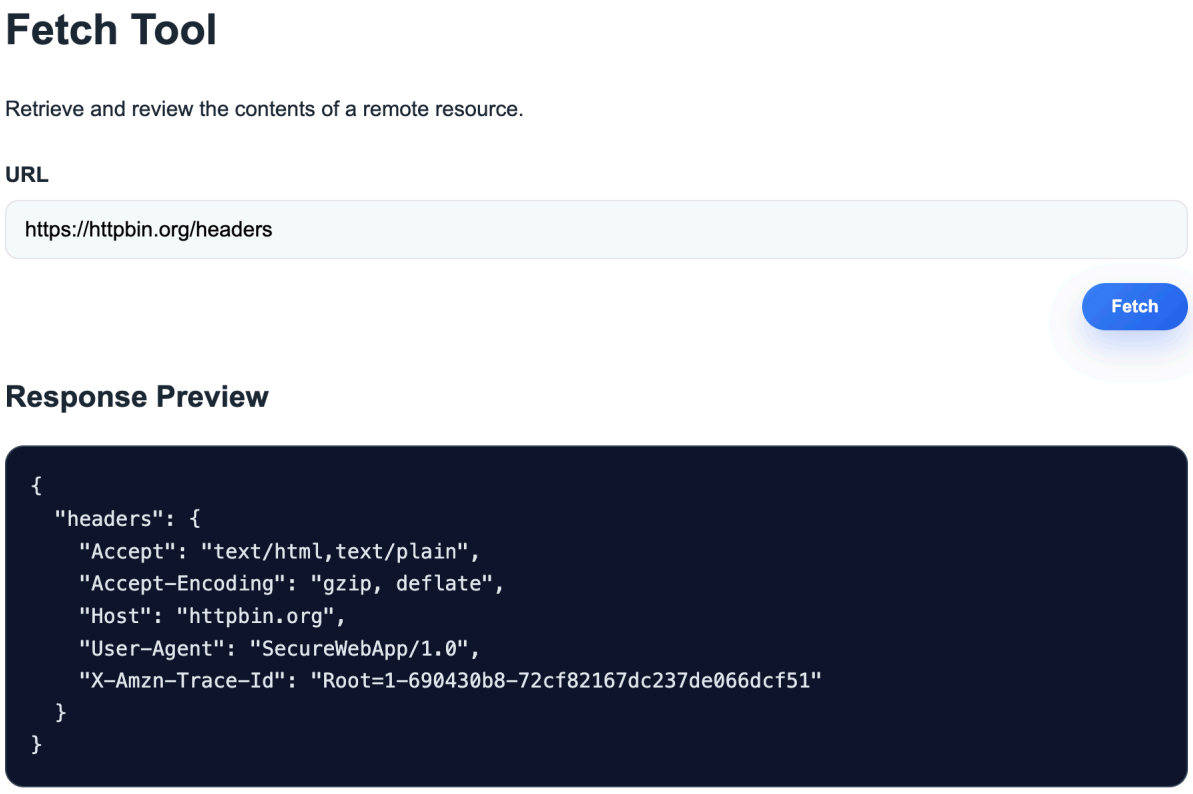
Seventh, check the response size and block requests when `len(r.content) > 1_048_576 (1MB)` to prevent resource exhaustion due to large responses.

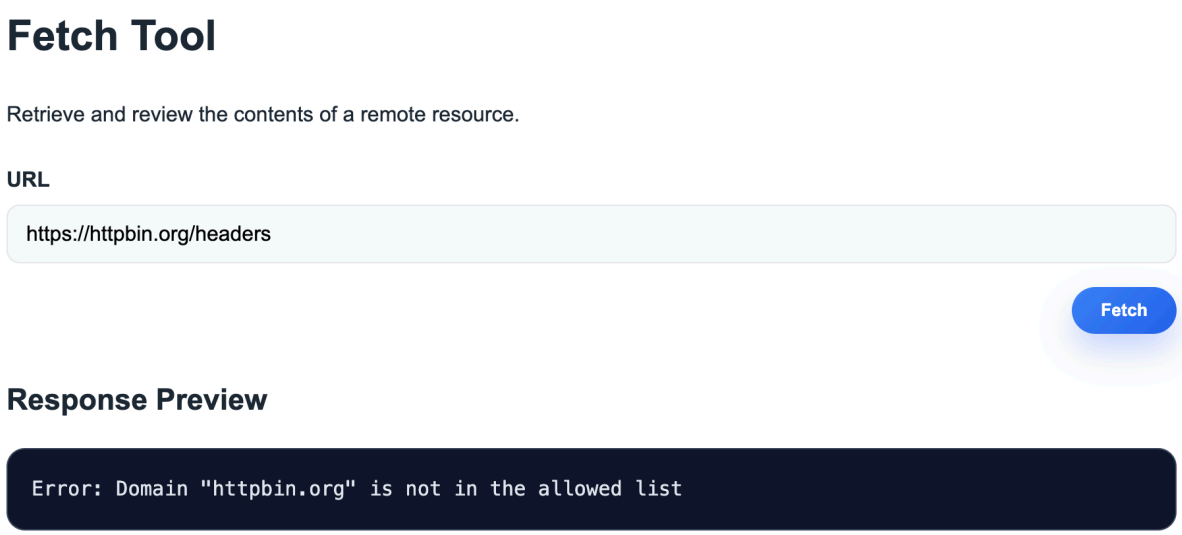
Finally, all attempts are logged for later incident investigation and monitoring.

As a result, the internal server file leakage observed earlier is prevented. Access to required domains can be enabled by adding them to the allowlist as needed.

After patching, I attempted to access internal files and send requests to httpbin.org. As a result, access to internal files was blocked, and I confirmed that information leakage in "User-Agent" has been eliminated. Currently,

httpbin.org is accessible because it is on the whitelist, but access would be blocked if it were removed from the whitelist.





4. Command Injection

bad code

```
# command injection vulnerability
@app.route('/Ping', methods=['GET', 'POST'])
def ping():
    ip_address = ''
    result = None
    if request.method == 'POST':
        ip_address = request.form.get('ip', '') # user input: command injection vulnerability
        count_flag = '-n' if os.name == 'nt' else '-c'
        command = "ping " + count_flag + " 3 " + ip_address # command assembled via string concatenation: vulnerable
        try:
            completed = subprocess.run(
                command,
                shell=True, # using shell: vulnerable
                capture_output=True,
                text=True,
                timeout=15,
            )
            result = completed.stdout or completed.stderr
        except Exception as e:
```

```
            result = f'Error running command: {e}'
    return render_template('ping.html', ip=ip_address, result=result)
```

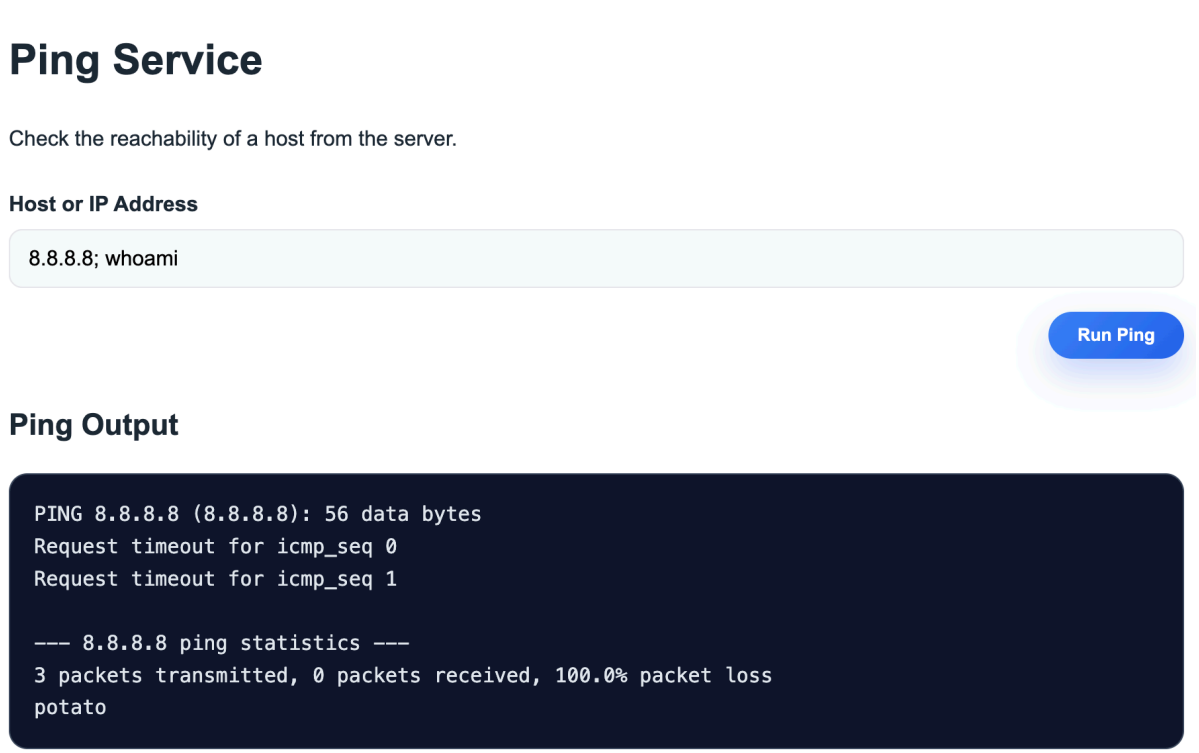
The Command Injection vulnerability occurs when user input is inserted directly into a system command and executed without validation. In this code, three critical security flaws combine:

First, no format validation or filtering is performed on the user input received via `request.form.get('ip', '')`. This means an attacker can input arbitrary strings including shell metacharacters such as semicolons, pipes, and ampersands.

Second, the command is constructed by string concatenation like `"ping " + count_flag + " 3 " + ip_address`. This approach fails to clearly separate user input from the command and allows the input to be interpreted as part of the command.

Third, `subprocess.run()` is executed with `shell=True`, which runs the command through a shell such as `/bin/sh -c` or `cmd.exe`. Shells interpret special characters as control syntax, so if an attacker inserts such metacharacters, additional commands can be executed beyond the intended ping command.

The existence of the vulnerability was confirmed as follows.



Patched code

```
# [Patched Code]
import re
import ipaddress
import logging

@app.route('/Ping', methods=['GET', 'POST'])
def ping():
    ip_address = ''
    result = None
    error = None

    if request.method == 'POST']:
        ip_address = request.form.get('ip', '').strip()

        # Request logging
        logging.info(f"User {session.get('user_id', 'anonymous')} attempted pi
ng to: {ip_address} from IP: {request.remote_addr}")

        # 1. Input length limit
        if len(ip_address) > 45:  # IPv6 max length 39 + margin
            error = 'Error: IP address too long'
            logging.warning(f"IP address too long: {ip_address}")
            return render_template('ping.html', ip=ip_address, result=error)

        # 2. IPv4/IPv6 format validation
        ipv4_pattern = r'^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]
|2[0-4][0-9]|[01]?[0-9][0-9]?)$'
        ipv6_pattern = r'^(([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}|::1|::)$'

        if not (re.match(ipv4_pattern, ip_address) or re.match(ipv6_pattern, ip_
address)):
            error = 'Error: Invalid IP address format'
            logging.warning(f"Invalid IP format: {ip_address}")
            return render_template('ping.html', ip=ip_address, result=error)

        # 3. Dangerous character check (extra layer)
        dangerous_chars = [';', '&', '|', '$', '`', '\n', '\r', '(', ')', '<', '>', '{', '}', '[', ']',
'\\', '"', "'"]
```

```
        if any(char in ip_address for char in dangerous_chars):
            error = 'Error: Invalid characters detected in IP address'
            logging.warning(f"Dangerous characters detected in IP: {ip_addres
s}")
            return render_template('ping.html', ip=ip_address, result=error)

        # 4. Parse and validate IP
        try:
            ip_obj = ipaddress.ip_address(ip_address)

            # Optionally block private/loopback for hardening
            if ip_obj.is_private or ip_obj.is_loopback:
                error = 'Error: Cannot ping private or loopback addresses'
                logging.warning(f"Attempted to ping private/loopback IP: {ip_addr
ess}")
                return render_template('ping.html', ip=ip_address, result=error)

            if ip_obj.is_reserved:
                error = 'Error: Cannot ping reserved IP addresses'
                logging.warning(f"Attempted to ping reserved IP: {ip_address}")
                return render_template('ping.html', ip=ip_address, result=error)

        except ValueError:
            error = 'Error: Invalid IP address'
            logging.warning(f"IP parsing failed: {ip_address}")
            return render_template('ping.html', ip=ip_address, result=error)

        # 5. Safe command execution (shell=False + arg list)
        count_flag = '-n' if os.name == 'nt' else '-c'
        command = ['ping', count_flag, '3', ip_address]

        try:
            completed = subprocess.run(
                command,
                shell=False,
                capture_output=True,
                text=True,
                timeout=15,
```

```
            )
            result = completed.stdout or completed.stderr
            logging.info(f"Successful ping to {ip_address}")

        except subprocess.TimeoutExpired:
            error = 'Error: Ping request timed out'
            logging.warning(f"Ping timeout for IP: {ip_address}")
        except Exception as e:
            error = f'Error: Command execution failed'
            logging.error(f"Ping execution error for IP {ip_address}: {e}")

    return render_template('ping.html', ip=ip_address, result=result)
```

The patched ping endpoint applies the defense-in-depth principle to block command injection.

First, it limits input length (with margin for IPv6) to filter out abnormally long payloads early, and strictly validates IPv4/IPv6 formats using regular expressions to exclude strings that do not match the form (for example, 8.8.8.8; rm -rf /).

Next, it adds a whitelist-reinforcing layer that explicitly checks for shell metacharacters such as ;, &, |, `, and newline characters to counter encoding bypass attempts.

Then, it parses the input with `ipaddress.ip_address()` to ensure the value is a valid IP and blocks access to private, loopback, and reserved addresses using is_private, is_loopback, and is_reserved, preventing internal network scanning and reconnaissance.

Finally, the command is constructed as an argument list like `['ping', count_flag, '3', ip_address]` and invoked with `subprocess.run(..., shell=False)`, eliminating shell interpretation so that even if earlier filters were bypassed, shell metacharacters cannot execute as separate commands.

All attempts are logged with the user identifier and remote IP, categorized by success, block, or exception, for monitoring. This neutralized the prior command injection attack.

5. Local File Inclusion (Directory traversal)

bad code

```
@app.route('/ViewFile')
def view_file():
    raw = request.args.get('filename', '')
    content = None
    error = None
    if raw:
        sanitized = raw.replace('../', '', 1)
        decoded = unquote(sanitized)
        file_path = os.path.join(LOG_DIR, decoded + '.log')
        if '\x00' in file_path:
            file_path = file_path.split('\x00', 1)[0]
        try:
            with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
                content = f.read()
        except Exception as e:
            error = str(e)
    return render_template('view.html', filename=raw, content=content, error
=error)
```

The original code assembles a filesystem path from user input that is not sufficiently validated or normalized and opens it directly, making it vulnerable to local file inclusion (LFI) and directory traversal attacks. Specifically, the flow of

decoding raw with unquote(), forcing an extension with `file_path = os.path.join(LOG_DIR, decoded + '.log')`, and then trimming on `\x00` if present using `split('\x00',1)[0]` creates a safety gap because of the logical order. An attacker can exploit the server's string handling order with null-byte encoding (%00) or various encoding bypasses in the input, and simple replacements like `replace('../', '', 1)` do not block patterns like `..%2f`, multi-repetition, or null-byte bypasses.

In practice, the bypass to expose sample.txt proceeds as follows: when the client sends `filename=sample.txt%00`, the server decodes it to `sample.txt\x00` with unquote(), then constructs something like `.../logs/sample.txt\x00.log` with `decoded + '.log'`. After splitting at the null byte, the result becomes `.../logs/sample.txt`. In other words, the `.log` extension is cut off, and the unintended local file sample.txt is opened and returned. Therefore, the code should immediately block null bytes after decoding, and fundamentally prevent traversal by combining a filename allowlist (for example, `^[A-Za-z0-9_\-]+$`) with boundary checks using `os.path.normpath` and `os.path.realpath` to ensure the path stays within LOG_DIR.

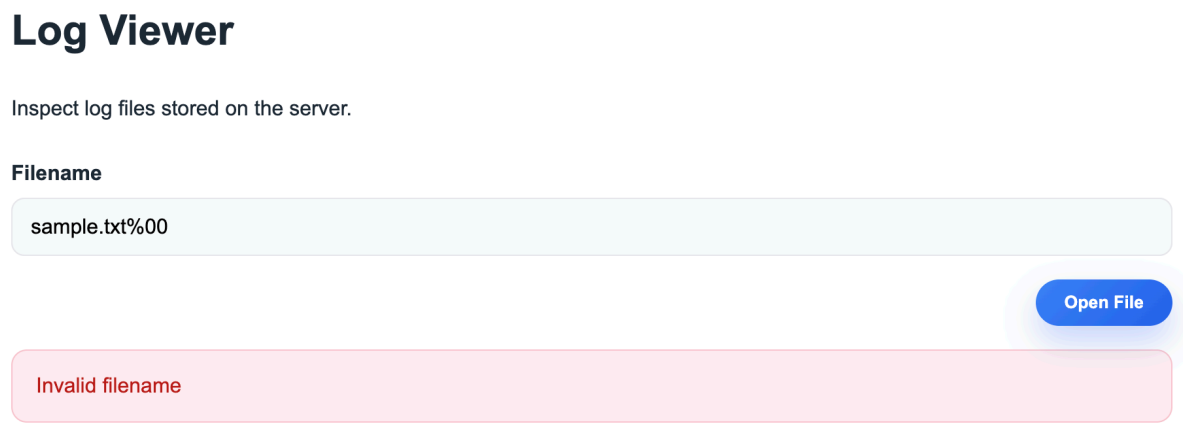
Patched code

```
# [Patched Code]
from urllib.parse import unquote
import os, re

FNAME_RE = re.compile(r'^[A-Za-z0-9_\-]+$')

@app.route('/ViewFile')
def view_file():
```

```
    raw = request.args.get('filename', '')
    content = None
    error = None
    if raw:
        decoded = unquote(raw)
        # 1) Immediately block null bytes
        if '\x00' in decoded:
            error = 'Invalid filename'
            return render_template('view.html', filename=raw, content=content,
error=error)
        # 2) Filename allowlist check
        if not FNAME_RE.match(decoded):
            error = 'Invalid filename'
            return render_template('view.html', filename=raw, content=content,
error=error)
        # 3) Compose path and normalize/boundary check
        filename = decoded + '.log'
        file_path = os.path.normpath(os.path.join(LOG_DIR, filename))
        if not file_path.startswith(os.path.realpath(LOG_DIR) + os.sep):
            error = 'Access denied'
            return render_template('view.html', filename=raw, content=content,
error=error)
        try:
            with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
                content = f.read()
        except Exception as e:
            error = str(e)
    return render_template('view.html', filename=raw, content=content, error
=error)
```

This patch applies defense-in-depth combining input normalization and boundary checks to block Local File Inclusion and directory traversal. Specifically, after decoding the user-provided filename parameter with unquote(), it immediately rejects inputs containing null bytes (x00), strictly validates allowed filename characters with a regex (`^[A-Za-z0-9_\-]+$`), then constructs `decoded + '.log'` , normalizes with `os.path.normpath(os.path.join(LOG_DIR, filename))` , and verifies the resulting path starts with `os.path.realpath(LOG_DIR)` to guarantee the

file always stays within LOG_DIR. Only when all validations pass does it safely open the file and return the response.



C

1. vulnerable1.c (Buffer Overflow)

```
// gcc -fno-stack-protector -z execstack -no-pie -o vulnerable1 vulnerable
1.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(){
    char secret[8] = "SECERET";
    char buffer[8];

    printf("Seceret message : %s\n",secret);

    printf("Input : ");
    gets(buffer);

    if (strcmp(secret, "COSE354") == 0){
        printf("Please patch this code!");
    } else {
        printf("Try again!");
        exit(1);
```

```
    }
}

int main(){
    func();
    return 0;
}
```

1-1. Exploit

The `get()` function in the code can be exploited by abusing its behavior and a buffer-overflow vulnerability. Whether the exploit succeeds is determined by whether the `secret` variable is overwritten by user input with the value `COSE354`, causing the program to print “Please patch this code!”.

`gets(buffer);` stores user input into `buffer` without checking the input length. Also, local variables inside a function are laid out contiguously on the stack (though this can vary with compiler and optimization settings).

Combining these two facts, if a user supplies a very long string for `buffer`, the input can overflow `buffer` and overwrite neighboring stack variables (such as `secret`).

For example, if the input is `aaaaaaaa` (8 `a` characters) followed by `COSE354`, a total of 15 bytes, then `buffer` will be filled with eight `a` characters and the following bytes (`COSE354`) will overflow into `secret[0]` through `secret[6]`. Finally, `gets` writes a terminating `\0` to complete the string.

As a result, `secret` becomes `COSE354`, the `strcmp` check succeeds, and the exploit works.

1-2. Patched code

```
// [patched code]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(){
    char secret[8] = "SECRET";  // fix typo
    char buffer[8];
```

```
    printf("Secret message: %s\n", secret);

    printf("Input: ");

    // use fgets instead of gets to prevent buffer overflow
    if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
        fprintf(stderr, "Input error\n");
        exit(1);
    }

    // delete newline character if present
    size_t len = strlen(buffer);
    if (len > 0 && buffer[len-1] == '\n') {
        buffer[len-1] = '\0';
    }

    if (strcmp(secret, "COSE354") == 0){
        printf("Please patch this code!\n");
    } else {
        printf("Try again!\n");
        exit(1);
    }
}

int main(){
    func();
    return 0;
}
```

To fix the buffer-overflow vulnerability, the first change was to replace `gets()` with `fgets()`. Calling `fgets(buffer, sizeof(buffer), stdin)` reads at most the size given as the second argument, so any input that would exceed the buffer boundary is prevented at source. Concretely, `fgets()` reads up to (size − 1) characters and automatically appends a null terminator (`'\0'`), so for `buffer[8]` a call like `fgets(buffer, 8, stdin)` will read at most seven characters and thus cannot overflow the buffer. This approach was chosen because `fgets()` is the safe input function recommended in the C standard library: unlike `gets()`, it accepts an explicit

buffer size so the programmer can guarantee it operates only within the intended memory bounds. In fact, `gets()` was deprecated in the ISO C99 standard (1999) and removed entirely in ISO C11 (2011) because it is considered dangerous.

Because `fgets()` reads the trailing newline character (`'\n'`) when present, code was added to remove that newline for consistent input handling. After checking the current string length with `strlen(buffer)`, if the final character is a newline it is replaced with a null terminator. This prevents unexpected results when later comparing strings with `strcmp()`. Also, the return value of `fgets()` is validated so input errors or EOF conditions are handled: if `fgets()` returns `NULL`, an error message is printed to standard error and the program exits, ensuring safe behavior even in exceptional conditions.

---

2. vulnerable2.c (Buffer Overflow)

```
// gcc -fno-stack-protector -z execstack -no-pie -o vulnerable2 vulnerable
2.c
// [Original vulnerable code]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func1(){
   char buffer[8];
   int i = 0;
   char c;

   printf("Enter Input : ");

   while((c = getchar()) != '\n' && c != EOF) {
      buffer[i++] = c; // No bounds checking; warning; buffer overflow possi
ble
   }
   buffer[i] = '\0';

   printf("Input : %s\n", buffer);
}
```

```
void func2(){
    char buffer[8];
    char input[10];

    printf("Enter Input : ");
    fgets(input, sizeof(input), stdin);

    sprintf(buffer, "User input: %s", input); // No bounds checking; warning;
buffer overflow possible

    printf("Input : %s\n", buffer);
}
void func3(){
    char buffer[8];
    char input[10];

    printf("Enter Input : ");
    scanf("%s",input);

    strcpy(buffer, input); // No bounds checking; warning; buffer overflow po
ssible

    printf("Copied : %s\n", buffer);
}

int main(){
    func1();
    func2();
    func3();
    return 0;
}
```

2-1. Exploit

The code contains buffer-overflow vulnerabilities in multiple functions. Below is a brief explanation of which function causes the overflow in each case and what input would trigger it.

I inspected the program's internals using LLDB on macOS.

1. `func1`

   The `buffer` array that receives input via `getchar()` has a fixed size, but the `while` loop increments the index `i` without bounds checking. Because of that, if input longer than 8 bytes is provided, it overflows the buffer. For example, giving an input like `a` repeated 20 times (`"aaaaaaaaaaaaaaaaaaaa"`) — i.e. any input larger than the buffer size — will cause a buffer overflow.

   ```
   potato@potatoui-MacBookAir vulnerable_c % lldb ./vulnerable2
   (lldb) target create "./vulnerable2"
   Current executable set to '/Users/potato/Desktop/KU/25-2/정보보호/assi
   gnment2_secure_coding_and_design/vulnerable_c/vulnerable2' (arm6
   4).
   (lldb) run
   Process 57163 launched: '/Users/potato/Desktop/KU/25-2/정보보호/assi
   gnment2_secure_coding_and_design/vulnerable_c/vulnerable2' (arm6
   4)
   Enter Input : aaaaaaaaaaaaaaaaaaaa
   Input : aaaaaaaaaaaaaaaaaaaa
   Process 57163 stopped
   * thread #1, queue = 'com.apple.main-thread', stop reason = signal SIG
   ABRT
       frame #0: 0x000000019eafa388 libsystem_kernel.dylib`__pthread_kil
   l + 8
   libsystem_kernel.dylib`__pthread_kill:
   →  0x19eafa388 <+8>:  b.lo   0x19eafa3a8    ; <+40>
      0x19eafa38c <+12>: pacibsp
      0x19eafa390 <+16>: stp    x29, x30, [sp, #-0x10]!
      0x19eafa394 <+20>: mov    x29, sp
   Target 0: (vulnerable2) stopped.
   ```

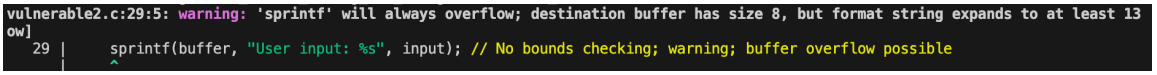
   I observed that supplying input larger than the buffer caused the program to raise `SIGABRT` (abort). In other words, the program terminated abnormally due to the overflow.

2. `func2`

   Inside `func2()` a fixed-size stack buffer (`buffer[8]`) is being written to with `sprintf` that has no length limit, so a buffer overflow always occurs.

Concretely, the prefix `"User input: "` alone requires 12 bytes (and even more if you account for the terminating `\0`), so the 8-byte `buffer` is guaranteed to overflow and corrupt stack memory.

There is compile-time warning log evidence for this.




```
vulnerable2.c:29:5: warning: 'sprintf' will always overflow; destination b
uffer has size 8, but format string expands to at least 13 [-Wformat-over
flow]
   29 |     sprintf(buffer, "User input: %s", input); // No bounds checking;
warning; buffer overflow poss
```

Therefore, a buffer overflow occurs here even without any special exploit—just the existing `sprintf`/formatting behavior is sufficient to overflow the buffer.

3. `func3`

In `main()` the functions are called sequentially, but `func2()` causes a buffer overflow and abnormal termination even with empty input, so I commented out `func2()` and proceeded to test `func3()`.

For `func3()` there are two major problems:

Vulnerability 1 — `scanf()` accepts unlimited input

`scanf("%s")` reads input until it encounters whitespace, so it will accept arbitrarily long input and ignore the size limit of the `input[10]` array. Because no field width is specified (e.g. `%9s`), the input length cannot be controlled and the first overflow can occur.

Vulnerability 2 — `strcpy()` copies unconditionally

`strcpy()` copies bytes until it reaches a null terminator without checking the destination buffer size. If the `input` contents exceed 8 bytes, `strcpy()` will write past `buffer`'s boundary and overwrite adjacent memory.

The input used to exploit this was `a*20` (20 `a` characters).

```
(lldb) run
There is a running process, kill it and restart?: [Y/n] y
```

```
Process 57163 exited with status = 9 (0x00000009) killed
Process 57959 launched: '/Users/potato/Desktop/KU/25-2/정보보호/ass
ignment2_secure_coding_and_design/vulnerable_c/vulnerable2' (arm6
4)
Enter Input : aaaa
Input : aaaa
Enter Input : aaaaaaaaaaaaaaaaaaaa
Process 57959 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BREA
KPOINT (code=1, subcode=0x19ea3adfc)
    frame #0: 0x000000019ea3adfc libsystem_c.dylib`__chk_fail_overflo
w + 24
libsystem_c.dylib`__chk_fail_overflow:
→  0x19ea3adfc <+24>: brk    #0x1

libsystem_c.dylib`__chk_fail_overlap:
    0x19ea3ae00 <+0>:  pacibsp
    0x19ea3ae04 <+4>:  stp    x29, x30, [sp, #-0x10]!
    0x19ea3ae08 <+8>:  mov    x29, sp
Target 0: (vulnerable2) stopped.
```

One noteworthy point in `func3()` is that `__chk_fail_overflow` was called. `__chk_fail_overflow` is a protection routine invoked by the compiler/runtime when a buffer-bounds violation is detected. This means macOS's runtime protection detected a buffer overflow and terminated the program. Therefore, we can confirm that a buffer overflow occurred.

2-2. Patched code

```
// [Patched secure code]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func1_patched(){
    char buffer[8];
    int i = 0;
```

```
    char c;

    printf("Enter Input : ");

    // patch1: added bounds checking
    while((c = getchar()) != '\n' && c != EOF) {
        if (i < sizeof(buffer) - 1) {  // make space for null terminator
            buffer[i++] = c;
        } else {
            // If buffer is full, ignore the rest of the input
            while((c = getchar()) != '\n' && c != EOF);
            break;
        }
    }
    buffer[i] = '\0';

    printf("Input : %s\n", buffer);
}

void func2_patched(){
    char buffer[64];  // patch 2: allocate sufficient buffer size
    char input[10];

    printf("Enter Input : ");
    fgets(input, sizeof(input), stdin);

    // delete newline character if present
    size_t len = strlen(input);
    if (len > 0 && input[len-1] == '\n') {
        input[len-1] = '\0';
    }

    // patch 3: use snprintf to limit buffer size
    snprintf(buffer, sizeof(buffer), "User input: %s", input);

    printf("Input : %s\n", buffer);

    // patch 6: clear input buffer
```

```
    while (getchar() != '\n');
}

void func3_patched(){
    char buffer[8];
    char input[10];

    printf("Enter Input : ");

    // patch 4: add length limit to scanf
    scanf("%9s", input);  // max 9 chars + null terminator

    // patch 5: use strncpy and null terminator
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';  // null terminator

    printf("Copied : %s\n", buffer);
}

int main(){
    func1_patched();
    func2_patched();
    func3_patched();
    return 0;
}
```

`func1_patched()` adds bounds-checking logic to fix the original code's unbounded buffer-write vulnerability. In the original code the `while` loop did `buffer[i++] = c` without checking whether the index `i` exceeded the array size, so if a user supplied 8 bytes or more the stack could be corrupted. In the patched version an `if (i < sizeof(buffer) - 1)` condition is added so the index is always validated before writing to the buffer. The reason for using `sizeof(buffer) - 1` is to reserve space for the string terminator (`'\0'`). If the input exceeds the buffer size, the implementation consumes and discards the remaining input via the inner loop `while ((c = getchar()) != '\n' && c != EOF)` so that the excess data does not affect subsequent input functions. This prevents buffer overflow completely while preserving correct program behavior.

`func2_patched()` addresses the unlimited-write problem caused by `sprintf()` with two main changes. First, the buffer size is increased substantially from 8 bytes to 64 bytes. The original code had a structural flaw: the prefix `"User input: "` alone requires 12 bytes, so attempting to write that into an 8-byte buffer overflowed unconditionally regardless of the actual user input. Second, `sprintf()` is replaced with `snprintf()` and used like `snprintf(buffer, sizeof(buffer), "User input: %s", input)`, explicitly bounding the output length so it never exceeds the buffer. `snprintf()` will truncate the output if it would exceed the given buffer size and always adds a null terminator, so it is safe from overflow. The patched function also strips the newline (`'\n'`) that `fgets()` may include, and clears any remaining input with `while (getchar() != '\n');` so that the input buffer is clean for the next function (`func3`). Final output is produced safely with `printf("Input : %s\n", buffer)`.

`func3_patched()` uses a defense-in-depth approach to fix the buffer-overflow caused by careless use of `strcpy()`. First, `scanf()` is given a length specifier and changed to `scanf("%9s", input)`, limiting input to at most 9 characters (excluding the terminator), which prevents the initial overflow into the 10-byte `input` array. Second, copying is done with `strncpy(buffer, input, sizeof(buffer) - 1)`, explicitly limiting the number of bytes written to the destination buffer; `sizeof(buffer) - 1` leaves room for the null terminator. Because `strncpy()` does not automatically append a terminator when the source is longer than the destination, the patch explicitly sets `buffer[sizeof(buffer) - 1] = '\0'` to guarantee correct termination. These layered defenses ensure no buffer overflow can occur, and the safely truncated string is printed with `printf("Copied : %s\n", buffer)`.

---

3. vulnerable3.c (Interger Over/Underflow)

```
// gcc -fno-stack-protector -z execstack -no-pie -o vulnerable3 vulnerable
3.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

typedef struct {
    uint32_t id;
    uint32_t balance;
} Account;
```

```
static void show(const Account *a){
    printf("[Account %" PRIu32 "] balance=%" PRIu32 " cents\n", a->id, a->
balance);
}

int deposit(Account *a, uint32_t amount){
   if((int32_t)amount < 0) return -1;
     a->balance += amount;
     return 0;
}

int withdraw(Account *a, int32_t amount){
   if (amount <= 0) return -1;
   uint32_t new_balance = a->balance - amount;
   if (new_balance < 0) return -2;
   a->balance = new_balance;
   return 0;
}

int adjust(Account *a, int32_t delta){
   uint32_t new_balance = a->balance + delta;
   a->balance = new_balance;
   return 0;
}

int main(){
   Account a = { .id =1, .balance = 1000 };

   a.balance = 4294967290;
   show(&a);
   if(deposit(&a, 10)==0){
      printf("Success!\n");
      show(&a);
   }
   printf("\n");
   a.balance = 100;
   show(&a);
```

```
    if(withdraw(&a, 500) == 0){
        printf("Success!\n");
        show(&a);
    }
    printf("\n");

    a.balance = 1000;
    show(&a);
    if(adjust(&a, -2000) == 0){
        printf("Success!\n");
        show(&a);
    }

    return 0;
}
```

3-1. Problem

1. Integer overflow vulnerability in `deposit()`

The `deposit()` function is supposed to add an amount to an account balance, but it has a serious vulnerability because it performs no check for integer overflow. The test inside the function — `if ((int32_t)amount < 0) return -1;` — only casts `amount` to a signed 32-bit integer to check for negativity; it does **not** verify whether the addition will exceed the maximum value of `uint32_t` (4,294,967,295). For example, if `a->balance = 4,294,967,290` and `amount = 10`, the mathematical result should be `4,294,967,300`, but that exceeds `UINT32_MAX` by 4. C unsigned-integer arithmetic wraps modulo 2^{32}, so the stored result becomes:

```
(4,294,967,290 + 10) % 2^32 = 4,294,967,300 % 4,294,967,296 = 4
```

Thus, depositing 10 cents into an account that already has 4,294,967,290 cents results in a final balance of only 4 cents — a catastrophic outcome.

2. Unsigned-underflow vulnerability in `withdraw()`

The `withdraw()` function attempts to withdraw money but implements a logically broken check because it ignores the semantics of unsigned integers. The code computes:

```
uint32_t new_balance = a->balance - amount;
if (new_balance < 0) return -2;
```

However, `new_balance` is `uint32_t`, so the condition `new_balance < 0` is always false. Unsigned integers cannot be negative; when a subtraction would go below zero, the result underflows and becomes a large positive number via modulo 2322^{32}232. For example, if `a->balance = 100` and `amount = 500`, mathematically the result is `-400`, but with unsigned arithmetic:

```
100 - 500 = 100 + (2^32 - 500) = 4,294,966,896
```

(That is, $2^{32} - 400 = 4,294,966,896$.) The outcome is that attempting to withdraw more than the balance inflates the stored balance to a huge value instead of rejecting the operation.

3. **Signed/unsigned mixing vulnerability in `adjust()`**

   The `adjust()` function allows adjusting the account balance by a signed delta (`int32_t delta`) but adds it directly to an unsigned balance:

   ```
   uint32_t new_balance = a->balance + delta;
   ```

   If `delta` is negative, C's integer conversion rules promote/convert the negative `delta` to `uint32_t`, producing a large unsigned value (two's complement interpretation). For instance, `delta = -2000` corresponds to the 32-bit two's-complement value `0xFFFFF830`, which as an unsigned 32-bit integer equals `4,294,965,296`. So if `a->balance = 1000`:

   ```
   1000 + 4,294,965,296 = 4,294,966,296
   ```

   Instead of reducing the balance by 2,000 (to −1,000), the stored balance becomes ~4.29 billion cents. Equally, if `delta` is positive and the sum exceeds `UINT32_MAX`, an overflow will wrap the value and produce a small, incorrect balance. The function also always returns `0` (success) and performs no validation or overflow/underflow checks, so callers are never informed that an invalid or dangerous adjustment occurred.

3-2. Patched code

```
// [Patched secure code]
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>
#include <limits.h>

typedef struct {
    uint32_t id;
    uint32_t balance;
} Account;

static void show(const Account *a){
     printf("[Account %" PRIu32 "] balance=%" PRIu32 " cents\n", a->id, a->
balance);
}

// patch1: deposit() - added overflow check
int deposit(Account *a, uint32_t amount){
    // check for negative when cast to signed
    if((int32_t)amount < 0) {
        fprintf(stderr, "Error: Invalid amount (negative when cast to signed)
\n");
        return -1;
    }

    // check for overflow before performing addition
    // safe check a->balance + amount > UINT32_MAX
    // transform equation: amount > UINT32_MAX - a->balance
    if (amount > UINT32_MAX - a->balance) {
        fprintf(stderr, "Error: Deposit would cause overflow (balance=%" PRIu
32 ", amount=%" PRIu32 ")\n",
                a->balance, amount);
        return -2;
    }

    a->balance += amount;
    return 0;
```

```
}

// patch2: withdraw() - correct underflow check
int withdraw(Account *a, int32_t amount){
    // validate positive amount
    if (amount <= 0) {
        fprintf(stderr, "Error: Withdrawal amount must be positive\n");
        return -1;
    }

    // correct check for not enough balance
    // uint32_t cannot be negative, so check before subtraction
    if (a->balance < (uint32_t)amount) {
        fprintf(stderr, "Error: Insufficient balance (have=%" PRIu32 ", need
=%" PRId32 ")\n",
                a->balance, amount);
        return -2;
    }

    // safe to subtract
    a->balance -= (uint32_t)amount;
    return 0;
}

// patch 3: adjust() - signed/unsigned safety handling
int adjust(Account *a, int32_t delta){
    // when delta is negative
    if (delta < 0) {
        // safe calculation for absolute value
        // using int64_t to handle INT32_MIN case
        uint32_t abs_delta = (uint32_t)(-(int64_t)delta);

        // check for underflow
        if (a->balance < abs_delta) {
            fprintf(stderr, "Error: Adjustment would cause underflow (balance
=%" PRIu32 ", delta=%" PRId32 ")\n",
                    a->balance, delta);
            return -1;
```

```
        }

        a->balance -= abs_delta;
    }
    // when delta is positive (increase)
    else if (delta > 0) {
        uint32_t pos_delta = (uint32_t)delta;

        // check for overflow
        if (pos_delta > UINT32_MAX - a->balance) {
            fprintf(stderr, "Error: Adjustment would cause overflow (balance
=%" PRIu32 ", delta=%" PRId32 ")\n",
                    a->balance, delta);
            return -2;
        }

        a->balance += pos_delta;
    }
    // when delta == 0, no operation is performed

    return 0;
}

int main(){
    Account a = { .id =1, .balance = 1000 };

    printf("=== Test 1: Deposit Overflow Prevention ===\n");
    a.balance = 4294967290;
    show(&a);
    printf("Attempting to deposit 10 cents...\n");
    if(deposit(&a, 10)==0){
        printf("Success!\n");
        show(&a);
    } else {
        printf("Failed! Overflow prevented.\n");
    }
    printf("\n");
```

```
printf("=== Test 2: Withdraw Underflow Prevention ===\n");
a.balance = 100;
show(&a);
printf("Attempting to withdraw 500 cents...\n");
if(withdraw(&a, 500) == 0){
    printf("Success!\n");
    show(&a);
} else {
    printf("Failed! Insufficient balance.\n");
}
printf("\n");

printf("=== Test 3: Adjust Underflow Prevention ===\n");
a.balance = 1000;
show(&a);
printf("Attempting to adjust by -2000 cents...\n");
if(adjust(&a, -2000) == 0){
    printf("Success!\n");
    show(&a);
} else {
    printf("Failed! Underflow prevented.\n");
}
printf("\n");

printf("=== Test 4: Normal Operations ===\n");
a.balance = 10000;
show(&a);
printf("Deposit 500 cents...\n");
deposit(&a, 500);
show(&a);
printf("Withdraw 200 cents...\n");
withdraw(&a, 200);
show(&a);
printf("Adjust by -100 cents...\n");
adjust(&a, -100);
show(&a);
```

```
    return 0;
}
```

While patching, I added test cases to the main function and included printf statements to more clearly demonstrate the experimental results.

1. `deposit()` function patch

The patch for `deposit()` adds a defensive mechanism to detect and block integer overflow before it happens. The original code performed `a->balance += amount` without any check for exceeding `UINT32_MAX`. The patched code adds this pre-check:

```
if (amount > UINT32_MAX - a->balance) {
    // overflow would occur
    return -2;
}
```

This uses an algebraic rearrangement to avoid performing the overflowing addition directly: instead of testing `a->balance + amount > UINT32_MAX` (which itself could overflow), it tests `amount > UINT32_MAX - a->balance`. The subtraction on the right-hand side is always safe to compute in the unsigned range. For example, if `a->balance = 4294967290` and `amount = 10`, then `10 > 4294967295 - 4294967290`, i.e. `10 > 5` is true, so the overflow is detected and the function returns an error code `-2` without modifying the balance. The patch also logs overflow attempts via `fprintf(stderr, ...)` to aid auditing and debugging.

2. `withdraw()` function patch

The `withdraw()` patch fixes the flawed unsigned-underflow check in the original code. The original attempt to detect underflow by computing `uint32_t new_balance = a->balance - amount; if (new_balance < 0) ...` is meaningless because an unsigned value can never be negative. The patched code instead prevents underflow by checking *before* subtracting:

```
if (a->balance < (uint32_t)amount) {
    // insufficient funds / underflow would occur
    return -2;
}
```

This direct comparison blocks the subtraction when the balance is less than the requested withdrawal. Casting `amount` to `uint32_t` makes the intended conversion explicit and improves readability; for non-negative `amount` the cast is safe. For example, with `a->balance = 100` and `amount = 500`, `100 < 500` is true, so the function returns `-2` and does not perform the subtraction. This prevents the pathological case where a withdrawal would turn `100` cents into `~4.29 billion` cents due to unsigned underflow.

3. `adjust()` function patch

The `adjust()` patch addresses the dangerous mixing of signed and unsigned integers by explicitly handling negative and positive `delta` values with separate branches. The original code did:

```
uint32_t new_balance = a->balance + delta;
```

which incorrectly promotes a negative `delta` into a large unsigned value. The patched logic branches on the sign of `delta`:

- If `delta < 0` (negative adjustment):
  - Compute the absolute adjustment safely using an intermediate 64-bit signed type to avoid `INT32_MIN` corner-case overflow:

    ```
    uint32_t abs_delta = (uint32_t)(-(int64_t)delta);
    ```

    (Using `int64_t` for negation prevents overflow when `delta == INT32_MIN`.)
  - Check for underflow:

    ```
    if (a->balance < abs_delta) return -1;
    a->balance -= abs_delta;
    ```

    For example, with `balance = 1000` and `delta = -2000`, `abs_delta = 2000` and `1000 < 2000` is true, so the function returns an error instead of producing a huge unsigned result.
- Else if `delta > 0` (positive adjustment):
  - Convert to unsigned and perform the same overflow check used in `deposit()`:

```
uint32_t pos_delta = (uint32_t)delta;
if (pos_delta > UINT32_MAX - a->balance) return -2;
a->balance += pos_delta;
```

This branching approach replaces the original always-`return 0` behavior with explicit error codes (`-1` or `-2`) depending on the failure type, so callers can distinguish underflow vs overflow. As a result, attempts such as adjusting `balance = 1000` by `delta = -2000` no longer convert into ~4.29 billion cents — they are rejected safely.

---

4. vulnerable4.c (format string)

```
// gcc -fno-stack-protector -z execstack -no-pie -o vulnerable4 vulnerable
4.c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <syslog.h>


void log_user_input(const char *input){
    FILE *f = fopen("log.txt","at");
    if(f){
        fprintf(f, input);
        fprintf(f,"\n");
        fclose(f);
    }
    openlog("log", LOG_PID | LOG_CONS, LOG_USER);
    syslog(LOG_INFO, input);
    closelog();
}

int main(int argc, char *argv[]) {
    if (argc != 2){
        fprintf(stderr, "Usage: %s <message>\n", argv[0]);
        return 1;
```

```
    }
    const char *msg = argv[1];
    log_user_input(msg);

    return 0;

}
```

4-1. Exploit

The code contains two serious format-string vulnerabilities inside `log_user_input()`.

First vulnerability — `fprintf(f, input);`

This call uses the user-supplied `input` string directly as `fprintf()`'s format string. Correct code should treat the user data as plain data, e.g. `fprintf(f, "%s", input);`. As written, any format specifiers in `input` (for example `%x`, `%s`, `%p`, `%n`) are interpreted and executed by `fprintf()`. An attacker can therefore inject specifiers to read or manipulate memory.

Second vulnerability — `syslog(LOG_INFO, input);`

`syslog()` is also a `printf`-family function that treats its second argument as a format string. Passing `input` directly to `syslog()` reproduces the same problem: attacker-controlled format specifiers will be executed and logged (to system log), leaking or corrupting memory.

Examples / consequences:

- If an attacker supplies `"%x.%x.%x.%x"`, the program will print four 4-byte stack values in hex, leaking stack contents (return addresses, local variables, library pointers, etc.) to `log.txt` and the system log.
- Use of the `%n` specifier is more dangerous: `%n` writes the number of bytes printed so far into an address provided by the corresponding argument. With carefully crafted inputs and stack manipulation, an attacker can write arbitrary values to arbitrary memory locations (e.g., overwrite a return address or an entry in the GOT), leading to control-flow hijacking and arbitrary code execution.

The results of exploiting the two vulnerabilities above are as follows.

```
potato@potatoui-MacBookAir vulnerable_c % ./vulnerable4 "%p.%p.%p.%p.%p.%p.%p.%p"
```

```
potato@potatoui-MacBookAir vulnerable_c % cat log.txt
0x20cace940.0x16d50b386.0x16d50aaf0.0x1028f455c.0xffffffff0009d71d.0
x16d50b386.0x16d50b160.0x2
```

```
potato@potatoui-MacBookAir vulnerable_c % ./vulnerable4 "%x.%x.%x.%
x"
potato@potatoui-MacBookAir vulnerable_c % cat log.txt
cace940.6b8ff38e.6b8feaf0.450055c
```

4-2. Patched code

```
// [Patched secure code]
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <syslog.h>

// patched secure logging function
void log_user_input(const char *input){
    FILE *f = fopen("log.txt","at");
    if(f){
        // patch 1: treat input as data, not format string
        fprintf(f, "%s", input);  // safe usage
        fprintf(f,"\n");
        fclose(f);
    }
    openlog("log", LOG_PID | LOG_CONS, LOG_USER);
    // patch 2: syslog also modified similarly
    syslog(LOG_INFO, "%s", input);  // safe usage
    closelog();
}

int main(int argc, char *argv[]) {
    if (argc != 2){
        fprintf(stderr, "Usage: %s <message>\n", argv[0]);
```

```
        return 1;
    }
    const char *msg = argv[1];
    log_user_input(msg);

    return 0;
}
```

The core change is to stop using the user-supplied `input` directly as a format string in two places inside `log_user_input()` and instead handle it as data via fixed format specifiers.

The first fix changes: `fprintf(f, input);` to `fprintf(f, "%s", input);`

This ensures the `input` string is treated as plain string data matched to the `"%s"` format specifier instead of being interpreted as a format string. As a result, any user-supplied format specifiers such as `"%x"`, `"%p"`, or `"%n"` are printed literally and are **not** executed as format operations. This blocks all form of format-string attacks (memory reads, arbitrary memory writes, crashes, etc.) at the source.

The second fix applies the same principle to the system logger: `syslog(LOG_INFO, input);` becomes `syslog(LOG_INFO, "%s", input);` so that system log entries are recorded at the same safety level.

These modifications do not change program logic or behavior beyond eliminating the vulnerability; they are minimal and sufficient—simply adding `"%s"` to two function calls. After the patch, attempting an attack like:

```
./vulnerable4_patched "AAAA%x%x%x%n"
```

will cause `log.txt` to contain the literal string:

```
AAAA%x%x%x%n
```

and will no longer leak stack memory or permit arbitrary memory writes. The program's safety is therefore fully preserved.

```
potato@potatoui-MacBookAir vulnerable_c % ./vulnerable4 "%x.%x.%x.%
x"
```

```
potato@potatoui-MacBookAir vulnerable_c % cat log.txt
%x.%x.%x.%x
```

```
potato@potatoui-MacBookAir vulnerable_c % ./vulnerable4 "%p.%p.%p.%
p.%p.%p.%p.%p"
potato@potatoui-MacBookAir vulnerable_c % cat log.txt
%p.%p.%p.%p.%p.%p.%p.%p
```

```
potato@potatoui-MacBookAir vulnerable_c % ./vulnerable4 "AAAA%x%x%
x%n"
potato@potatoui-MacBookAir vulnerable_c % cat log.txt
AAAA%x%x%x%n
```

---

5. vulnerable5.c (Race condition)

```
// gcc -fno-stack-protector -z execstack -no-pie -pthread -o vulnerable5 v
ulnerable5.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

typedef struct {
    int ticket_id;
    int available_seats;
    int total_sold;
    char event_name[50];
} TicketSystem;

TicketSystem concert = {
    .ticket_id = 1,
    .available_seats = 10,
    .total_sold = 0,
```

```
    .event_name = "SECURITY Concert"
};

typedef struct {
    int user_id;
    char name[20];
} User;


void* book_ticket(void* arg) {
    User *user = (User*)arg;

    printf("[%s] Checking availability...\n", user->name);

    if (concert.available_seats > 0) {
        printf("[%s] Found available seat! Processing payment...\n", user->na
me);
        usleep(500000);

        concert.available_seats--;
        concert.total_sold++;

        printf("[%s] ✓ Booking SUCCESS! Remaining seats: %d\n", user->nam
e, concert.available_seats);
    } else {
        printf("[%s] × SOLD OUT!\n", user->name);
    }

    return NULL;
}

void print_info() {
    printf("Available Seats: %d\n", concert.available_seats);
    printf("Total Sold: %d\n", concert.total_sold);
}

int main() {
    pthread_t threads[20];
```

```
    User users[20];

    printf("=== Initial State ===\n");
    print_info();

    printf("Starting booking...\n\n");

    for (int i = 0; i < 20; i++) {
        users[i].user_id = i + 1;
        snprintf(users[i].name, sizeof(users[i].name), "User%02d", i + 1);
        pthread_create(&threads[i], NULL, book_ticket, &users[i]);
    }

    for (int i = 0; i < 20; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("\n=== Final State ===\n");
    print_info();

    printf("Expected: 10 sold, 0 remaining\n");
    printf("Actual: %d sold, %d remaining\n", concert.total_sold, concert.avai
lable_seats);

    if (concert.total_sold > 10) {
        printf("\nOversold by %d tickets!\n", concert.total_sold - 10);
    }

    return 0;
}
```

5-1. Problem

The `book_ticket()` function suffers a race condition because the check `if (concert.available_seats > 0)` and the subsequent updates `concert.available_seats--` and `concert.total_sold++` are not performed atomically. More specifically, twenty threads concurrently read and write the `available_seats` and `total_sold` fields of the `concert` structure without any mutex or other synchronization protecting those

operations. As a result, multiple threads can pass the `available_seats == 1` test at the same time and then — during the `usleep(500000)` delay that yields the CPU and allows context switches — perform the seat-decrement and sale-increment steps concurrently. The practical outcome is overbooking: even if only 10 seats actually exist, more than 10 tickets can be sold.

This is a classic TOCTOU (Time-Of-Check to Time-Of-Use) vulnerability that can cause severe business logic errors in systems like financial transactions or inventory management.

The program's observed output demonstrates this overbooking behavior.

```
=== Final State ===
Available Seats: -10
Total Sold: 20
Expected: 10 sold, 0 remaining
Actual: 20 sold, -10 remaining

Oversold by 10 tickets!
```

5-2. Patched code

```
// [patched secure code]
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

typedef struct {
    int ticket_id;
    int available_seats;
    int total_sold;
    char event_name[50];
    pthread_mutex_t lock;  // add: mutex lock
} TicketSystem;

TicketSystem concert = {
    .ticket_id = 1,
```

```
    .available_seats = 10,
    .total_sold = 0,
    .event_name = "SECURITY Concert",
    .lock = PTHREAD_MUTEX_INITIALIZER  // add: mutex initialization
};

typedef struct {
    int user_id;
    char name[20];
} User;


void* book_ticket(void* arg) {
    User *user = (User*)arg;

    printf("[%s] Checking availability...\n", user->name);

    // add: protect critical section
    pthread_mutex_lock(&concert.lock);

    if (concert.available_seats > 0) {
        printf("[%s] Found available seat! Processing payment...\n", user->na
me);
        usleep(500000);

        concert.available_seats--;
        concert.total_sold++;

        printf("[%s] ✓ Booking SUCCESS! Remaining seats: %d\n", user->nam
e, concert.available_seats);
    } else {
        printf("[%s] × SOLD OUT!\n", user->name);
    }

    // add: end critical section protection
    pthread_mutex_unlock(&concert.lock);

    return NULL;
```

```
}

void print_info() {
    // add: protect read operations with mutex
    pthread_mutex_lock(&concert.lock);
    printf("Available Seats: %d\n", concert.available_seats);
    printf("Total Sold: %d\n", concert.total_sold);
    pthread_mutex_unlock(&concert.lock);
}

int main() {
    pthread_t threads[20];
    User users[20];

    printf("=== Initial State ===\n");
    print_info();

    printf("Starting booking...\n\n");

    for (int i = 0; i < 20; i++) {
        users[i].user_id = i + 1;
        snprintf(users[i].name, sizeof(users[i].name), "User%02d", i + 1);
        pthread_create(&threads[i], NULL, book_ticket, &users[i]);
    }

    for (int i = 0; i < 20; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("\n=== Final State ===\n");
    print_info();

    printf("Expected: 10 sold, 0 remaining\n");
    printf("Actual: %d sold, %d remaining\n", concert.total_sold, concert.avai
lable_seats);

    if (concert.total_sold > 10) {
        printf("\nOversold by %d tickets!\n", concert.total_sold - 10);
```

```
    }

    // add: clear mutex
    pthread_mutex_destroy(&concert.lock);

    return 0;
}
```

To fix the race-condition vulnerability I implemented a **pthread mutex–based synchronization** mechanism. I added a `pthread_mutex_t lock` field to the `TicketSystem` structure and initialized it (e.g. with `PTHREAD_MUTEX_INITIALIZER`) so there is a mutex protecting the shared `concert` data.

The core patch wraps the `book_ticket()` function's critical section with the mutex. I surround the seat-check, payment handling, seat decrement, and `total_sold` increment with:

```
pthread_mutex_lock(&concert.lock);
/* check available_seats, process payment, available_seats--, total_sold++
*/
pthread_mutex_unlock(&concert.lock);
```

This ensures only one thread can execute that sequence at a time, eliminating the TOCTOU (time-of-check/time-of-use) window. I also protected `print_info()` reads with the same mutex to keep read consistency, and I call `pthread_mutex_destroy()` at program shutdown to free resources. With this change, even if 20 threads attempt booking concurrently, exactly 10 tickets are sold and overbooking no longer occurs.

```
potato@potatoui-MacBookAir vulnerable_c % cd "/Users/potato/Desktop/K
U/25-2/정보보호/assignment2_secure_coding_and_design/vulnerable_c/" &&
gcc vulne
rable5.c -o vulnerable5 && "/Users/potato/Desktop/KU/25-2/정보보호/assig
nment2_secure_coding_and_design/vulnerable_c/"vulnerable5
=== Initial State ===
Available Seats: 10
Total Sold: 0
Starting booking...
```

```
[User01] Checking availability...
[User01] Found available seat! Processing payment...
[User02] Checking availability...
[User03] Checking availability...
[User04] Checking availability...
[User05] Checking availability...
[User06] Checking availability...
[User07] Checking availability...
[User08] Checking availability...
[User09] Checking availability...
[User10] Checking availability...
[User11] Checking availability...
[User12] Checking availability...
[User13] Checking availability...
[User14] Checking availability...
[User15] Checking availability...
[User16] Checking availability...
[User17] Checking availability...
[User18] Checking availability...
[User19] Checking availability...
[User20] Checking availability...
[User01] ✓ Booking SUCCESS! Remaining seats: 9
[User02] Found available seat! Processing payment...
[User02] ✓ Booking SUCCESS! Remaining seats: 8
[User03] Found available seat! Processing payment...
[User03] ✓ Booking SUCCESS! Remaining seats: 7
[User04] Found available seat! Processing payment...
[User04] ✓ Booking SUCCESS! Remaining seats: 6
[User05] Found available seat! Processing payment...
[User05] ✓ Booking SUCCESS! Remaining seats: 5
[User06] Found available seat! Processing payment...
[User06] ✓ Booking SUCCESS! Remaining seats: 4
[User07] Found available seat! Processing payment...
[User07] ✓ Booking SUCCESS! Remaining seats: 3
[User08] Found available seat! Processing payment...
[User08] ✓ Booking SUCCESS! Remaining seats: 2
[User09] Found available seat! Processing payment...
[User09] ✓ Booking SUCCESS! Remaining seats: 1
```

```
[User10] Found available seat! Processing payment...
[User10] ✓ Booking SUCCESS! Remaining seats: 0
[User11] × SOLD OUT!
[User12] × SOLD OUT!
[User13] × SOLD OUT!
[User14] × SOLD OUT!
[User15] × SOLD OUT!
[User16] × SOLD OUT!
[User17] × SOLD OUT!
[User18] × SOLD OUT!
[User19] × SOLD OUT!
[User20] × SOLD OUT!

=== Final State ===
Available Seats: 0
Total Sold: 10
Expected: 10 sold, 0 remaining
Actual: 10 sold, 0 remaining
```