# Defensive Programming

## Contents

## What is a Code Vulnerability?

Many exploits and security breaches are caused by insecure code which makes the program vulnerable to attack. A code vulnerability is a flaw or weakness in a software application's code that can be exploited by attackers to compromise the system's security. It can result from errors, oversights, or inadequate coding practices.

Such vulnerabilities can allow unauthorized access, data breaches, or manipulation of the program's behavior, leading to potential system crashes, information leaks, or unauthorized actions, compromising the integrity, confidentiality, and availability of the software and the data it handles.

## Bad Coding Practices

Bad coding practices create exploitable vulnerabilities within software applications. Neglecting input validation, using outdated functions, or mishandling memory can introduce weaknesses that attackers can exploit.

For instance, insufficient input validation might allow attackers to inject malicious code, execute arbitrary commands, or manipulate data through buffer overflows, SQL injection, or cross-site scripting attacks. Using deprecated functions or failing to perform proper bounds checking in loops can lead to exploitable vulnerabilities that attackers can leverage to gain unauthorized access, corrupt data, or cause the application to crash.

These vulnerabilities can be exploited to compromise the integrity, confidentiality, or availability of the software, emphasizing the critical importance of following secure coding practices to mitigate such risks.

## Defensive Programming

Defensive Programming 1 (A 3 part series of short videos on Defensive Programing using Python.)

Defensive programming refers to a set of programming practices aimed at creating robust and resilient software that can withstand unexpected inputs, errors, or attacks while maintaining its functionality, security, and reliability. It involves adopting proactive strategies to anticipate potential failures, vulnerabilities, and misuse, and to mitigate these risks effectively. Here are key principles of defensive coding:

**Input Validation:** Always thoroughly validate and sanitize all user and data inputs to prevent unexpected or malicious data from compromising the application's integrity. This helps thwart attacks like SQL injection, buffer overflows, and other forms of injection attacks.

**Error Handling:** Implement comprehensive error handling mechanisms to gracefully manage unexpected conditions or failures, providing informative error messages and handling exceptions to prevent crashes or unexpected behaviors.

**Boundary and Range Checking:** Ensure that loops, data structures, and arithmetic operations are properly bounded and checked to avoid integer overflows, out-of-bounds memory access, or unexpected behavior due to incorrect data manipulation.

**Memory Management:** Use safe memory allocation practices, avoid raw pointers whenever possible, and leverage memory management tools like smart pointers to minimize memory leaks, dangling pointers, and other memory-related vulnerabilities.

**Secure Coding Practices:** Follow secure coding standards and utilize secure libraries, functions, and APIs. Regularly update dependencies to patch vulnerabilities and adhere to best practices for cryptography, authentication, and authorization.

**Code Reviews and Testing:** Conduct thorough code reviews and implement robust testing procedures (unit testing, integration testing, etc.) to identify and rectify vulnerabilities, bugs, or weaknesses early in the development cycle.

**Collaborate with other developers:** Paired code reviews can help facilitate knowledge sharing among team members. Developers can learn from one another and share best practices to improve overall code quality and security.

**Documentation and Comments:** Maintain clear, concise, and up-to-date documentation and comments to enhance code readability, ease maintenance, and facilitate collaboration among developers.

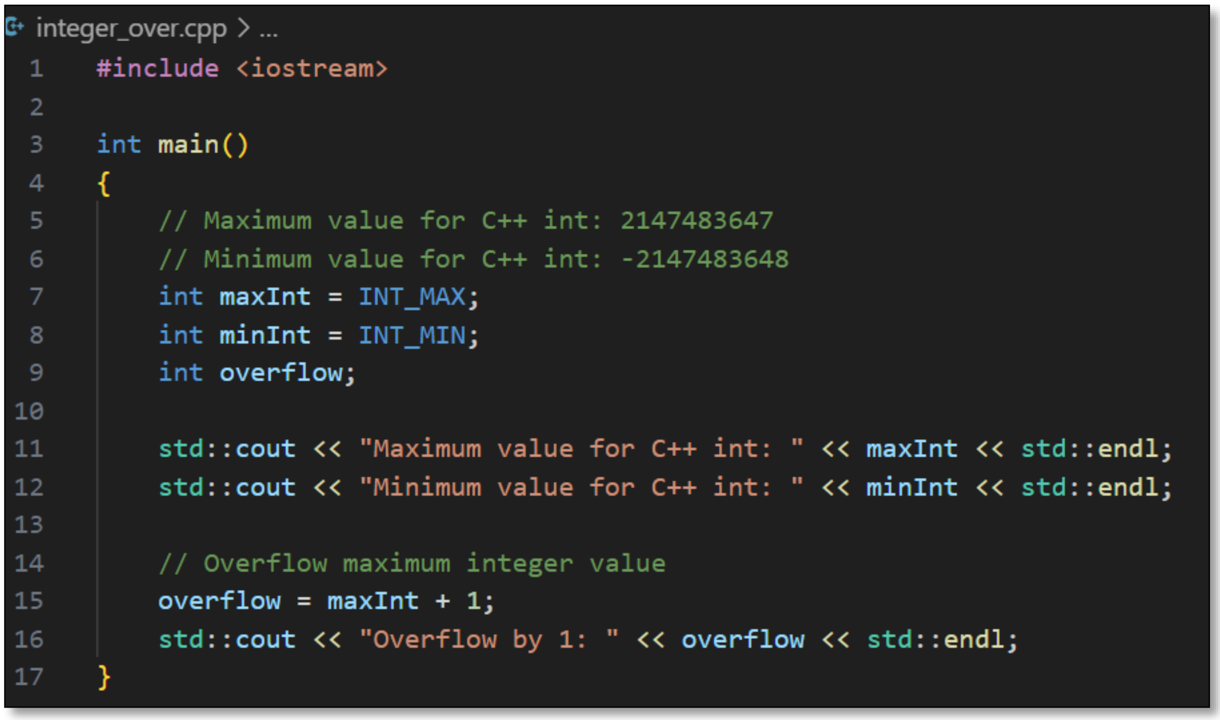
**Principle of Least Privilege:** Follow the principle of least privilege by granting minimal access or permissions necessary for the application, reducing the attack surface and potential impact of security breaches.

**Ongoing Training:** Educate developers and team members on a continuing basis. This can include best coding practices, security standards, and secure coding guidelines.

Defensive coding prioritizes building software that anticipates and guards against potential threats, errors, or misuse, fostering resilience and ensuring a more secure and reliable application in the face of evolving challenges.

## C++ Integer Overflow

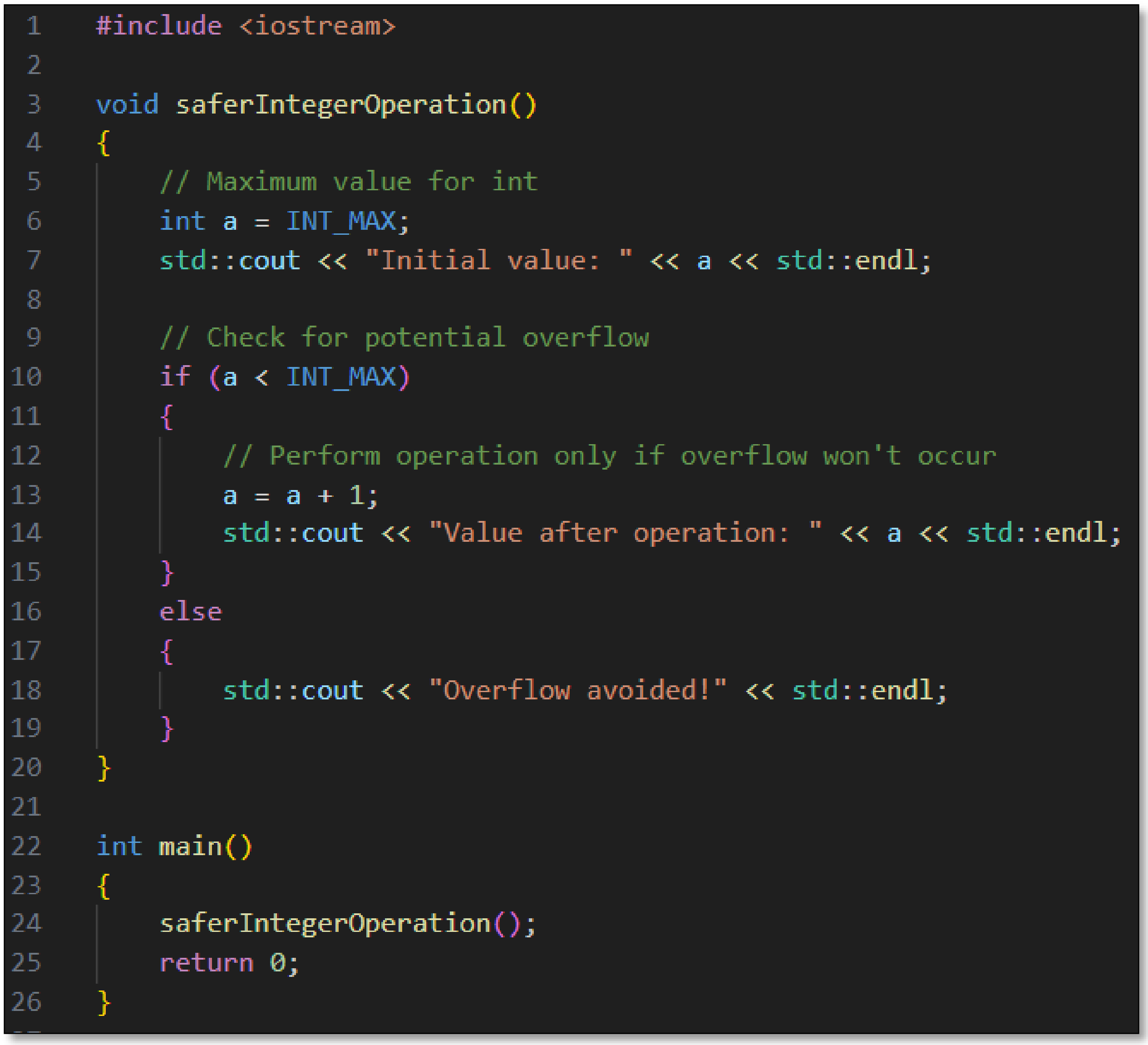
An integer overflow occurs when arithmetic operations on integers exceed the maximum or minimum value for the data type. Exploiting an integer overflow involves manipulating input to trigger the overflow, then using the result value to execute arbitrary code.


integer_over.cpp > ...

```cpp
#include <iostream>

int main()
{
    // Maximum value for C++ int: 2147483647
    // Minimum value for C++ int: -2147483648
    int maxInt = INT_MAX;
    int minInt = INT_MIN;
    int overflow;

    std::cout << "Maximum value for C++ int: " << maxInt << std::endl;
    std::cout << "Minimum value for C++ int: " << minInt << std::endl;

    // Overflow maximum integer value
    overflow = maxInt + 1;
    std::cout << "Overflow by 1: " << overflow << std::endl;
}
```

The program attempts to increment an **int** variable `a` by 1. This causes an integer overflow or wrap around to the minimum int value as the operation exceeds the maximum value represented by **INT_MAX**.

**Mitigation**

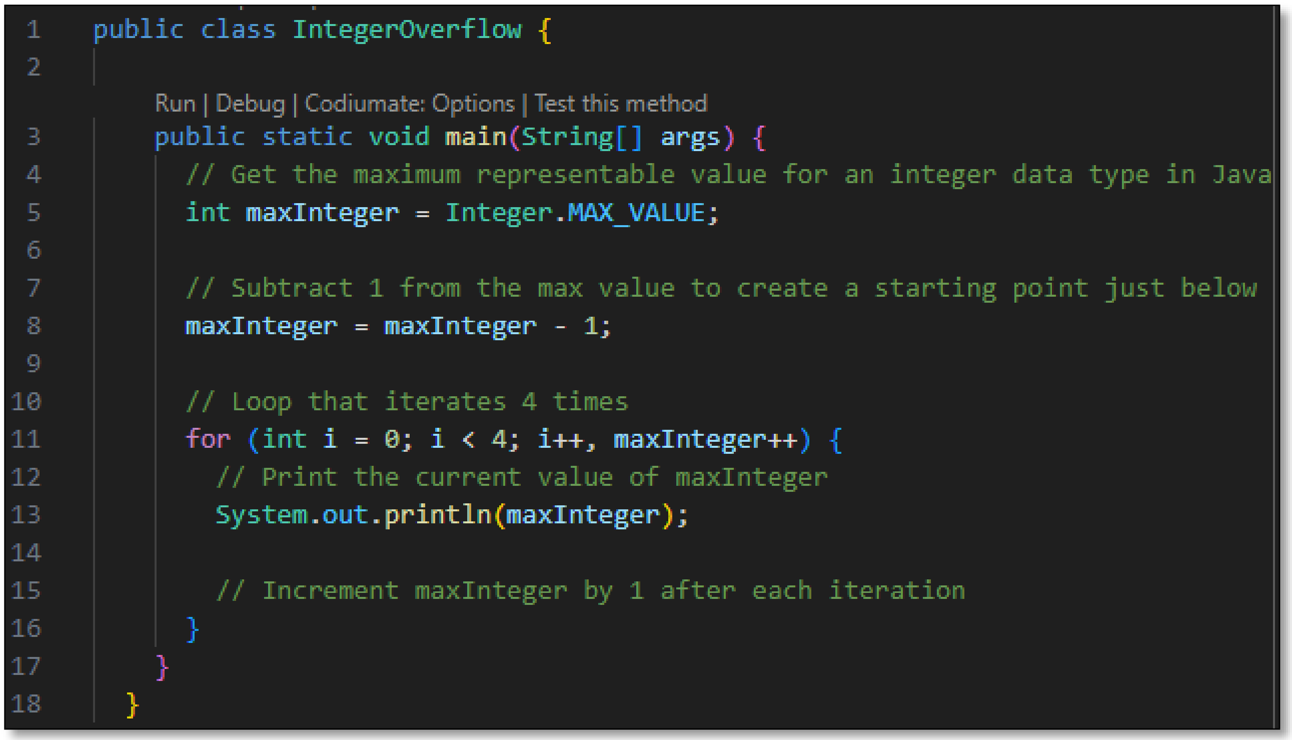


```cpp
#include <iostream>

void saferIntegerOperation()
{
    // Maximum value for int
    int a = INT_MAX;
    std::cout << "Initial value: " << a << std::endl;

    // Check for potential overflow
    if (a < INT_MAX)
    {
        // Perform operation only if overflow won't occur
        a = a + 1;
        std::cout << "Value after operation: " << a << std::endl;
    }
    else
    {
        std::cout << "Overflow avoided!" << std::endl;
    }
}

int main()
{
    saferIntegerOperation();
    return 0;
}
```

Before performing the increment operation, input validation is performed to ensure that the operation won't cause an overflow.

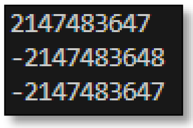
## Java Integer Overflow

This code demonstrates the concept of integer overflow. When **maxInteger** is incremented after reaching its maximum value, it "overflows" and wraps around to the minimum negative value an **int** can hold. This can lead to unexpected results in your program. It's essential to be aware of integer overflow and use data types with larger ranges (like **long**) or handle potential overflows in your code if necessary.



```java
public class IntegerOverflow {

    Run | Debug | Codiumate: Options | Test this method
    public static void main(String[] args) {
      // Get the maximum representable value for an integer data type in Java
      int maxInteger = Integer.MAX_VALUE;

      // Subtract 1 from the max value to create a starting point just below
      maxInteger = maxInteger - 1;

      // Loop that iterates 4 times
      for (int i = 0; i < 4; i++, maxInteger++) {
        // Print the current value of maxInteger
        System.out.println(maxInteger);

        // Increment maxInteger by 1 after each iteration
      }
    }
  }
```

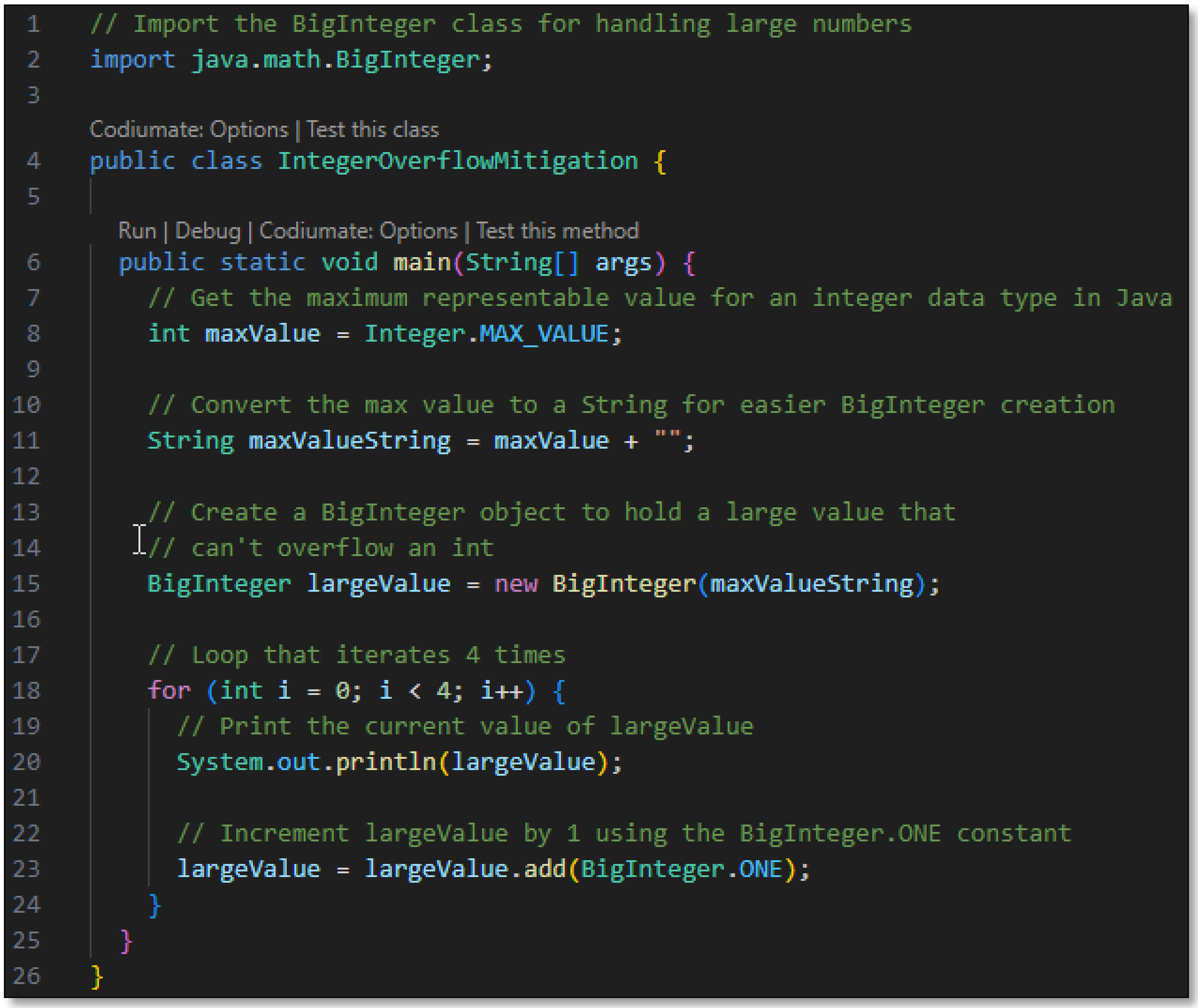
Example run:



```
2147483647
-2147483648
-2147483647
```

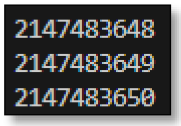
This code demonstrates how to avoid integer overflow by using the **BigInteger** class for calculations that might exceed the limits of an **int.** This ensures your program can handle larger numbers without unexpected results.

**Mitigation**



```
// Import the BigInteger class for handling large numbers
import java.math.BigInteger;

Codiumate: Options | Test this class
public class IntegerOverflowMitigation {

  Run | Debug | Codiumate: Options | Test this method
  public static void main(String[] args) {
    // Get the maximum representable value for an integer data type in Java
    int maxValue = Integer.MAX_VALUE;

    // Convert the max value to a String for easier BigInteger creation
    String maxValueString = maxValue + "";

    // Create a BigInteger object to hold a large value that
    // can't overflow an int
    BigInteger largeValue = new BigInteger(maxValueString);

    // Loop that iterates 4 times
    for (int i = 0; i < 4; i++) {
      // Print the current value of largeValue
      System.out.println(largeValue);

      // Increment largeValue by 1 using the BigInteger.ONE constant
      largeValue = largeValue.add(BigInteger.ONE);
    }
  }
}
```

Example run:

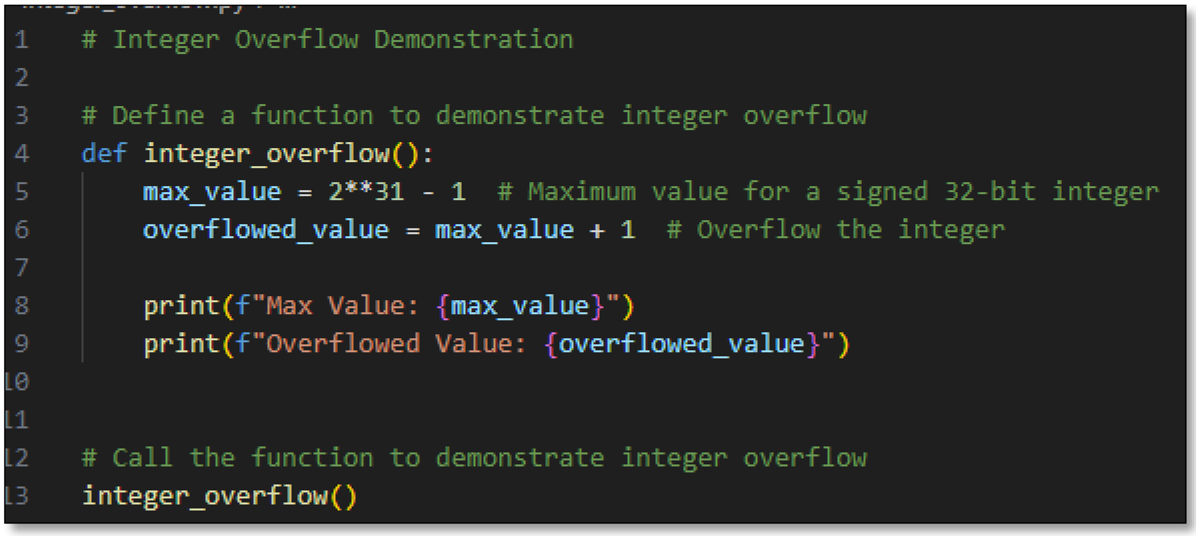


```
2147483648
2147483649
2147483650
```

## Python is Overflow Safe

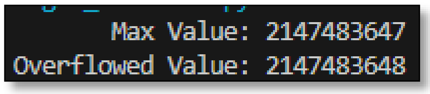
The following Python code does not demonstrate integer overflow in the way it might be expected to, due to Python's handling of integers. Python automatically handles large integers by allocating more memory to store their values, thus avoiding the traditional overflow issue seen in languages with fixed-size integer types (like C or Java for signed 32-bit integers).

To demonstrate integer overflow in Python, you would typically need to simulate the overflow behavior by manually enforcing the integer size limits, as Python itself does not have a built-in mechanism for integer overflow due to its dynamic nature.



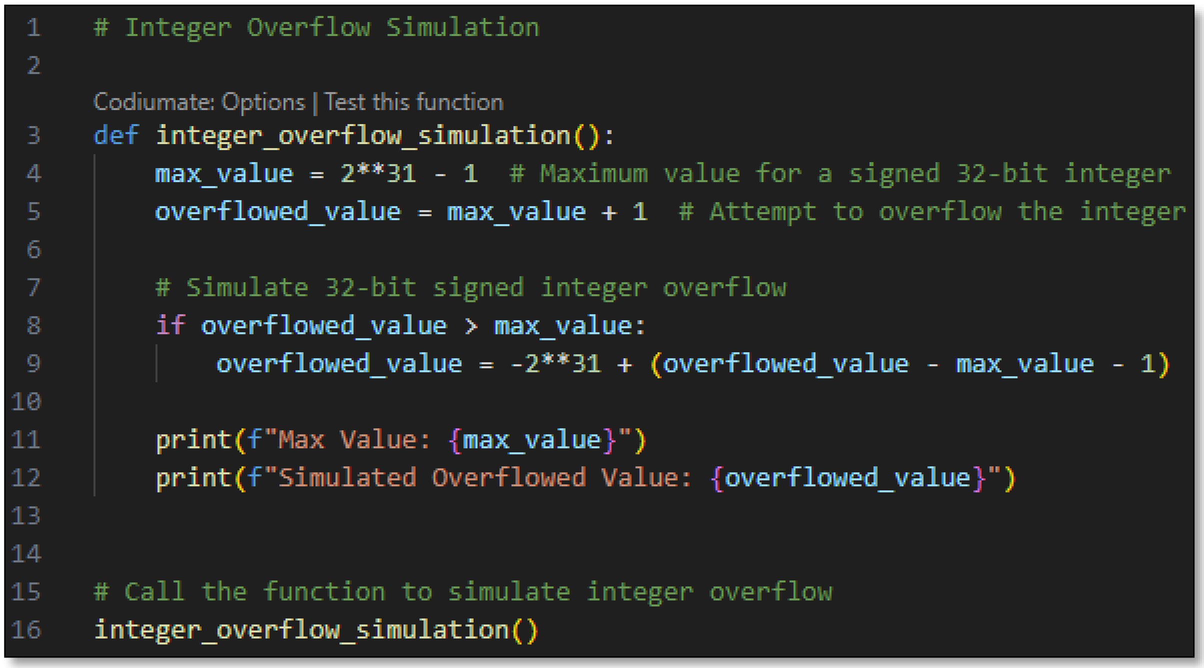
```python
# Integer Overflow Demonstration

# Define a function to demonstrate integer overflow
def integer_overflow():
    max_value = 2**31 - 1  # Maximum value for a signed 32-bit integer
    overflowed_value = max_value + 1  # Overflow the integer

    print(f"Max Value: {max_value}")
    print(f"Overflowed Value: {overflowed_value}")


# Call the function to demonstrate integer overflow
integer_overflow()
```

Example run:



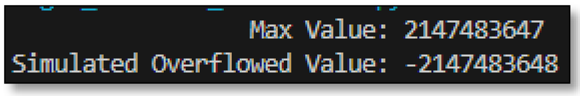
```
      Max Value: 2147483647
Overflowed Value: 2147483648
```

Here's a revised version of the code that simulates integer overflow by using arithmetic to mimic the behavior of a 32-bit signed integer:



```python
# Integer Overflow Simulation

Codiumate: Options | Test this function
def integer_overflow_simulation():
    max_value = 2**31 - 1  # Maximum value for a signed 32-bit integer
    overflowed_value = max_value + 1  # Attempt to overflow the integer

    # Simulate 32-bit signed integer overflow
    if overflowed_value > max_value:
        overflowed_value = -2**31 + (overflowed_value - max_value - 1)

    print(f"Max Value: {max_value}")
    print(f"Simulated Overflowed Value: {overflowed_value}")


# Call the function to simulate integer overflow
integer_overflow_simulation()
```

Example run:

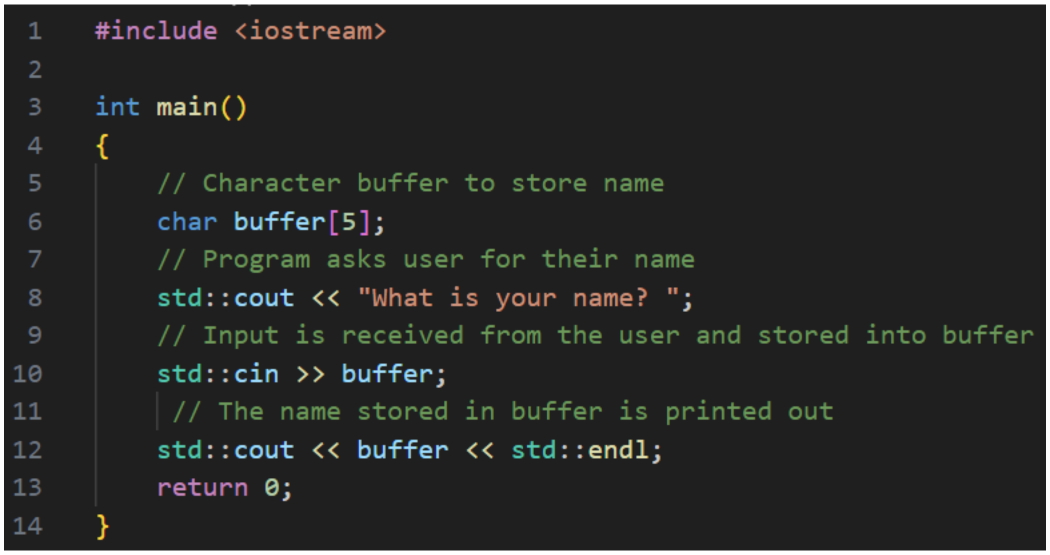


```
           Max Value: 2147483647
Simulated Overflowed Value: -2147483648
```

This code manually calculates the overflowed value to simulate how it would behave in a system with 32-bit signed integers, where exceeding the maximum value wraps around to the minimum value.

## Buffer Overflow

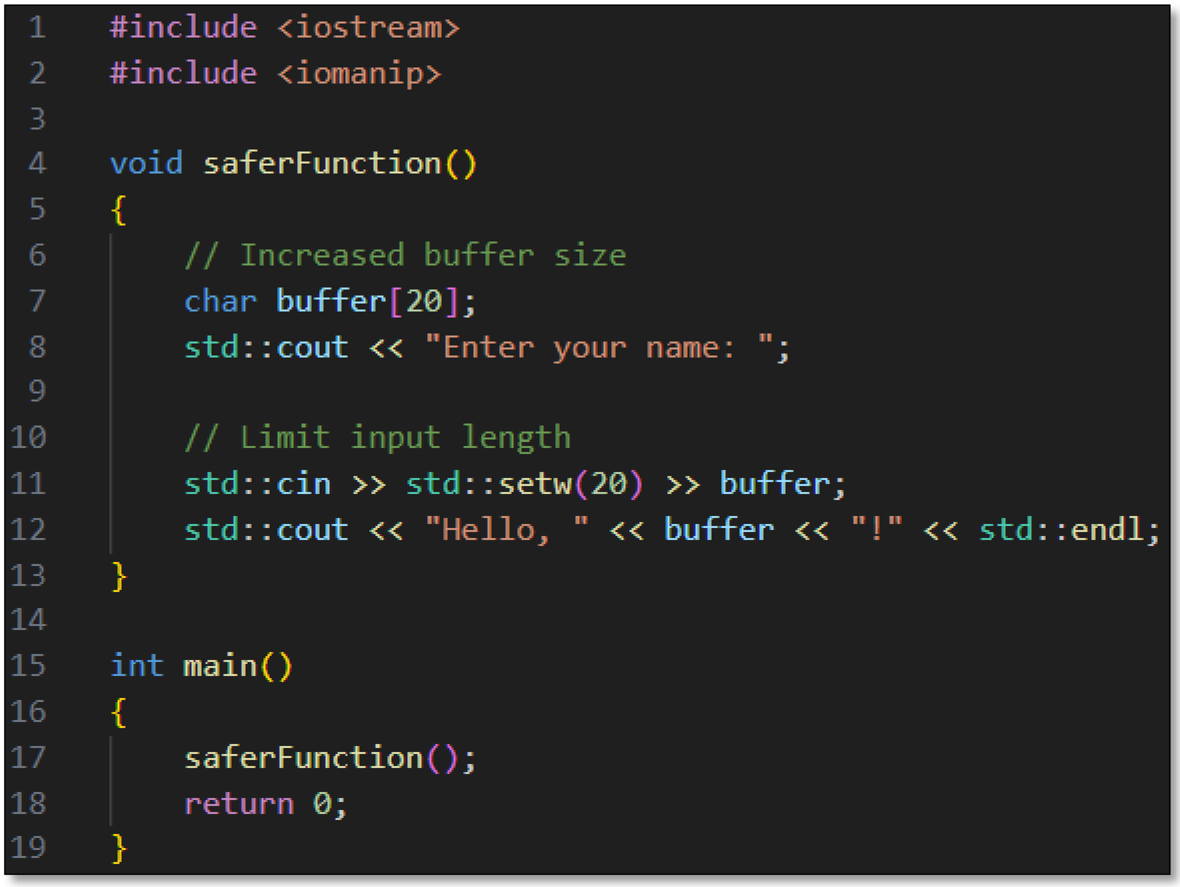
A buffer overflow occurs when a program writes more data into a buffer (an allocated memory space) than it can hold. This leads to the excess data overflowing into adjacent memory locations, potentially overwriting critical data or even allowing attackers to execute arbitrary code, causing crashes or enabling malicious actions.



```cpp
#include <iostream>

int main()
{
    // Character buffer to store name
    char buffer[5];
    // Program asks user for their name
    std::cout << "What is your name? ";
    // Input is received from the user and stored into buffer
    std::cin >> buffer;
    // The name stored in buffer is printed out
    std::cout << buffer << std::endl;
    return 0;
}
```

In this example, the program attempts to read user input into a character array **buffer** of size 5. However, if the user inputs more than 5 characters, it overflows the buffer, causing unpredictable behavior.

**Mitigation**



```cpp
#include <iostream>
#include <iomanip>

void saferFunction()
{
    // Increased buffer size
    char buffer[20];
    std::cout << "Enter your name: ";

    // Limit input length
    std::cin >> std::setw(20) >> buffer;
    std::cout << "Hello, " << buffer << "!" << std::endl;
}

int main()
{
    saferFunction();
    return 0;
}
```

This example uses 2 mitigation strategies.

- The buffer size is increased to 20 characters, reducing the risk of overflow.
- **std::setw()** is used to limit the input length to prevent overflowing the buffer.
- Input validation could also have been used to prevent more characters input than the buffer can hold.

## Assignment Submission

- Attach tutorials and screenshots of successful runs to assignment in BlackBoard.