HOMEWORK 1

Zehra Kolat, 206001007

March 8, 2025

1 Array Indexing Attack

The following C++ code demonstrates an array indexing vulnerability. The vulnerability occurs when an array is accessed outside of its bounds, leading to potential memory corruption:
Vulnerable Code:

```cpp
#include <iostream>
using namespace std;

void arrayIndexingAttack() {
    int arr[5] = {1, 2, 3, 4, 5};
    cout << "BeforeAttack:" << arr[0] << endl;
    arr[10] = 100;  // Writing outside of array bounds
    cout << "AfterAttack:" << arr[0] << endl;
}

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

Effect on Memory: arr[10] is actually trying to access unallocated memory in array arr. Possible output: The outputs may vary depending on the system, because writing outside array bounds leads to undefined behavior. Example outputs could be:
`1.Normal behavior`
Before attack: 1
After attack: 1
(Overwriting the memory had no effect.)
`2.Unexpected behavior`
Before attack: 1
After attack: 347865
(The value of arr[0] was accidentally changed in memory.)
`3.Program crash (Segmentation Fault):`
Before attack: 1
Segmentation fault (core dumped)
(The program crashed because the accessed memory area was unusable.)
The result: data corruption, unexpected behavior, and program crashes.

Security Risks of This Code

- **No bounds checking on arrays:** Accessing `arr[10]` is out of bounds, leading to undefined behavior.
- **Memory corruption risk:** Writing to out-of-bounds memory can overwrite critical data.
- **Potential attack vector:** If this is based on user input, an attacker could exploit this vulnerability to manipulate memory.

Solution Suggestions

- Check array bounds.

- Prevent invalid index accesses.

Mitigated (Secure) Code:

```
#include <iostream>
using namespace std;

void safeArrayUsage() {
    int arr[5] = {1, 2, 3, 4, 5};
    int index;
    cout << "EnterIndex(0-4):";
    cin >> index;

    if (index >= 0 && index < 5) {
        cout << "Value:" << arr[index] << endl;
    } else {
        cout << "InvalidIndex!" << endl;
    }
}

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

The size of this array is 5, so valid indices are arr[0] to arr[4]. The program prompts the user to input an index and stores the input in the index variable. The program checks if the entered index is valid (in if statement): If the index is between 0 and 4 (inclusive), it accesses the value at that index in the array and prints it to the screen. If the index is outside the valid range, it prints "Invalid index!" to notify the user of an invalid input. Finally, the main() function calls the safeArrayUsage() function. Program Flow in summary, The program prompts the user to enter an index and checks if the index is valid. For example, if the user input is 3, the output is the value of the 3rd index of the array. Otherwise, the user input is 7, the output is "Invalid index!". The code provides security by performing bounds checking on user input to prevent out-of-bounds access. `Reasoning Behind the Mitigation:` Limit checks have been added to user input. Invalid index accesses have been prevented.

2 Pointer Subterfuge Attack

This attack demonstrates a pointer manipulation attack, where a pointer is redirected to an arbitrary memory address and then modified, which could lead to memory corruption or undefined behavior.

Vulnerable Code:

```
#include <iostream>
using namespace std;

void pointerAttack() {
    int x = 10;
    int *ptr = &x;
    cout << "BeforeAttack:" << *ptr << endl;

    ptr = (int*) 0x12345678;  // Assigning to a random memory address
    *ptr = 20;  // This may cause memory corruption!

    cout << "AfterAttack:" << *ptr << endl;
}

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

Effect on memory: An integer x is initialized to 10. A pointer *ptr* is assigned the address of x ($\&x$), so it points to x. The value at the address pointed to by *ptr* (which is x) is printed, displaying Before attack:10. Next, The pointer ptr is manipulated to point to a random memory address 0x12345678. Then, the value at this new address is changed to 20. Since 0x12345678 is not a valid address that the program has allocated, this results in undefined behavior and could potentially corrupt memory, crash the program, or cause other unpredictable outcomes. The program attempts to print the value stored at the ptr, but since the memory is not valid, this could result in a crash or unpredictable behavior.

Potential Issues

- **Memory Corruption:** By redirecting ptr to an arbitrary address, the program writes to a random location in memory, which may corrupt the memory space of the program or other parts of the system.
- **Undefined Behavior:** The code performs an operation that could lead to unpredictable results, including crashes, because the program is accessing an address that is not assigned by the system or within the program's allocated space
- **Security Risk:**This type of attack is common in exploiting vulnerabilities where an attacker may redirect pointers to malicious locations to alter memory, potentially gaining control over the program.

Solution Suggestions:

- Null pointer check.
- Avoid Pointer Manipulation.
- Use Smart Pointers like *unique_ptr*

Mitigated (Secure) Code:

```
#include <iostream>
using namespace std;

void safePointerUsage() {
    int x = 10;
    int *ptr = &x;

    if (ptr != nullptr) {
        cout << "Value:" << *ptr << endl;
    } else {
        cout << "NullPointerDetected!"<< endl;
    }
}

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

Checking whether the pointer is valid or not & preventing the pointer from being redirected to arbitrary addresses. `Reasoning Behind the Mitigation:` This code provides a basic null pointer check and avoidance of pointer manipulation to ensure pointer safety. For further security we can use smart pointers or use different tools. This mitigated code ensures safe memory handling and performing necessary checks before accessing memory, significantly reducing security vulnerabilities related to pointer manipulation.

3 ARC (Absolute Return Call) Injection

This attack demonstrates a function pointer attack vulnerability, where the program allows a user to supply an arbitrary function address to be executed, which could lead to arbitrary code execution if the user provides a malicious address.

Vulnerable Code:

```
#include <iostream>
using namespace std;

void secretFunction() {
    cout << "Hacked!ArbitraryCodeExecutionAchieved!" << endl;
}

void vulnerableFunction() {
    void (*funcPtr)();
    funcPtr = nullptr;

    cout << "EnterFunctionAddress:";
    cin >> reinterpret_cast<void*&>(funcPtr);

    funcPtr();  // If the user inputs a malicious address, an attack will
        occur!
}

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

Effect on memory:

A function pointer named funcPtr is initially set to nullptr. This pointer will hold the address of a function which will be called later. The program prompts the user to input a function address. It is assigned to the function pointer using reinterpret_cast <void*& >(funcPtr). This allows the user to enter any memory address.After obtaining the function address, the program attempts to call the function via funcPtr. If the user provides a malicious address, this could execute arbitrary code, potentially causing a buffer overflow or executing the secretFunction() or other injected code.

Potential Issues

- The vulnerableFunction() allows a user to provide a function pointer address, which could point to malicious code. This is a security vulnerability known as arbitrary code execution.
- In this case, an attacker could enter the address of secretFunction(), causing it to be executed, which prints a message: Hacked! Arbitrary code execution achieved!.
- If the user provides a different function address, the program will call that function, and if the attacker provides an address in a writable and executable region of memory, arbitrary code could be executed, leading to potentially dangerous consequences like system compromise.

Solution Suggestions

- **Validate User Input:**Instead of accepting arbitrary function addresses from user input, validate the address or prevent user input from influencing memory addresses.

Mitigated (Secure) Code:

```
#include <iostream>
using namespace std;

void safeFunctionCall() {
    void (*funcPtr)() = nullptr;

```

```
    if (funcPtr != nullptr) {
        funcPtr();
    } else {
        cout << "InvalidFunctionPointer!" << endl;
    }
}

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

This code demonstrates how to mitigate the function pointer attack vulnerability by performing a null pointer check before calling the function via the pointer. Another way to say invalid function pointers are checked before being called. **Reasoning Behind the Mitigation:** The function pointer funcPtr is initialized to nullptr, meaning it doesn't point to any function at first.Before attempting to call the function, the code checks whether funcPtr is not null. If it is null, the program will print "Invalid function pointer!" instead of calling the function. This ensures that no function is called unless the pointer points to a valid function.It ensures that the function pointer is valid before calling it. By adding the null check, it protects the program from accessing invalid memory locations and prevents malicious exploitation via function pointer manipulation.

4 Stack Smashing

This occurs when data overwrites adjacent memory regions beyond the allocated memory space, which could allow attackers to execute arbitrary code.
Vulnerable Code:

```
#include <iostream>
#include <cstring>
using namespace std;

void stackSmashingAttack() {
    char buffer[8];
    cout << "EnterInput:";
    cin >> buffer;  // If the user inputs a long string, overflow occurs.
}

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

Effect on memory: A character array buffer of size 8 is declared, meaning it can hold up to 7 characters plus a null terminator (/0) for the string. If the user enters more than 7 characters (8 including the null terminator), it will overflow into adjacent memory, potentially corrupting the stack and causing unintended behavior. For example, when my input was a word containing 9 characters, I got an output like this: "*** stack smashing detected ***: terminated" **Potential Issues:** This attack occurs when an attacker inputs data that exceeds the size of the buffer, which can change the return address of the function or other control information. This could allow the attacker to redirect the program's flow, execute arbitrary functions, or cause undefined behavior. The control flow of the program can be changed.

Solution Suggestions:

- Limit Input Size

Mitigated Code:

```
#include <iostream>
```

```cpp
#include <cstring>
using namespace std;

void safeStackUsage() {
    char buffer[8];
    cout << "EnterInput:";
    cin.width(8);  // Limit input to a maximum of 7 characters (the last one
        is reserved for null terminator)
    cin >> buffer;
}

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

cin.width(8), is used to limit the number of characters the user can input, preventing a buffer overflow. **Reasoning Behind the Mitigation:** The code now safely handles user input, avoiding the potential security vulnerability posed by the original code. Mitigating these risks involves validating input lengths and using secure coding practices like input bounds checking to prevent overflows.

5 Heap Smashing

Similar to a stack smashing, a heap smashing occurs when data overflows beyond the allocated memory on the heap. It can lead to the corruption of adjacent memory, potentially allowing an attacker to manipulate the program's behavior.

Vulnerable Code:

```cpp
#include <iostream>
#include <cstring>
using namespace std;

void heapSmashingAttack() {
    char *buffer = new char[8];
    cout << "EnterInput:";
    cin >> buffer;  // Potential overflow!
    delete[] buffer;
}

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

Effect on memory: If the user enters more than 7 characters (8 including the null terminator), it will overflow the allocated memory on the heap, potentially corrupting adjacent memory regions.After the user input is processed, the buffer is deallocated using delete[ ]. However, if a heap overflow occurred, the adjacent memory could be corrupted, and the program might behave unpredictably or cause memory issues.

Potential Issues: If an attacker provides more than 7 characters, the data will overflow into adjacent memory locations. In some cases, this could overwrite important data, such as control structures, which could lead to arbitrary code execution or other unintended behaviors. For example, an attacker might be able to manipulate the heap to redirect function pointers or overwrite variables, which could result in malicious code execution or a crash.Other variables in memory may be overwritten. Result: Data corruption or arbitrary code execution.

Solution Suggestions:

- Limit Input Size
- Prevent memory leaks

Mitigated Code:

```
#include <iostream>
#include <cstring>
using namespace std;

void safeHeapUsage() {
    char *buffer = new char[8];
    cout << "EnterInput:";
    cin.width(8);  // Limit input to a maximum of 7 characters (the last one
        is reserved for null terminator)
    cin >> buffer;
    delete[] buffer;
}

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

Input is limited with cin.width(8). (Instead of using cin.width(), you could use also cin.getline() to control input size.) This means that only 7 characters can be entered before the program stops reading input (the 8th character position is reserved for the null terminator). This helps prevent a buffer overflow by ensuring that no more than the allocated memory (8 bytes in total) is written to buffer. Delete[ ], is used to prevent memory leaks.

Reasoning Behind the Mitigation: this is an important first step in ensuring the program handles user input safely and protects against buffer overflow vulnerabilities. Additionally, considering modern practices such as using dynamic containers like std::vector can further reduce risks and improve overall safety.

Teacher, thank you for patiently reading my homework. I hope it was the homework you wanted. Sorry if it's too long.