Computer and Network Security

Project: Integer Overflows in C++

October 2024

Vignesh Aravindh B
CS22B2004

Index

1 Introduction

Integer overflow is a critical vulnerability that occurs when an arithmetic operation results in a value that exceeds the storage capacity of the integer type used. This situation can lead to unexpected behavior, including memory corruption, crashes, and security vulnerabilities such as buffer overflows. In languages like C++, where manual memory management is common, the risks associated with integer overflows are particularly pronounced. Such vulnerabilities can be exploited by attackers to manipulate program behavior or gain unauthorized access to system resources.

To mitigate these risks, it is essential to identify potential points of overflow in the code. This project aims to demonstrate a parser utilizing regular expressions to detect instances where integer overflows may occur. This parser analyzes source code to identify risky operations. By proactively addressing these vulnerabilities, developers can reduce the likelihood of security breaches and improve the robustness of their applications.

The C/C++ language is the most vulnerable due to its underlying design philosophy. The C/C++ is designed to provide low-level control over system resources to programmers, however this comes with a cost that the programmer is responsible for writing a clean code. Hence the compilers of C/C++ do not perform extensive runtime tests for inappropriate memory access and when handles the compiler results in undefined behavior, leading to unpredictable situations.

2 Methodology and Code Explanation

The methodology of this project involves the development of a parser designed to detect potential vulnerabilities associated with integer overflows in C++ code. The parser leverages **regular expressions** to analyze source code for specific patterns that may indicate risks, such as casting operations, arithmetic operations, and variable declarations. The approach can be broken down into several key components:

2.1 1. Design of the Parser

The parser is implemented as a C++ class, ‘Parser‘, which contains various methods responsible for different types of checks. The main function, ‘parse‘, is responsible for reading a specified C++ source file line by line. Each line is then subjected to a series of checks to identify potential vulnerabilities.

2.2 2. Regular Expressions for Detection

Regular expressions (regex) are employed throughout the parser to match patterns in the code. The following key checks are implemented:

- **Variable Declarations**: The `checkVariableDeclarations` method uses the regex pattern:

```
R"((unsigned\s+short|unsigned\s+int|int|long|unsigned\s+long|float|double
    )\s+(\w+)\s*=\s*([^;]+);)"
```

- **Arithmetic Operations**: The `checkArithmeticOperation` method looks for arithmetic expressions using the regex pattern:

```
R"((\w+)\s*([+\-*/])\s*(\w+))"
```

- **Casting Operations**: The `checkCastingOperations` method identifies static casts using the regex pattern:

```
R"((\w+)\s*=\s*static_cast<(\w+)>\s*\((\w+)\))"
```

- **String Length Assignments**: The `checkStringLengthAssignment` method checks for assignments using the regex pattern:

```
R"((\w+)\s*=\s*strlen\((\w+)\))"
```

- **Comparison Operations**: The `checkComparisonOperations` method analyzes comparisons using the regex pattern:

```
R"((\w+)\s*([<>=!]=?)\s*(\w+))"
```

2.3 3. Warning Management

To manage warnings efficiently, the parser maintains a `set` called `issuedWarnings` to avoid duplicate warnings for the same line of code. This enhances the usability of the tool, allowing developers to focus on unique issues rather than being overwhelmed by repetitive messages.

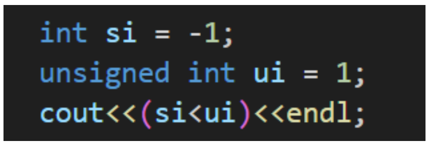
2.4 4. Data Structure for Variable Types - Symbol Table Map

The `variableTypes` unordered map is used to store the types of variables declared in the code. This information is essential for performing type comparisons and checks in arithmetic and comparison operations, helping to ensure that the parser accurately identifies potential vulnerabilities.

2.5 5. Usage and Results

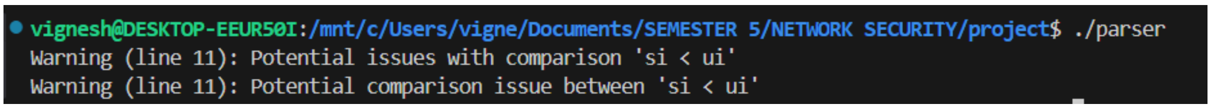
The parser is executed by calling the `parse` method with the filename of the C++ source code as an argument. It produces console warnings indicating potential issues found in the code, such as implicit casting risks, potential overflows, and unsafe assignments. Two test cases have been displayed below in Figure 1 and Figure 2. The output allows developers to quickly identify and address vulnerabilities that could lead to security risks in their applications.

Overall, the parser provides a systematic approach to analyzing C++ code for integer overflow vulnerabilities, that can help to significantly enhancing the safety and robustness of software developed in this language.



```
int si = -1;
unsigned int ui = 1;
cout<<(si<ui)<<endl;
```

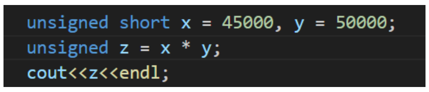
(a) Parser Output



```
vignesh@DESKTOP-EEUR50I:/mnt/c/Users/vigne/Documents/SEMESTER 5/NETWORK SECURITY/project$ ./parser
Warning (line 11): Potential issues with comparison 'si < ui'
Warning (line 11): Potential comparison issue between 'si < ui'
```

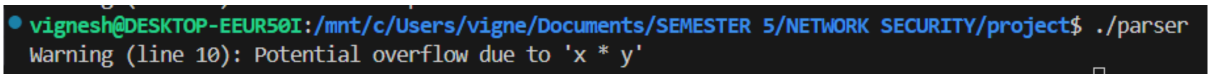
(b) Test Code

Figure 1: Parser output and corresponding test code 1



```
unsigned short x = 45000, y = 50000;
unsigned z = x * y;
cout<<z<<endl;
```

(a) Parser Output



```
vignesh@DESKTOP-EEUR50I:/mnt/c/Users/vigne/Documents/SEMESTER 5/NETWORK SECURITY/project$ ./parser
Warning (line 10): Potential overflow due to 'x * y'
```

(b) Test Code

Figure 2: Parser output and corresponding test code 2

3 Future Work

The following section outlines the potential areas for further development and improvement of the parser:

- **Extending Detection Capabilities:** Future versions of the parser could enhance detection capabilities by covering additional vulnerabilities, such as buffer overflows, format string vulnerabilities, and uninitialized memory usage, making it a more comprehensive tool for C++ security analysis.

- **Integration with IDEs:** Integrating the parser as a plugin for popular IDEs (e.g., Visual Studio Code, CLion) could enhance its use, providing developers with more effective feedback on npotential vulnerabilities during development and deployment.
- **Machine Learning:** Using machine learning algorithms to identify common patterns in unsafe code can improve the parser's accuracy in detecting issues, potentially reducing false positives.
- **Performance Optimization:** As the parser is extended, optimizing its performance for large codebases will become increasingly important. Techniques such as multithreading could be explored to enhance efficiency.

These enhancements will aim to make the parser a more versatile, powerful and independent tool for ensuring C++ code security, usability, and efficiency.

4 Scope

This project is focused on developing a parser that identifies potential security vulnerabilities in C++ code related to casting operations, operator conversions, arithmetic and comparison operations, and integer overflow issues. The primary objectives of this parser include:

- **Targeted Vulnerability Detection:** The parser is designed to detect specific vulnerabilities that commonly lead to security risks in C++ code, such as implicit casting, unsafe assignments, overflow/underflow, and issues with 64-bit portability.
- **Console-Based Output:** For simplicity and ease of integration, the parser provides a console-based output with warnings and alerts, allowing developers to quickly identify and address potential issues without the need for additional software.
- **Static Code Analysis Only:** This tool performs static code analysis, meaning it examines the code without executing it. This approach is effective for detecting syntactical and structural vulnerabilities but does not account for runtime behaviors.
- **Focused on Integer-Related Vulnerabilities:** Although the parser highlights integer overflow/underflow vulnerabilities, it does not cover all possible C++ security issues, such as buffer overflows, pointer dereferencing issues, or memory leaks.

5 Conclusion

This project implements a C++ code parser that identifies potential vulnerabilities, specifically focusing on arithmetic overflow, implicit casting issues, and comparisons between signed and unsigned variables. This parser systematically examines each file in a given directory, providing warnings on code segments that could lead to undefined behavior or security risks. By automating the detection of common errors, this tool can aid developers in enhancing the reliability and security of C++ applications.

References

[1] Common Weakness Enumeration (CWE190) - Integer Overflow or Wraparound. CWE 190

[2] Information Security - Integer Overflow Information Security

[3] Chat GPT Chat GPT

[4] 24 DEADLY SINS OF SOFTWARE SECURITY Programming Flaws and How to Fix Them

- Michael Howard, David LeBlanc, and John Viega