COMPUTER NETWORK SECURITY

Project : Integer Overflows in C++

Vignesh Aravindh B – CS22B2004

Project Workflow Overview

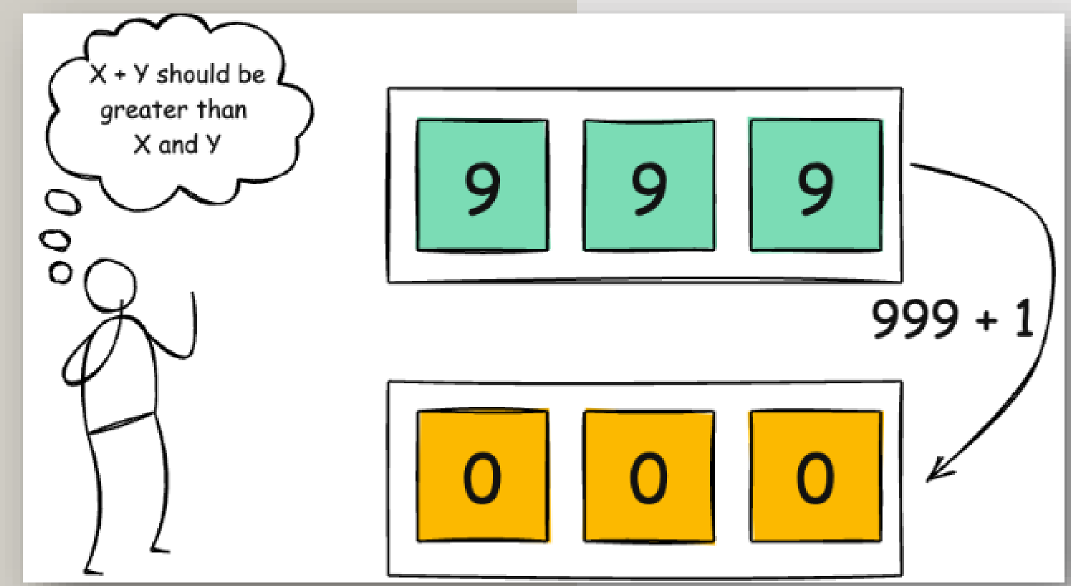
1. Literature Review
2. Sinful C / C++
3. Regular Expression
4. Parser Overview

STATUS SUMMARY

1. Read the book, internet sources and papers related to Integer Overflows.
2. Tested Custom Sinful Codes in C++ and noted their observations.
3. Made a summary of the Sin and how it is implemented in C++.
4. Designed several RE’s for static parsing of the code.

LITERATURE REVIEW

WHAT IS IT?



Source – CWE 190

Integer Overflow – CWE 190

- The product performs a calculation that can produce an integer overflow or wraparound when the logic assumes that the resulting value will always be larger than the original value. This occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may become a very small or negative number.

Integer Overflow – My definition

- Integer overflow occurs when an arithmetic operation exceeds the maximum (or minimum) value that can be stored in an integer variable, potentially leading to memory corruption such as buffer overflows when invariably used in memory allocation, etc.

CWE* - Common Weakness Enumeration

SINFUL C/C ++

The following are the potential sources of the Weakness in the Code that could be exploited to make it Vulnerability:

- Casting Operations
- Operator Conversions
- Arithmetic Operations
- Comparison Operations
- Binary Operations
- 64-bit Portability Issues
- Compiler Optimisations

THREE RULES

```
if( a + b < c)
{
    cout<<"Something"<<endl;
}
```

- If either of the operands are of type **unsigned long**, then both are converted to **unsigned long**.
- If both the operands are 32-bits long or less, then are promoted to **int** type for the purpose of operation, which is called as integer promotion in C++.
- If one of the operand is 64-bit, then the other operand is also promoted or upcast to 64-bit with an **unsigned 64-bit** value being an upper bound.

Operand Type 1	Operand Type 2	Implicit Type Cast
Signed short	Signed int	Both are promoted to 32-bit int type
Signed short	Signed long	The short is upcast to 64-bit for operation
Signed int	Signed short	Both are promoted to 32-bit int type
Signed int	Signed long	The int is upcast to 64-bit for operation
Signed long	Signed short	The short is upcast to 64-bit for operation
Signed long	Signed int	The int is upcast to 64-bit for operation
Unsigned short	Unsigned int	Both are promoted to 32-bit int type
Unsigned short	Unsigned long	The short is promoted to unsigned long
Unsigned int	Unsigned short	Both are promoted to 32-bit int type
Unsigned int	Unsigned long	The int is promoted to unsigned long
Unsigned long	Unsigned int	The int is promoted to unsigned long
Unsigned long	Unsigned short	The short is promoted to unsigned long

Operand Type 1	Operand Type 2	Implicit Type Cast
Signed short	Unsigned long	The short is promoted to **unsigned** long int
Signed short	Unsigned int	The short is upcast to **unsigned** int
Signed short	Unsigned short	Both are upcast to **signed** int
Signed int	Unsigned long	The int is upcast is upcast to **unsigned** long
Signed int	Unsigned int	The int is upcast to **unsigned** int
Signed int	Unsigned short	The short is promoted to **signed** int
Signed long	Unsigned long	The signed long is upcast to **unsigned** long
Signed long	Unsigned int	Both are promoted to **unsigned** long
Signed long	Unsigned short	Both are promoted to **unsigned** long

Other Weaknesses

```
unsigned short x = 45000, y = 50000;
unsigned int z = x * y;
```

Figure 1.1

```
int cch = strlen(str);
```

Figure 1.2

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
      if(x + y < x)
            return false;

      return true;
}
```

Figure 1.3

The figures show how C/C++ sacrifices security.

- In the figure 1.1 everything appears fine unless the implicitly typecasted x and y to signed int, their, product lead to a very large value that is not representable in the range of signed int.
- In the next figure, well, a very common statement written by many while working with string manipulations. One forgets that the return type of the **strlen()** is **size_t** which is unsigned int and the nightmares that occurs when assigning a **unsigned int** to **int(signed)** when it lies outside the bounds of int, then the value just gets wrapped to around some negative value, imagine using ***cch*** in memory allocation.

As seen until now, every operation in C/C++ comes with a vulnerability due to the underlying facts of its Design Philosophy and the Compiler Behavior.

1) **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.
2) **Compiler Behavior** – 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.

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < ui);
```

Figure 2.1

```
int si = -1;
unsigned ui = 1;
printf("%d\n", si < (int)ui);
```

Figure 2.2

Figure 2.1: A noticeable flaw in the code is that when a signed int and unsigned long int are in operation with a comparison or any other operator then the signed int is upcast to unsigned long int, consider we are using 8-bit for the unsigned int then the -1 would become 255, which, makes the statement to print 0 although the expected outcome is 1, which is corrected in Figure 2.2.

REGULAR EXPRESSIONS FOR TYPE CHECKING

1) Variable Declaration :

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

2) Arithmetic Operations :

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

3) Casting Implicit and Static :

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

4) 64-bit Issues (strlen()):

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

5) Comparison Operators:

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

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.

THANK YOU