# PIB - Final Report

## Software Vulnerabilities and Secure Programming

Leon Radó

December 14, 2025


SLOVAK UNIVERSITY OF
TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS
AND INFORMATION TECHNOLOGIES

# Contents

# 1 Introduction

Modern software depends on correct memory management, even small mistakes in how memory is used can lead to serious security vulnerabilities. Low-level languages like *C/C++* give programmers direct memory control, which is powerful but also risky. These languages don't provide automatic bounds checking when working with arrays or buffers, allow access to arbitrary memory locations and the programmer has to allocate and de-allocate the memory by himself.

To better understand how these vulnerabilities occur it's important to understand how memory works. Modern systems divide memory into regions, each serving a different purpose. Vulnerabilities often occur when data in one region overwrites memory in another.

| **Memory Region** | **Purpose** | **Example** |
|---|---|---|
| Stack | Local variables, function calls | `int a = 5;` |
| Heap | Dynamically allocated memory | `malloc();` |
| Data Segment | Global/static variables | `static int counter;` |
| Code Segment | Compiled program code | |

Table 1: Memory regions with their purpose

Software memory bugs to this day remain one of the most common critical exploits of software. It's estimated that around *60–70%* of browser and kernel security bugs in *C/C++* code bases are caused by memory misuse. To minimize these risks, software must be tested using specialized analysis tools that detect memory errors early in development. Apart from that, modern operating systems provide restrictions that help prevent certain forms of attacks.

# 2 Software Vulnerabilities

In this section, several of the most common software vulnerabilities with their code examples and outcomes will be shown. Since most of these bugs lead to undefined behavior, it's not ensured that outcomes will be same in different circumstances.

## 2.1 Buffer Overflow

Buffer Overflow occurs when a program writes more data to a memory buffer than it was allocated to hold. The excess data overwrites adjacent memory, which can corrupt program state or cause crashes.

**Common Causes:**

- Unsafe string/copy functions (`strcpy`, `gets`) without bounds
- Missing bounds checks when reading input
- Excessive recursion
- Incorrect length calculations

The following code example demonstrates how such a vulnerability can occur. In the code, the long input is copied into a buffer of 16 bytes. This causes the input to overflow into adjacent memory.

```
1  int main() {
2      char *a = malloc(16);
3      char *b = malloc(16);
4
5      char input[64] = "Long input causing buffer
           overflow when copied to a";
6
7      ...
8
9      strcpy(a, input);
10
11     ...
12 }
```

In the console output below, which shows the buffers along with their addresses and contents, it can be clearly observed that buffer *a* overflows into buffer *b*.

```
Before overflow:
    a @ 00000162cbaf6ab0 : 'AAAAAAAAAAAAAAA'
    b @ 00000162cbaf6ad0 : 'BBBBBBBBBBBBBBB'
After overflow:
    a @ 00000162cbaf6ab0 : 'Long input causing buffer overflow when
        copied to a'
    b @ 00000162cbaf6ad0 : 'ow when copied to a'
```

## 2.2 Format-String Attack

Format-string attack occurs when untrusted input is used directly as a format string in functions such as `printf` without sanitization. The format string controls how the function reads and displays memory, which allows attackers to read program memory and leak sensitive data. It can also cause crashes if an invalid read occurs.

**Common Causes:**

- Passing user input directly to print functions (`printf(input);`)
- Not distinguishing between data and format instructions
- Lack of input validation or sanitization

The following code example demonstrates how such a vulnerability can occur. In this example, user input containing format specifiers is passed directly to `printf`, causing it to interpret the input as a format string and read values from the stack.

```
1  int main() {
2      char input[] = "%x %x %x %x"
3      int secret = 0xfffffffe;
4
5      ...
6
7      printf(input);
8
9      return 0;
10 }
```

In the console output below, it can be observed that the format specifiers cause `printf` to read and display unintended memory contents from the stack, including the secret.

```
Secret (plain): fffffffe

Format string attack output:
fffffffe 61ff54 4019db 25207825
```

## 2.3 Use-After-Free

A use-after-free vulnerability occurs when a program continues to use a pointer after the memory it references has been freed. Accessing freed memory can corrupt data, cause crashes, and create hard to reproduce bugs in long running programs.

**Common Causes:**

- Freeing memory but keeping dangling pointers
- Returning freed pointers from functions
- When `free()` is called in one branch but pointer is used elsewhere

The following code example demonstrates how such a vulnerability can occur. In this example, memory is freed in one code branch, but the dangling pointer is later reused after a new allocation reuses the same memory address.

```
1  int main() {
2      ...
3      char *p = malloc(16);
4      strcpy(p, "PPPPPPPP");
5
6      if (input == 1) {
7          printf("%s", "[if] code branch\n");
8          free(p);
9      }
10     ...
11
12     char *q = malloc(16);
13     strcpy(q, "QQQQQQQQ");
14     ...
15     strcpy(p, "PPPPPPPP");
16
17     ...
18 }
```

In the console output below, it can be observed that after memory pointed to by *p* is freed, a new allocation for *q* reuses the same address. Writing with the dangling pointer *p* changes the contents of *q*.

```
p: @ 00000235f5df6a70 : PPPPPPPP
[if] code branch
q: @ 00000235f5df6a70 : QQQQQQQQ
Value of q: PPPPPPPP
```

## 2.4 Memory Leak

Memory Leak occurs when a program continues to allocate memory without freeing it, causing unused memory to accumulate. Memory leaks can lead to performance degradation, excessive memory usage, or program crashes (OOM).

**Common Causes:**

- Forgetting to call `free()` for dynamically allocated memory
- Losing references to allocated memory (overwriting pointers)
- Circular references in complex data structures

The following code example demonstrates how such a vulnerability can occur. In the code, *1MB* memory is allocated in a loop without being freed, causing the program to consume more and more memory over time.

```
1  int main() {
2      const size_t SIZE = 1024 * 1024;
3
4      for (int i = 0; i < 10000; i++) {
5          char *p = malloc(SIZE);
6          memset(p, 'A', SIZE-1); p[SIZE- 1] = '\0';
7
8          ...
9      }
10
11     return 0;
12 }
```

In the console output below, it can be observed that memory usage increases with each allocation. When memory limits are exceeded, the program may be terminated by the operating system.

```
Allocated 155 MB of memory
Allocated 156 MB of

"State": {
    "Status": "exited",
    "OOMKilled": true,
}
```

## 2.5 Summary

In this section, we analyzed several common memory related vulnerabilities in low-level programming languages, including Buffer Overflow, Format-String Attack, Use-After-Free, and Memory Leak. All of these vulnerabilities were caused by improper memory management, and they vary in their exploitability, detection, and severity.

**Buffer Overflow** and **Format-String attack** vulnerabilities are the most severe, as they can allow attackers to execute arbitrary code, read sensitive information, or corrupt program state.

**Use-After-Free** bugs are often less exploitable, but they may only appear under specific runtime conditions, making them difficult to detect, but are not that severe.

**Memory Leaks** are less exploitable, but can degrade performance, cause applications to become unresponsive, and eventually lead to program crashes due to running out of memory.

| **Vulnerability** | **Exploitability** | **Detection** | **Severity** |
|---|---|---|---|
| Buffer Overflow | High | Medium | High |
| Format-String Attack | High | Medium | High |
| Use-After-Free | Medium | Low | Medium |
| Memory Leak | Low | High | Medium |

Table 2: Summary of memory vulnerabilities

# 3 Software Analysis Tools

To minimize the risks of software vulnerability exploits, it is always good practice to test programs using specialized analysis tools. These tools are capable of detecting most common security bugs and differ in their purpose, the operating systems they support, and the scenarios they target.

| **Feature** | **Static Analysis** | **Dynamic Analysis** |
|---|---|---|
| When It Runs | Before execution | During execution |
| Detects | Risky patterns, ... | Memory leaks, ... |
| Can Find Memory Errors | Limited | Yes |
| False Positives | Higher | Lower |
| Performance Impact | None | Slows execution |
| Best For | Early development | Reproducing crashes |

Table 3: Comparison of static and dynamic analysis tools

## 3.1 Dynamic Analysis

Dynamic analysis tests the program during execution to detect incorrect memory operations that may not be visible in the source code. There are many tools for dynamic code analysis, in these experiments we will be using **Valgrind**.

- Finds runtime memory errors (buffer overflows, invalid reads/writes)
- Detects bugs that static analysis often cannot see
- Helps reproduce crashes and undefined behaviour
- Supports debugging with precise error locations

**Buffer Overflow**

The following code example from previous section demonstrates buffer overflow vulnerability.

```
1 int main() {
2     char *a = malloc(16);
3     char *b = malloc(16);
4
5     char input[64] = "Long input causing buffer
        overflow when copied to a";
6
```

```
7       ...
8
9       strcpy(a, input);
10
11      ...
12  }
```

When we test this program, we get expected output that recognizes invalid write of size 1 with precise error location at line 18. At the bottom we can see also message that states what was the probable cause of the error.

```
=3169 = Invalid write of size 1
=3169 = by 0x109345: main (buffer_overflow.c:18)
...
This is probably caused by your program erroneously writing past
the end of a heap block and corrupting heap metadata.
```

**Use-After-Free**

The following code example from previous section demonstrates use-after-free vulnerability.

```
1   int main() {
2       ...
3       char *p = malloc(16);
4       strcpy(p, "PPPPPPPP");
5
6       if (input == 1) {
7           printf("%s", "[if] code branch\n");
8           free(p);
9       }
10      ...
11
12      char *q = malloc(16);
13      strcpy(q, "QQQQQQQQ");
14      ...
15      strcpy(p, "PPPPPPPP");
16
17      ...
18  }
```

When we test this program, we get invalid write of size 8 at line 23. The output also shows that the memory block being written to was previously freed, including information about where the memory was allocated and freed.

```
=2547 = Invalid write of size 8
=2547 = at 0x109280: main (use_after_free.c:23)
=2547 = Address 0x4a74040 is 0 bytes inside a block of size 16 free'd
=2547 = by 0x10921E: main (use_after_free.c:14)
=2547 = Block was alloc'd at
=2547 = by 0x1091C5: main (use_after_free.c:8)
```

**Memory Leak**

The following code example from previous section demonstrates memory leak vulnerability.

```
1  int main() {
2      const size_t SIZE = 1024 * 1024;
3
4      for (int i = 0; i < 3; i ++) {
5          char *p = malloc(SIZE);
6
7          memset(p, 'A', SIZE-1); p[SIZE-1] = '\0';
8      }
9
10     return 0;
11 }
```

When we test this program, the output states that 3 memory blocks are still allocated at program exit. The report indicates that the allocated memory was never freed, resulting in definitely lost memory blocks.

```
=3154 = 3,145,728 bytes in 3 blocks are definitely lost
        in loss record 1 of 1
=3154 = by 0x1091B1: main (memory_leak.c:9)
```

## 3.2 Static Analysis

Static analysis inspects the source code without executing it, detecting potential memory safety issues early in development. Apart from that, static code analysis works well for identifying logical or syntax mistakes, etc. There are multiple tools for static analysis, in this report, **Cppcheck** is used.

- Finds logical mistakes and unsafe patterns before compilation
- Detects misuse of pointers, uninitialized variables, etc.
- Helps enforce coding standards and secure programming principles

In the following section, examples of static code analysis for different errors or warnings from the code will be presented. Each code snippet will have its corresponding output. Since the messages are self-explanatory, no additional comments will be given.

**Buffer Overflow**

```
1 char *data = malloc(8);
2 strcpy(data, "AAAAAAAAAAA");
```

```
bad_memory.c:7:12: error: Buffer is accessed out of bounds: data
    [bufferAccessOutOfBounds]
    strcpy(data, "AAAAAAAAAAA");
```

**Double Free**

```
1 free(data);
2 free(data);
```

```
bad_memory.c:12:5: error: Memory pointed to by 'data' is freed twice.
    [doubleFree]
    free(data);
```

**Null Pointer Dereference**

```
1 char *ptr = NULL;
2 printf("%c\n", ptr[0]);
```

```
bad_memory.c:15:20: error: Null pointer dereference: ptr
    [nullPointer]
    printf("%c\n", ptr[0]);
```

**Array Access Out of Bounds**

```
1 int arr[3] = { 1, 2, 3 };
2 printf("%d\n", arr[5]);
```

```
bad_memory.c:18:23: error: Array 'arr[3]' accessed at index 5.
    [arrayIndexOutOfBounds]
    printf("%d\n", arr[5]);
```

**Division by Zero**

```
1  int x = 10 / (size - size);
```

```
bad_io.c:9:16: error: Division by zero.
    [zerodiv]
    int x = 10 / (size - size);
```

**Value Never Used**

```
1  int x = 10 / (size - size);
```

```
bad_io.c:9:11: style: Variable 'x' is assigned a value that is never used.
    [unreadVariable]
    int x = 10 / (size - size);
```

**Uninitialized Variable**

```
1  int y;
2  if (y > 0) {
3    printf("%d\n", y);
4  }
```

```
bad_io.c:12:9: error: Uninitialized variable: y
    [uninitvar]
    if (y > 0) {
```

**Obsolete Function**

```
1  char input[10];
2  gets(input);
```

```
bad_io.c:17:5: warning: Obsolete function 'gets' called.
    [getsCalled]
    gets(input);
```

## 3.3 Evaluation

In this section, we evaluated several programs containing different types of memory and logical vulnerabilities using various analysis tools. The purpose of this evaluation is to compare the success of static and dynamic code analysis in detecting these issues, and to highlight their strengths and limitations.

**Dynamic code analysis** successfully detected all memory vulnerabilities mentioned in the previous section, except for format-string attacks, which are not easily detectable at runtime. On the other hand, dynamic analysis was unable to detect non-memory-related issues, such as uninitialized variables or obsolete function calls.

**Static code analysis** detected memory misuse only in some cases and under certain conditions. It was able to identify resource leaks and double-free errors, but only when the code was deterministic. Some static analysis tools were even able to detect format-string attacks. For non-memory-related issues, static analysis performed very well.

| **Analysis Type** | **Memory Issues** | **Logical Issues** | **Exploitation Issues** |
|---|---|---|---|
| Dynamic Analysis | High | Low | Low |
| Static Analysis | Medium | High | Medium |

Table 4: Success of code analysis on different issue types

# 4 Operating System Restrictions

Modern operating systems provide several runtime protections to reduce the risk of memory exploits in memory-unsafe languages. These protections lower the risk of attacks but do not solve memory bugs themselves. It's still important to test the code.

**Important Protections:**

- Address Space Layout Randomization (ASLR)
- Non-Executable Memory (NX)
- Stack Canaries

## 4.1 Address Space Layout Randomization

ASLR randomizes the locations of memory regions for each process to make attacks like buffer overflow harder to exploit. It randomizes locations of stack, heap, program code, and library code and makes it difficult to predict memory addresses for code injection or another attacks. ASLR is enabled by default and works without need of any changes to source code.

In Linux systems, it's possible to temporarily disable ASLR with following command and options *(2 - Full ASLR, 1 - Partial ASLR, 0 - ASLR disabled)*.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

## 4.2 Non-Executable

NX marks certain memory regions as non-executable, preventing attackers from running injected code. If attacker overflows a buffer and injects malicious bytes, NX prevents the CPU from running it. Modern CPUs include an NX/XD bit that the OS uses to enforce non-executable memory, when a program tries to execute non-executable memory, the process is terminated.

Most modern systems do not allow disabling NX globally because it is CPU-level. It must be disabled in BIOS and is not recommended.

# 5 Memory Safe Languages

Although it is not always possible, using modern memory-safe languages can prevent almost all memory vulnerabilities. In this section, we compare a *Rust* program with a *C/C++* program, because *Rust* is also considered a low-level language and serves a purpose most similar to *C/C++*. In addition, all high-level languages such as *Python*, *JavaScript*, and others are also memory-safe and enforce principles similar to *Rust's* ownership rules under the hood.

Consider the following example of a comparison between *C/C++* and *Rust* programs. *C/C++* programs usually compile with no problems, although they contain memory vulnerabilities. *Rust* programs do not even compile if they contain any memory misuse.

**C/C++**

```
1  int main() {
2      int *data = malloc(sizeof(int));
3      *data = 42;
4
5      free(data);
6      printf("%d", *data);
7
8      return 0;
9  }
```

```
-330996080
Process finished with exit code 0
```

**Rust**

```
1  fn main() {
2      let mut data = Box::new(42);
3
4      drop(data);
5
6      println!("{}", data);
7  }
```

```
error[E0382]: borrow of moved value: ‘data‘
  --> comparison.rs:6:20
   |
2  | let mut data = Box::new(42);
   | -------- move occurs because ‘data‘ has type ‘Box<i32>‘,
   |          which does not implement the ‘Copy‘ trait
3  |
4  | drop(data);
   | ---- value moved here
5  |
6  | println!("{}", data);
   |          ^^^^ value borrowed here after move
   |
error: aborting due to 1 previous error; 1 warning emitted
```

# 6 Recommendations

Based on the analysis and experiments performed in this project, several recommendations can be made to reduce the risk of memory-related vulnerabilities in software programmed in low-level languages.

**Integrate Static and Dynamic Analysis Early**

Static and dynamic analysis tools should be integrated into the development process as early as possible instead of being used only before release. Combining tools such as *Cppcheck* for static analysis and *Valgrind* for dynamic analysis helps detect both logical errors and runtime memory vulnerabilities during development, which can significantly reduce the cost of fixing these errors later.

**Adopt Memory-Safe Languages Where Possible**

Whenever it's possible, same purpose memory-safe languages should be preferred, especially for new modules and high-risk components. Languages such as *Rust* provide strong compile-time checks against memory errors while still offering low-level control similar to *C*/*C*++. High-level languages reduce the risk of memory misuse even more.

**Enable Operating System Protections by Default**

Operating system security mechanisms such as Address Space Layout Randomization (ASLR), Non-Executable Memory (NX), and Stack Canaries should be enabled by default in all build and deployment environments. Although these mechanisms do not eliminate memory bugs, they significantly increase the difficulty of exploiting them.

**Follow Secure Coding Guidelines**

Developers should follow secure coding guidelines to minimize the introduction of vulnerabilities. This includes validating all external input, avoiding unsafe APIs such as `gets()` and `strcpy()`, and using modern, safer alternatives that perform bounds checking. This also applies to the use of trusted and non-vulnerable libraries.

# 7 Conclusion

In this project, we analyzed common memory-related vulnerabilities in low-level programming languages and evaluated different methods for detecting them. Several common vulnerabilities were successfully demonstrated, including Buffer Overflow, Format-String Attack, Use-After-Free, and Memory Leak. All experiments were executed in controlled environments, and the selected tools and operating systems worked as expected during testing.

Both static and dynamic analysis tools performed well, but each had its own strengths and limitations. Dynamic analysis, using Valgrind, was able to detect most runtime memory errors such as buffer overflows, use-after-free bugs, and memory leaks. However, it was not able to reliably detect format-string attacks, since these issues do not always cause direct memory erros at runtime. Static analysis tools, such as Cppcheck, were effective at detecting logical errors, unsafe patterns, and some memory issues like double-free and resource leaks. Their main limitation was the inability to detect non-deterministic behavior and some of the runtime memory errors.

The experiments also showed that memory-safe programming languages can prevent many memory-related vulnerabilities. The comparison between *C/C++* and *Rust* demonstrated that Rust prevents memory misuse at compile time using ownership and borrowing rules. Programs containing memory errors do not compile, which significantly reduces the risk of these vulnerabilities.

Overall, the results show that combining static and dynamic analysis tools, secure coding practices, operating system protections, and memory-safe languages provides the best approach to reducing memory-related vulnerabilities. While no single method can fully eliminate these issues, their combined use improves software security and reliability.

# 8 References


OWASP Foundation. *Buffer Overflow*
https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

Wikipedia. *Buffer Overflow*
https://en.wikipedia.org/wiki/Buffer_overflow

OWASP Foundation. *Format String Attack*
https://owasp.org/www-community/attacks/Format_string_attack

Wikipedia. *Uncontrolled Format String*
https://en.wikipedia.org/wiki/Uncontrolled_format_string

OWASP Foundation. *Using Freed Memory*
https://owasp.org/www-community/vulnerabilities/Using_freed_memory

Wikipedia. *Dangling Pointer*
https://en.wikipedia.org/wiki/Dangling_pointer

OWASP Foundation. *Memory Leak*
https://owasp.org/www-community/vulnerabilities/Memory_leak

Wikipedia. *Memory Leak*
https://en.wikipedia.org/wiki/Memory_leak

**These OWASP and Wikipedia resources were used to describe common memory-related security vulnerabilities, their causes, and impact.**

Docker Inc. *Docker Documentation*
https://docs.docker.com/manuals/

**The Docker documentation was used to configure memory limits and to run memory-leak experiments in an isolated environment.**

Consumer Reports Innovation Lab. *Memory Safety Convening Report*
https://innovation.consumerreports.org/Memory-Safety-Convening-Report-.pdf

**This report provides statistics and analysis on memory safety vulnerabilities and discusses their impact on modern software development.**

Wikipedia. *Dynamic Program Analysis*
https://en.wikipedia.org/wiki/Dynamic_program_analysis

Valgrind Developers. *Valgrind User Manual*
https://valgrind.org/docs/manual/manual.html

Wikipedia. *Static Program Analysis*
https://en.wikipedia.org/wiki/Static_program_analysis

Cppcheck Team. *Cppcheck Manual*
https://cppcheck.sourceforge.io/manual.html

**These resources were used to explain static and dynamic code analysis and to understand the functionality of the corresponding analysis tools.**

Danilo Az. *Differences Between ASLR, KASLR, and KARL*
https://www.daniloaz.com/en/blog/differences-between-aslr-kaslr-and-karl

ResearchGate. *Defeating Code-Reuse Attacks with Binary Instrumentation*
https://www.researchgate.net/publication/333828764

Linux Audit. *Linux ASLR and randomize_va_space Setting*
https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting

Medium. *Security: NX Bit (Non-Executable Memory)*
https://medium.com/@boutnaru/security-nx-bit-non-executable-18759fd2802e

**These articles were used to describe operating system memory protections, their purpose, and their role in preventing memory exploitation.**

U.S. Department of Defense. *Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development*
https://media.defense.gov/2025/Jun/23/2003742198/-1/-1/0/CSI_MEMORY_SAFE_LANGUAGES_REDUCING_VULNERABILITIES_IN_MODERN_SOFTWARE_DEVELOPMENT.PDF

**This report discusses memory-safe programming languages and analyzes their effectiveness in reducing memory-related vulnerabilities.**