# Secure Coding Writeup

# Luke Matsui | COSE354: Information Security | October 30, 2025

## Introduction

During my security analysis of both C and Python vulnerable code, I discovered multiple vulnerabilities that we have learned in class. This writeup documents each vulnerability found, the reasoning behind the fixes, and solutions that I implemented. Covered issues include buffer overflows, integer overflows, format string bugs, race conditions, SQL injection, SSRF, command injection, path traversal, and insecure configurations.

## Vulnerable C code

### vulnerable1.c - Buffer Overflow and String Comparison Issues

#### Buffer Overflow Vulnerability

**Vulnerable Code:**

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

**Safe Code:**

|  |
| --- |
| printf("Input : "); if (fgets(buffer, sizeof(buffer), stdin) == NULL) {  printf("Input Error\n");  exit(1); } |

Patch Explanation: I replaced the dangerous gets() function with fgets() because gets() has no bounds checking and would happily write beyond the buffer boundaries. Fgets() also includes size checking that helps prevent overflow.

#### Unsafe String Comparison

**Vulnerable Code:**

|  |
| --- |
| if (strcmp(seceret, "COSE354") == 0){ |

**Safe Code:**

|  |
| --- |
| if (strncmp(seceret, "COSE354", sizeof(seceret)) == 0){ |

Patch Explanation: At first I thought strcmp() was fine, but then realized it could read beyond buffer bounds if strings were not properly terminated. I then switched it to safe function strncmp() with a length limit.

### vulnerable2.c — Multiple Buffer Overflow Vulnerabilities

#### Bounds Checking in Input Loop

**Vulnerable Code:**

|  |
| --- |
| while((c = getchar()) != '\n' && c != EOF) {  buffer[i++] = c; } |

**Safe Code:**

|  |
| --- |
| while(i < sizeof(buffer) - 1 && (c = getchar()) != '\n' && c != EOF) {  buffer[i++] = c; } |

Patch Explanation: This while loop was reading characters without checking buffer limits. I added a simple bounds check i < sizeof(buffer) - 1 to stop writing when the buffer fills up.

#### Unsafe String Formatting

**Vulnerable Code:**

|  |
| --- |
| sprintf(buffer, "User input: %s", input); |

**Safe Code:**

|  |
| --- |
| snprintf(buffer, sizeof(buffer), "User input: %s", input); |

Patch Explanation: The sprintf() function was a clear risk since it doesn't check destination size. I replaced it with snprintf() which includes buffer size limits to prevent overflow.

#### Unsafe Input and String Copying

**Vulnerable Code:**

|  |
| --- |
| scanf("%s", input); strcpy(buffer, input); |

**Safe Code:**

|  |
| --- |
| scanf("%9s", input); strncpy(buffer, input, sizeof(buffer) - 1); buffer[sizeof(buffer) - 1] = '\0'; |

Patch Explanation: This had multiple issues - scanf() without limits and strcpy() without bounds. I added field width to scanf(). Initially I thought strncpy() would be fine, but it does not guarantee a trailing '\0' if truncate, so that necessitated explicit null-termination.

### vulnerable3.c — Integer Overflow/Underflow Vulnerabilities

#### Unsafe Unsigned Addition

**Vulnerable Code:**

|  |
| --- |
| a->balance += amount; |

**Safe Code:**

|  |
| --- |
| if (a->balance > UINT32\_MAX - amount) {  return -2; // Overflow } a->balance += amount; |

Patch Explanation: Checked for overflow before addition to prevent wraparound.

#### Unsafe Unsigned Subtraction

**Vulnerable Code:**

|  |
| --- |
| uint32\_t new\_balance = a->balance - amount; if (new\_balance < 0) return -2; |

**Safe Code:**

|  |
| --- |
| if (a->balance < (uint32\_t)amount) {  return -2; // Underflow } uint32\_t new\_balance = a->balance - amount; |

Patch Explanation: Checked for underflow before subtraction to prevent wraparound.

#### Unsafe Signed to Unsigned Conversion

**Vulnerable Code:**

|  |
| --- |
| uint32\_t new\_balance = a->balance + delta; |

**Safe Code:**

|  |
| --- |
| if (delta >= 0) {  if (a->balance > UINT32\_MAX - (uint32\_t)delta) {  return -1; // Overflow  } } else {  if (a->balance < (uint32\_t)(-delta)) {  return -2; // Underflow  } } uint32\_t new\_balance = a->balance + delta; |

Patch Explanation: Checked both overflow and underflow.

### vulnerable4.c — Format String Vulnerabilities

#### Unsafe File Output

**Vulnerable Code:**

|  |
| --- |
| fprintf(f, input); |

**Safe Code:**

|  |
| --- |
| fprintf(f, "%s", input); |

Patch Explanation: Prevent format string injection by using %s and passing user data as an arg.

#### Unsafe System Logging

**Vulnerable Code:**

|  |
| --- |
| syslog(LOG\_INFO, input); |

**Safe Code:**

|  |
| --- |
| syslog(LOG\_INFO, "%s", input); |

Patch Explanation: Same issue as the file output - user input was being treated as a format string, and made the same fixes.

### vulnerable5.c — Race Condition (TOCTOU)

#### TOCTOU

**Vulnerable Code:**

|  |
| --- |
| if (concert.available\_seats > 0) {  usleep(500000);   concert.available\_seats--;  concert.total\_sold++; } |

**Safe Code:**

|  |
| --- |
| pthread\_mutex\_lock(&concert.mutex); if (concert.available\_seats > 0) {  usleep(500000);  concert.available\_seats--;  concert.total\_sold++; } pthread\_mutex\_unlock(&concert.mutex); |

Patch Explanation: Added a mutex from lecture – took a long time to figure out how to do!!

## Python / Flask Web Application Vulnerabilities

### VULNERABILITY 1: SQL Injection in Login

**Vulnerable Code:**

|  |
| --- |
| @app.route('/Login', methods=['GET', 'POST']) def login():  # ...   query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'"  cur.execute(query) |

**Safe Code:**

|  |
| --- |
| @app.route('/Login', methods=['GET', 'POST']) def login():  # ...  query = "SELECT \* FROM users WHERE username = ? AND password = ?"  cur.execute(query, (username, password)) |

Patch Explanation: The vulnerable code used string concatenation to build SQL queries, allowing SQL injection, so I switched to parameterized queries which properly separate SQL commands from user data, preventing injection.

### VULNERABILITY 2: Server Side Request Forgery (SSRF)

**Vulnerable Code:**

|  |
| --- |
| @app.route('/Fetch', methods=['GET', 'POST']) def fetch():  # ...  r = requests.get(url, timeout=10)  content = r.text[:4096] |

**Safe Code (key excerpts):**

|  |
| --- |
| def host\_resolves\_to\_disallowed\_ip(hostname: str) -> bool:  infos = socket.getaddrinfo(hostname, None, proto=socket.IPPROTO\_TCP)  for family, \_, \_, \_, sockaddr in infos:  ip\_str = sockaddr[0]  try:  ip = ipaddress.ip\_address(ip\_str)  if (ip.is\_private or ip.is\_loopback or ip.is\_link\_local or   ip.is\_multicast or ip.is\_reserved):  return True  except ValueError:  continue  return False  @app.route('/Fetch', methods=['GET', 'POST']) def fetch():  parsed = urlparse(url)  if parsed.scheme not in ('http', 'https'):  content = 'Only http and https URLs are allowed'  return render\_template('fetch.html', url=url, content=content)    hostname = parsed.hostname  if host\_resolves\_to\_disallowed\_ip(hostname):  content = 'Blocked URL (resolves to internal or disallowed IP address)'  return render\_template('fetch.html', url=url, content=content)    r = requests.get(url, timeout=10, allow\_redirects=False, stream=True)  buf, read = [], 0  for chunk in r.iter\_content(chunk\_size=1024):  read += len(chunk)  if read > MAX\_BYTES:  buf.append(chunk[: MAX\_BYTES - (read - len(chunk))])  break  buf.append(chunk)  content = b''.join(buf).decode('utf-8', errors='replace') |

Patch Explanation: The original code allowed requests to any URL without restrictions, and this required multiple layers of defense. I added URL scheme validation, implemented IP resolution checks to block internal addresses, disabled redirects that could bypass checks, and added response size limits. It took a few iterations to get all the protections in place.

### VULNERABILITY 3: Command Injection

**Vulnerable Code:**

|  |
| --- |
| @app.route('/Ping', methods=['GET', 'POST']) def ping\_host():  # ...  command = ['ping', count\_flag, '3', ip\_address]  completed = subprocess.run(command, capture\_output=True, text=True, timeout=15) |

**Safe Code (key excerpts):**

|  |
| --- |
| \_HOSTNAME\_RE = re.compile(r'^[A-Za-z0-9]([A-Za-z0-9\.-]{0,253}[A-Za-z0-9])?$')  def is\_valid\_hostname(host: str) -> bool:  return bool(\_HOSTNAME\_RE.match(host))  def is\_valid\_ip\_or\_hostname(value: str) -> bool:  try:  ipaddress.ip\_address(value)  return True  except ValueError:  return is\_valid\_hostname(value)  # Input validation checks if ip\_address.startswith('-'):  return render\_template('ping.html', ip=ip\_address, result='Invalid host (starts with "-")')  if any(ch.isspace() for ch in ip\_address):  return render\_template('ping.html', ip=ip\_address, result='Invalid host (contains whitespace)')  if not is\_valid\_ip\_or\_hostname(ip\_address):  return render\_template('ping.html', ip=ip\_address, result='Invalid hostname or IP')  command = ['ping', count\_flag, '3', ip\_address] completed = subprocess.run(command, capture\_output=True, text=True, timeout=15) |

Patch Explanation: The vulnerable code passed user input directly to system commands. I decided to implement regex validation for hostnames, proper IP validation, and blocked dangerous characters like hyphens and whitespace that could enable command injection.

### VULNERABILITY 4: Path Traversal

**Vulnerable Code:**

|  |
| --- |
| @app.route('/ViewFile') def view\_file():  # ...  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]  with open(file\_path, 'r', encoding='utf-8', errors='ignore') as f:  content = f.read() |

**Safe Code:**

|  |
| --- |
| @app.route('/ViewFile') def view\_file():  # ...  decoded = unquote(raw)  decoded = decoded.split('\x00', 1)[0]  candidate = Path(LOG\_DIR) / (decoded + '.log')    base\_resolved = Path(LOG\_DIR).resolve(strict=False)  candidate\_resolved = candidate.resolve(strict=False)    try:  candidate\_resolved.relative\_to(base\_resolved)  except ValueError:  error = 'Access denied'  return render\_template('view.html', filename=raw, content=content, error=error)    with candidate\_resolved.open('r', encoding='utf-8', errors='ignore') as f:  content = f.read() |

Patch Explanation: he original "fix" of replacing ../ once was completely inadequate. I replaced it with proper path canonicalization using pathlib, and implemented directory containment checks using resolve() and relative\_to() methods to ensure files remain within the authorized directory.

### VULNERABILITY 5: Insecure Configuration

**Vulnerable Code:**

|  |
| --- |
| app = Flask(\_\_name\_\_) app.secret\_key = 'very\_very\_secret' # ... app.run(host='0.0.0.0', port=5000, debug=True) |

**Safe Code:**

|  |
| --- |
| app = Flask(\_\_name\_\_) \_secret = os.getenv('FLASK\_SECRET\_KEY') or os.getenv('SECRET\_KEY') if not \_secret:  \_secret = secrets.token\_urlsafe(32)  logging.warning("No FLASK\_SECRET\_KEY found, set FLASK\_SECRET\_KEY in your environment") app.secret\_key = \_secret  app.config.update(  SESSION\_COOKIE\_HTTPONLY=True,  SESSION\_COOKIE\_SAMESITE='Lax',  SESSION\_COOKIE\_SECURE=os.getenv('FLASK\_COOKIE\_SECURE', '0') in ('1', 'true', 'True'), )  DEBUG\_MODE = os.getenv('FLASK\_DEBUG', '0').lower() in ('1','true','yes') app.run(host='0.0.0.0', port=int(os.getenv('PORT', 5000)), debug=DEBUG\_MODE) |

Patch Explanation: I moved everything to environment variables with secure fallbacks, added proper cookie security flags, and made debug mode configurable for safe deployment.