

L3: Vulnerability Assessment and Secure Coding

Software Analysis

IIT Guwahati · August 4, 2026

Outline

- A taxonomy of software vulnerabilities
- Memory safety
 - Null dereference, use-after-free, buffer overflow, leaks
- Arithmetic
 - Integer overflow, division by zero
- Tainted input
 - Information flow, injection, format strings, SQL
- Side channels

Common Types of Software Vulnerabilities

Memory safety

- null pointer dereference
- dangling ptrs /
use-after-free
- buffer overflows
- memory leaks

Arithmetic

- integer overflows
- division by zero

Tainted inputs

- tainted information flow
- code injection
- format string
- SQL injection

Side channels

- timing leakage
- / timing attacks

Memory Safety

Null dereference, use-after-free, buffer overflow, leaks

Null Pointer Dereference

Occurs when an application dereferences a pointer it expects to be valid, but which is **NULL** — typically causing a **crash or exit**.

Real case: CVE-2020-1971 — OpenSSL.^[1]

- `GENERAL_NAME_cmp()` mishandled X.509 `EDIPartyName` → NULL deref → crash.
- Reachable *before* signatures are verified, via automatic CRL download.
- Affected *all* OpenSSL 1.1.1 and 1.0.2 versions.

Usually **denial of service**, not code execution — the page at address 0 is unmapped, so it crashes rather than running attacker code. On some embedded systems with no MMU, address 0 *is* mapped — and then it is much worse.

[1] NVD: CVE-2020-1971 · CWE-476

Null Pointer Dereference: A Vulnerable Example

```
typedef struct Student {
    int id;
    char *name;
} Student;

Student students[10] = { ... };

Student *findStudentRecord(int sID) {
    for (int i = 0; i < 10; i++)
        if (students[i].id == sID) return &students[i];
    return nullptr;           // not found
}

int main(int argc, char **argv) {
    int stuID;
    scanf("%d", &stuID);
    Student *student = findStudentRecord(stuID);
    printf("%s\n", student->name); // student may be NULL!
}
```

Input any ID not in the table → **crash**. Fix: `assert(student != nullptr);` before use.

Dangling References and Use-After-Free

A **dangling reference** is a reference that does not resolve to a valid memory object — e.g. a pointer to memory that has already been freed.

Real case: CVE-2019-5786 — Google Chrome FileReader.^[1]

- Use-after-free in the FileReader API, reachable from *any web page*.
- **Exploited in the wild** as a zero-day (Feb 2019); gave code execution in the renderer.
- Chained with a Windows win32k.sys bug to **escape the sandbox** and own the machine.

Unlike a null deref, freed memory is often *still mapped* — so the attacker can **reallocate it with data they control**. That is why UAF means code execution, not just a crash.

[1] NVD: CVE-2019-5786 · CWE-416 · Tenable analysis

Use-After-Free: A Vulnerable Example

```
char *ptr = (char *) malloc(SIZE);
...
if (err) {
    abrt = 1;
    free(ptr);           // freed here
}
...
if (abrt) {
    logError("Operation aborted before commit", ptr); // used here!
}
```

Do we have a bug? If so, how do we fix it?

ptr is freed on the error path, then read on the very next error path.

The two ifs are far apart in real code — which is exactly why nobody notices.

Use-After-Free: The Fix

```
char *ptr = (char *) malloc(SIZE);
...
if (err) {
    abrt = 1;
    logError("Operation aborted before commit", ptr); // use FIRST
    free(ptr);                                         // then free
    ptr = nullptr;                                    // then poison
}
...
```

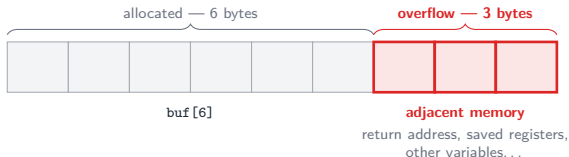
- Use the pointer **before** freeing it.
- Set it to **nullptr** immediately after freeing.

Why poison? A later use now **crashes predictably** (null deref) instead of **silently corrupting** attacker-controlled memory.

We traded a **code-execution** bug for a **denial-of-service** bug. That is a good trade.

Buffer Overflow

Occurs when data written **exceeds the capacity** of a buffer, overwriting **adjacent memory**.



Real case: CVE-2021-3156 —sudo.^[1]

- Heap overflow via mishandled backslashes; hidden in plain sight since **2011**.
- **Any unprivileged local user** → **root**, in the *default* config. Even *nobody*.
- Confirmed on Ubuntu 20.04, Debian 10, Fedora 33.

[1] Qualys, CVE-2021-3156 · CWE-787 · public PoCs exist — worawit/CVE-2021-3156

Buffer Overflow: A Vulnerable Example

```
void bufferRead() {  
    int n = 0;  
    int ret = scanf("%d", &n);  
    if (ret != 1 || n > 100) { return; }    // n <= 100 ... but n >= 0?  
  
    char *p = (char *) malloc(n);  
    int y = n;  
    if (p == NULL) return;  
    p[y] = 'a';                            // writes p[n] -- off by one!  
    free(p);  
}
```

Do we have a bug? Where?

malloc(n) gives valid indices p[0] ... p[n-1]. Writing p[y] where y == n is **one byte past the end**.

Worse: n is never checked against 0 — a **negative** n passes the guard.

Buffer Overflow: An Assertion-Guarded Example

```
void bufferRead() {  
    int n = 0;  
    int ret = scanf("%d", &n);  
    if (ret != 1 || n > 100) { return; }  
  
    char *p = (char *) malloc(n);  
    int y = n;  
    if (p == NULL) return;  
    assert(y < n);           // the safety property  
    p[y] = 'a';  
    free(p);  
}
```

The assertion states the property: *the index must be within bounds.*

Memory Leaks

A memory leak occurs when dynamically allocated memory is **never freed** along some program execution path.

Real case: CVE-2021-3764 — Linux kernel CCP driver.^[1]

- Leak in `ccp_run_aes_gcm_cmd()` in the crypto co-processor driver.
- A local user repeatedly invokes the operation; each call leaks a little memory.
- Eventually **exhausts system memory** → denial of service.

[1] NVD: CVE-2021-3764 · CWE-401

Memory Leaks: A Vulnerable Example

```
List *create_list(unsigned num) {
    List *list = new List();
    list->head = new Node{0, nullptr};
    Node *current = list->head;
    for (unsigned i = 0; i < num - 1; i++) {
        current = new Node();           // allocated, never linked in!
        current = current->next;        // current->next is garbage
    }
    return list;
}

void free_list(List *l) {
    Node *current = l->head;
    while (current != nullptr) {
        Node *next = current->next;
        delete current;
        current = next;
    }
    // the List object itself is never deleted!
}
```

Three separate bugs. Can you find them all?

Memory Leaks: The Fix

```
List *create_list(unsigned num) {  
    if (num == 0) return nullptr;    // 3. guard the underflow  
    List *list = new List();  
    list->head = new Node{0, nullptr};  
    Node *current = list->head;  
    for (unsigned i = 0; i < num - 1; i++) {  
        current->next = new Node{(int)i, nullptr};    // 1. link it in  
        current = current->next;  
    }  
    return list;  
}  
  
void free_list(List *l) {  
    Node *current = l->head;  
    while (current != nullptr) {  
        Node *next = current->next;  
        delete current;  
        current = next;  
    }  
    delete l;    // 2. free the List too  
}  
  
int main() {  
    List *l = create_list(10);  
    free_list(l);  
    l = nullptr;    // avoid misuse later  
}
```

num is unsigned: num-1 underflows to 4,294,967,295 when num == 0

Arithmetic

Integer overflow and division by zero

Integer Overflow

An integer overflow occurs when an arithmetic operation produces a value **outside the representable range** — above the maximum or below the minimum.

Maximum representable unsigned value by register width:

4 bits $2^4 - 1 = 15$

8 bits $2^8 - 1 = 255$

16 bits $2^{16} - 1 = 65,535$

32 bits $2^{32} - 1 = 4,294,967,295$

64 bits $2^{64} - 1 = 18,446,744,073,709,551,615$

128 bits $2^{128} - 1 = 340,282,366,920,938,463,463,374,607,431,768,211,455$

Signed vs. Unsigned: Two Very Different Failures

Signed overflow

Undefined behaviour.

The compiler
may assume it
never happens
— and optimise
on that assumption.

Anything may happen.

Unsigned overflow

Wraparound

— well defined.

Arithmetic is reduced
 $(a+b)$ modulo 2^n ,
where n is the
number of value bits.

Defined — but
rarely intended.

$\text{INT_MIN} = -2,147,483,648$ $\text{INT_MAX} = +2,147,483,647$ $\text{UINT_MAX} = 4,294,967,295$

$\text{UINT_MAX} + 1 = 0$ $\text{UINT_MAX} + 2 = 1$ $\text{UINT_MAX} + 3 = 2$

Signed Overflow

Incorrect way to detect overflow:

```
signed int x = ...;
if (x > x + 1) {           // "if adding 1 made it smaller, we overflowed"
    // handle potential overflow
}
```

$x + 1$ overflows when $x == \text{INT_MAX}$ — **undefined**. The compiler reasons: “*signed overflow cannot happen, so $x > x+1$ is always false*” and **deletes the check entirely**.

Correct way:

```
signed int x = ...;
if (x > INT_MAX - 1) {     // test BEFORE overflowing
    // handle potential overflow
}
```

Never let the overflow happen and then look for it. **Test before you compute.**

Signed Overflow Example

```
signed int i = 1;
while (i > 0) {
    i *= 2;          // be careful using signed integers as loop bounds!
}
```

What you expect: `i` doubles, eventually overflows to a negative value, loop exits.

What actually happens under **-O3** with **GCC 4.7.2** (Debian 4.7.2-5):^[1]

The compiler reasons that a positive signed `int` doubled can never become negative *without* undefined behaviour — so `i > 0` must always be true. It performs branch simplification and emits an **infinite loop**.

The behaviour was undefined, so the compiler was **entirely within its rights**.

Mitigations: `-fwrapv`, `-fno-strict-overflow`, or `-fsanitize=undefined` to catch it at runtime.

[1] GCC optimizations based on integer overflow · cf. Ian Lance Taylor (GCC), *Signed Overflow* · Sui, COMP6131, UNSW

Integer Overflow: A Real Vulnerable Example

```
int nresp = packet_get_int();           // attacker-controlled!
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

Do we have a bug? Where?

CVE-2002-0639 — OpenSSH 2.9.9–3.3, challenge-response authentication.^[1]

- If $nresp \geq 1073741824$ (2^{30}) and `sizeof(char*)` is 4, then $nresp * 4$ **overflows** 2^{32} and wraps to a **small number**.
- `xmalloc()` therefore allocates a *tiny* buffer — but the loop still runs `nresp` times.
- Result: **heap buffer overflow** → **remote root**, pre-authentication.

[1] NVD: CVE-2002-0639 · OpenSSH advisory · fixed in OpenSSH 3.4 · Sui, COMP6131, UNSW

Integer Overflow: Assertion-Guarded

Make sure the size received is under a user budget:

```
int nresp = packet_get_int();
assert(nresp <= userDefinedSize);
if (nresp > 0) { response = xmalloc(nresp * sizeof(char*)); ... }
```

Better — make sure the *allocation itself* is under budget:

```
int nresp = packet_get_int();

if (nresp > 0) {
    assert(nresp <= userDefinedSize / sizeof(char*));
    response = xmalloc(nresp * sizeof(char *));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

Note the **division**: it bounds `nresp` *without ever performing* `nresp * sizeof(char*)`.
Same rule as before — **test before you compute**.

Division by Zero

Dividing by zero is **undefined behaviour** for integer operands (C11 Standard). It can crash the program or be miscompiled.

```
unsigned computeAverageResponseTime(unsigned totalTime,
                                     unsigned numRequests) {
    return totalTime / numRequests;    // numRequests may be 0!
}
```

Is this secure?

```
unsigned computeAverageResponseTime(unsigned totalTime,
                                     unsigned numRequests) {
    assert(numRequests > 0);
    return totalTime / numRequests;
}
```

CVE-2017-16649 — Linux kernel `usbnet_generic_cdc_bind()`: a **crafted USB device** triggers a divide-by-zero and **crashes the kernel**.^[1]

[1] NVD: CVE-2017-16649 · CWE-369 · Sui, COMP6131, UNSW

Tainted Input

Information flow, injection, format strings, SQL

Tainted Information Flow

Malicious user input may cause unexpected behaviour, information leakage, or attacks.

```
void main(int argc, char **argv) {  
    char *pMsg = packet_get_string();           // TAINTED source  
    ParseMsg((LOGIN_MSG_BODY*) pMsg);  
}  
  
void ParseMsg(LOGIN_MSG_BODY *stLoginMsgBody) {  
    for (int ulIndex = 0;  
        ulIndex < stLoginMsgBody->usLoginPwdLen; ulIndex++) {  
        // do something                          tainted loop bound!  
    }  
}
```

Do we have a bug? Where?

The loop bound comes **straight from the network**. A huge usLoginPwdLen → the loop runs essentially forever → **the target hangs**.

Fix: `assert(stLoginMsgBody->usLoginPwdLen < userDefinedSize);`

Code Injection

Exploitation caused by processing **invalid data**. The result can be disastrous — e.g. disclosing confidential information.

```
string user_id, command;  
command = "cat user_info/";  
cin >> user_id;  
system(command + user_id);    // runs it in a shell!
```

Is this secure?

system() hands the string to /bin/sh — so shell **metacharacters** are interpreted. Enter 05 && ifconfig and the shell runs cat user_info/05 **and then** ifconfig.

```
cin >> user_id;  
assert(isdigit(user_id));    // reject anything but digits  
system(command + user_id);
```

CVE-2014-6271 — “Shellshock”: Bash executed commands appended to function definitions in **environment variables**. CGI scripts made it **remote**, unauthenticated code execution. CVSS **9.8**.^[1]

[1] NVD: CVE-2014-6271 · CWE-78 · Sui, COMP6131, UNSW

Format String

Occurs when **attacker-controlled input** is passed as the *format string* argument to printf-family functions.

```
#include <stdio.h>
void main(int argc, char **argv) {
    printf("%s\n", argv[1]);    // SAFE: input is an argument
    printf(argv[1]);           // VULNERABLE: input IS the format
}
```

./example "Hello %s%s%s%s%s%s"

- First printf prints the string *literally*. **Safe**.
- Second printf interprets each %s as a **pointer to a string**, walks the stack, hits an invalid address → **crash**.

It gets worse than a crash — ./example "Hello %p %p %p %p" **leaks stack memory**:

Hello 000E133E 000E133E 0057F000 CCCCCCCC

- And %n *writes* to memory → **arbitrary code execution**.

Check CVE-2012-0809 — format string in sudo's ^[1]

[1] NVD: CVE-2012-0809 · sudo advisory · CWE-134 · Sui, COMP6131, UNSW

SQL Injection

The attacker **appends additional SQL** to a query built by the application.

```
txtUserId = getRequestString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Is this vulnerable? Why?

The intent was to select *one* user by id. Enter **105 OR 1=1**:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1
```

OR 1=1 makes the WHERE clause **always true** → the database returns **every row**.

```
txtUserId = getRequestString("UserId");  
assert(isdigit(txtUserId)); // it is supposed to be an integer  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Side Channels

When correct code still leaks

Side Channel: Timing Leakage

```
bool isSecureCmp(char *ca, char *cb, int length) {  
    for (int i = 0; i < length; i++)  
        if (ca[i] != cb[i])  
            return false;    // stops at the FIRST mismatch  
    return true;  
}
```

The function is **functionally correct** — and **leaks the answer through its runtime**. Suppose each character takes 1 ms:

'aaaaaaaaaaaaaaaa'	vs 'V1cHt2S67DADJIm9s'	1 ms (0 correct)
'baaaaaaaaaaaaaaaaa'	vs 'V1cHt2S67DADJIm9s'	1 ms (0 correct)
'Vaaaaaaaaaaaaaaaa'	vs 'V1cHt2S67DADJIm9s'	2 ms (1 correct!)
'V1aaaaaaaaaaaaaaaa'	vs 'V1cHt2S67DADJIm9s'	3 ms (2 correct!)

The timing **tells you when you got a character right**. Guess character by character: $O(n \times k)$ instead of $O(k^n)$.

CVE-2013-2061 — OpenVPN used `memcmp()` to compare HMAC tokens; the non-constant-time comparison enabled a **remote timing attack**. Fix: constant-time compare (`CRYPTO_memcmp`).^[1]

[1] NVD: CVE-2013-2061 · CWE-208 · Sui, COMP6131, UNSW