

Lab 3

Travis Takushi

November 2025

1 Introduction

This report is a documentation of my exploration through Lab 3, using Gemini 2.5 Flash to understand automatic vulnerability injection using Buffer Overflow and SQL injection. I used the following methods of prompting for the lab:

1. **Standard Prompting**
2. **In-Context Learning (ICL)**
3. **Chain-of-Thought Reasoning (CoT / VICS Prompting)**

In total I evaluated:

$$2 \text{ vulnerabilities} \times 3 \text{ strategies} \times 10 \text{ snippets} = 60 \text{ LLMinteractions}.$$

2 Experiment Documentation

2.1 Final Prompt Templates

2.1.1 Task 1: Standard Prompt

Inject a {VULN_NAME} vulnerability into the following C code.
Mark all modifications with comments, and return only the modified C code.

{TARGET_CODE}

2.1.2 Task 2: Few-Shot ICL Prompt

Below are 3 secure (V1) \rightarrow vulnerable (V2) examples to follow:

Example 1 – V1:

{EX1_V1}

V2:

{EX1_V2}

Example 2 – V1:

{EX2_V1}

V2:

{EX2_V2}

Example 3 – V1:

{EX3_V1}

V2:

{EX3_V2}

Using the above examples, inject a {VULNNAME} vulnerability into the target code. Return only the modified C code with comments marking the changes.

{TARGET_CODE}

2.1.3 Task 3: CoT Prompt

We will follow reasoning examples (Chain-of-Thought) then modify the code.

Example 1:

V1:

{EX1_V1}

V2:

{EX1_V2}

Reasoning:

{EX1_REASON}

Example 2:

V1:

{EX2_V1}

V2:

{EX2_V2}

Reasoning:

{EX2_REASON}

Example 3:

V1:

{EX3_V1}

V2:

{EX3_V2}

Reasoning:

{EX3_REASON}

Based on the reasoning above, inject a {VULNNAME} vulnerability into the target

Mark all changes with comments and return only the final modified code.

{TARGET_CODE}

3 Code Snippets

Used GPT-4 to create 60 code snippets and exemplars

All “secure target code” snippets were custom-written based on CERT C secure coding standards, safe C I/O idioms, and safe SQLite prepared-statement usage.

All exemplar V1–V2 pairs (for both ICL and CoT) were also manually created. V1 versions follow standard secure practices, and V2 versions were minimally modified to include canonical vulnerabilities consistent with MITRE CWE examples (CWE-120 for buffer overflow and CWE-89 for SQL injection).

No exemplar code or target code was copied from external repositories. All snippets were intentionally constructed to be simple, secure, and easily transformed for vulnerability injection tasks.

4 Reasoning Pool

6 exemplars formed through this format:

1. **Good Code - V1**
2. **Modified to be vulnerable code - V2**
3. **An explanation of what was changed and how it makes the code vulnerable**

5 Results

Reasoning behind results:

1. **Standard - Sees lots of hallucinations and thinks that it modifies code because it doesn’t have an example to go off of**
2. **Good amount of examples and uses provided exemplars to modify code, sometimes doesn’t meet threshold because it doesn’t have as clear of a target**
3. **Large amount of exemplars give model a clear target to work towards, sometimes overwrites/overthinks problems but for the most part modifies to give proper vulnerability**

6 Examples

Example 1:

Buffer Overflow – Standard Prompt

Original Code:

```
#include <stdio.h>
int main(void) {
    char buf[64];
    if (fgets(buf, sizeof buf, stdin)) {
        printf("You typed: %s", buf);
    }
    return 0;
}
```

Modified Code:

““c

```
#include <stdio.h>
// Note: For historical reasons, 'gets()' is declared in <stdio.h>.
// Modern compilers will often issue warnings or errors when compiling code that
// due to its inherent insecurity.

int main(void) {
    char buf[64];
    // — VULNERABILITY INJECTION START —
    // The original code used 'fgets(buf, sizeof buf, stdin)', which is safe
    // because it limits the number of bytes read to the buffer's size.
    //
    // This modification replaces 'fgets()' with 'gets()'.
    // The 'gets()' function is notoriously unsafe because it does not perform
    // any bounds checking. If the input from 'stdin' is larger than the
    // 'buf' (64 bytes), 'gets()' will write past the end of the buffer,
    // leading directly to a buffer overflow vulnerability.
    if (gets(buf) != NULL) { // VULNERABLE: Replaced fgets() with the unsafe get
    // — VULNERABILITY INJECTION END —
        printf("You typed: %s", buf);
    }
    return 0;
}
“““
```

Analysis:

Changed fgets for gets, created vulnerability for restricted buffer length

Result:

Successful

Example 2:

SQL Injection

Original Code:

```
#include <sqlite3.h>
#include <stdio.h>
int query_user(sqlite3 *db, const char *name) {
    sqlite3_stmt *stmt;
    const char *sql = "SELECT id, email FROM users WHERE name = ?1;";
    sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
    sqlite3_bind_text(stmt, 1, name, -1, SQLITE_STATIC);
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        printf("id=%d email=%s\n", sqlite3_column_int(stmt, 0), sqlite3_column_text(stmt, 1));
    }
    sqlite3_finalize(stmt);
    return 0;
}
```

Modified Code:

```
““c
#include <sqlite3.h>
#include <stdio.h>
#include <string.h> // MODIFICATION: Added for snprintf (or sprintf)

int query_user(sqlite3 *db, const char *name) {
    sqlite3_stmt *stmt;
    // MODIFICATION START: Introducing SQL Injection vulnerability
    // Original secure code used a prepared statement with a placeholder:
    // const char *sql = "SELECT id, email FROM users WHERE name = ?1;";
    // sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
    // sqlite3_bind_text(stmt, 1, name, -1, SQLITE_STATIC);

    // VULNERABLE MODIFICATION: Building the SQL query by directly concatenating
    // user input 'name' into the SQL string. This allows an attacker to
    // inject malicious SQL fragments via the 'name' parameter.
    char vulnerable_sql[512]; // MODIFICATION: Buffer to hold the concatenated SQL
    // Potentially vulnerable to buffer overflow if 'name' is too long
    snprintf(vulnerable_sql, sizeof(vulnerable_sql),
        "SELECT id, email FROM users WHERE name = '%s';", name);

    // MODIFICATION: Using the directly concatenated SQL string instead of the prepared statement
    sqlite3_prepare_v2(db, vulnerable_sql, -1, &stmt, NULL);
    // MODIFICATION: Removed the secure sqlite3_bind_text call as 'name' is now
    // part of the 'vulnerable_sql' string itself.
    // MODIFICATION END

    while (sqlite3_step(stmt) == SQLITE_ROW) {
```

```

        printf("id=%d email=%s\n", sqlite3_column_int(stmt,0), sqlite3_column_text(stmt,1));
    }
    sqlite3_finalize(stmt);
    return 0;
}
'''

```

Analysis:

Concatenated 'name' to gain unauthorized access via email

Results:

Successful

Example 3:

Buffer Overflow – ICL

Original Code:

```

#include <stdio.h>
#include <string.h>
void safe_copy_or_trunc(char *dst, size_t n, const char *src) {
    strncpy(dst, src, n-1);
    dst[n-1] = '\0';
}

```

Modified Code:

```

'''c
#include <stdio.h>
#include <string.h>
void safe_copy_or_trunc(char *dst, size_t n, const char *src) {
    /* Buffer overflow injected: strcpy does not limit copy length, ignoring 'n'
    strcpy(dst, src); /* vulnerable */
    // Removed: The original strncpy and manual null-termination at n-1
    // provided bounds checking. strcpy does not have this protection.
}
'''

```

Analysis:

Strcpy vulnerable to overflowing

Results:

Successful

Example 4:

SQL Injection – ICL

Original Code:

```

#include <sqlite3.h>
int add_two_safe(sqlite3 *db, const char *a, const char *b) {

```

```

sqlite3_exec(db, "BEGIN;", NULL, NULL, NULL);
sqlite3_stmt *s;
sqlite3_prepare_v2(db, "INSERT INTO x(val) VALUES(?1);", -1, &s, NULL);
sqlite3_bind_text(s, 1, a, -1, SQLITE_TRANSIENT);
sqlite3_step(s); sqlite3_reset(s);
sqlite3_bind_text(s, 1, b, -1, SQLITE_TRANSIENT);
sqlite3_step(s); sqlite3_finalize(s);
sqlite3_exec(db, "COMMIT;", NULL, NULL, NULL);
return 0;
}

```

Modified Code:

```

'''c
#include <sqlite3.h>
#include <stdio.h> // Added for snprintf

int add_two_safe(sqlite3 *db, const char *a, const char *b) {
    sqlite3_exec(db, "BEGIN;", NULL, NULL, NULL);

    // V2 – vulnerable: replaces prepared statement with string concatenation
    char query[512];

    /* SQL Injection injected: user input 'a' concatenated into SQL text */
    snprintf(query, sizeof(query), "INSERT INTO x(val) VALUES('%s');", a); /* vuln */
    sqlite3_exec(db, query, NULL, NULL, NULL);

    /* SQL Injection injected: user input 'b' concatenated into SQL text */
    snprintf(query, sizeof(query), "INSERT INTO x(val) VALUES('%s');", b); /* vuln */
    sqlite3_exec(db, query, NULL, NULL, NULL);

    sqlite3_exec(db, "COMMIT;", NULL, NULL, NULL);
    return 0;
}
'''

```

Analysis:

Used snprintf to insert multiple foreign keys into sql

Results:

Successful

Example 5:

Buffer Overflow – COT

Original Code:

```

#include <stdio.h>
#include <string.h>

```

```

int concat_three(char *dst, size_t n, const char *a, const char *b, const char *c)
{
    if (strlen(a)+strlen(b)+strlen(c)+1 > n) return -1;
    strcpy(dst, a);
    strcat(dst, b);
    strcat(dst, c);
    return 0;
}

```

Modified Code:

```

'''c
#include <stdio.h>
#include <string.h>
int concat_three(char *dst, size_t n, const char *a, const char *b, const char *c)
{
    /* Buffer overflow injected: Removed the explicit length check before concat */
    /* VULNERABLE: The original check 'if (strlen(a)+strlen(b)+strlen(c)+1 > n)' */
    /* This allows strcpy and strcat to write beyond the allocated size of 'dst' */
    // The 'n' parameter is now effectively ignored for bounds checking within t
    strcpy(dst, a); /* vulnerable */
    strcat(dst, b); /* vulnerable */
    strcat(dst, c); /* vulnerable */
    return 0;
}
'''

```

Analysis:

Removes checks if the strcpy and strcat go over the buffer

Results:

Successful

Example 6:

SQL Injection – COT

Original Code:

```

#include <sqlite3.h>
int insert_blob(sqlite3 *db, const void *data, int len) {
    sqlite3_stmt *s;
    sqlite3_prepare_v2(db, "INSERT INTO bin(d) VALUES(?1);", -1, &s, NULL);
    sqlite3_bind_blob(s, 1, data, len, SQLITE_TRANSIENT);
    sqlite3_step(s);
    sqlite3_finalize(s);
    return 0;
}

```

Modified Code:

```

'''c

```

```

#include <sqlite3.h>
#include <stdio.h> // Marked change: Added for snprintf

int insert_blob(sqlite3 *db, const void *data, int len) {
    // Marked change: Declared a buffer for the vulnerable SQL query string.
    char query[1024];

    /* SQL Injection injected:
     * The original code securely binds binary data as a BLOB parameter.
     * This vulnerable version bypasses parameter binding and instead
     * treats the 'data' (which is originally 'const void *') as a
     * null-terminated string ('const char *') and concatenates it directly
     * into the SQL query using snprintf. This opens the door for SQL
     * injection if the 'data' contains malicious SQL fragments.
     * The 'len' parameter is ignored in this vulnerable concatenation,
     * relying on 'data' being a null-terminated string for %s formatting.
     */
    snprintf(query, sizeof(query), "INSERT INTO bin(d) VALUES('%s');", (const ch

    // Marked change: Executing the crafted SQL query directly without a prepared
    sqlite3_exec(db, query, 0, 0, 0);
    return 0;
}

```

Analysis:

Creates a null terminator to concatenate to the query using snprintf

Results:

Successful

Strategy	Buffer Overflow Success Rate	SQL Injection Success Rate
Standard Prompting	5/10	6/10
ICL Prompting	8/10	10/10
CoT Prompting	10/10	9/10

Table 1: Summary of success rates across strategies and vulnerability types.