Discussion-2 Focus on C/C++ Securely Working with Strings

Discussion Topic:

Working with strings can lead to various security flaws and errors in software development using the C++ language.

What are the common string manipulation errors that can be encountered?
How can these errors be resolved and/or limited?
What tips can be utilized to identify security vulnerabilities related to strings in C++?

Be sure to provide an appropriate source code example to illustrate your points.

My Post:

Hello Class,

In software development, working with strings is notoriously error-prone and can generate security vulnerabilities, especially in C++. C++ is a flexible programming language that allows manual memory management and when combined with the primitive nature of strings, it is very easy for developers to accidentally introduce errors and dangerous vulnerabilities when manipulating strings.

Below is a list of common string-manipulating errors:

- **Buffer Overflows:** one of the most common errors leading to significant security vulnerabilities. This happened usually when copying data to a buffer that is not large enough to hold that data (Ballman, 2016), especially when functions such as '`strcpy()`', which does not check the length of the destination when coping an array of characters. A solution is to use functions like '`strncpy()`', which limit the number of characters copied; however using '`sdt::string`' is often a better solution as it is dynamic length, meaning is not a bounded array and it is no need to manage the null terminators '`\0`' Making sure that a destination string buffer is compliant with the length of the string stored is compliant with the SEI CERT C++ CODING STANDARD STR50-CPP "**Guarantee that storage for strings has sufficient space for character data and the null terminator**" (SEI External WIKI, n.d., STR50-CPP).

See the code examples below:

```
// Example 1: Buffer Overflow (STR50-CPP)
void bufferOverflowExample() {
    char buf[12];
    // Error: Buffer overflow if input is longer than 12 characters
    std::cin >> buf;  // Solution: Use std::string instead
}
```

```
void bufferOverflowSolution() {
    std::string input;
    std::cin >> input;  // Using std::string avoids overflow
}
```

- **Null-Terminator:** When C-style strings are used '`char[]`' not allocating the null-terminator '`\0`' at the end of the character array may result in undefined behavior and memory corruption, as well as creating vulgarities. Without a null terminator, functions may read out-of-bounds memory and can be exploited for nefarious purposes. Thus, always add '`\0`', if the string is 9 characters long the character array should have a length of 10 to accommodate '`\0`' at the end of it. This is part of STR50-CPP.

See the code examples below:

```
// Example 2: Null-Terminator Issue (STR50-CPP)
void nullTerminatorIssue() {
    char buf[12] = {'p', 'a', 'n', 'd', 'a', 's'}; // Missing null terminator
    std::cout << buf << std::endl;  // Undefined behavior
}

void nullTerminatorSolution() {
    char buf[10] = {'p', 'a', 'n', 'd', 'a', 's', '\0'}; // Add null terminator
explicitly
    std::cout << buf << std::endl;
}
```

- **Unbounded String Input:** When C-style strings are used '`char[]`', always limits the input size, that is when using '`std::cin`' to the length of the character array receiving the inputs to the length of the array – 1 (to accommodate '`\0`'). Failing to do so may cause data overflows. This part of STR50-CPP.

See the code examples below:

```
// Example 3: Unbounded Input (STR50-CPP)
void unboundedInputIssue() {
    char buf[10];
    std::cin >> buf;  // No limit on input size, may cause overflow
}

void boundedInputSolution() {
    char buf[10];
    std::cin.width(10);
    std::cin >> buf;  // Limits input to 9 characters
}
```

- **Strings Null Pointers:** Do not create a '`std::string`' from a null pointer, for example, using '`std::getenv()`', that is getting an environment variable without checking if the variable exists, this may result in a null pointer if the environment variable does not exists. Allocating the value of a null pointer to a '`std::string`' may lead to undefined behavior and dereferencing a null pointer, that is trying to access an invalid memory address. Making sure that a 'std::string' is not created from a null pointer is

compliant with the SEI CERT C++ CODING STANDARD STR51-CPP "**Do not attempt to create a std::string from a null pointer**" (SEI External WIKI, n.d., STR51-CPP).

See the code examples below:

```
// Example 4: Null Pointer with std::string (STR51-CPP)
void nullPointerStringIssue() {
    const char *envVar = std::getenv("LINK_TO_NOWHERE");
    std::string str(envVar);  // Undefined behavior if envVar is nullptr
}

void nullPointerStringSolution() {
    const char *envVar = std::getenv("LINK_TO_NOWHERE");
    std::string str(envVar ? envVar : "");  // Safely handle nullptr
}
```

- **String Invalid Iterator Usage:** Making sure that a string iterator is still valid after modifying a string is crucial as it can result in the iterator not reflecting changes to the string properly. This often occurs when using the '`insert()`' or '`replace()`' methods. Thus, revalidating the iterator by relocating the value of the string after each string modification is essential. Making sure that the string iterator is still valid after modifying a string is compliant with the SEI CERT C++ CODING STANDARD STR52-CPP "**Use valid references, pointers, and iterators to reference elements of a basic_string**" (SEI External WIKI, n.d., STR52-CPP).

See the code examples below:

```
// Example 5: Invalid Iterator Usage (STR52-CPP)
void invalidIteratorIssue() {
    std::string input = "example";
    std::string::iterator it = input.begin();
    input.insert(it, 'E');  // Insert invalidates iterators
    ++it;  // Undefined behavior - iterator has been invalidated
}

void validIteratorSolution() {
    std::string input = "example";
    std::string::iterator it = input.begin();
    it = input.insert(it, 'E');  // Update iterator to avoid invalidation issues
    ++it;
}
```

- **String Unchecked Range Access:** Do not access elements outside the valid range of a string, this often occurs when using functions like '`front()`' and '`back()`', this can lead to undefined behavior. Thus, always perform range checks before trying to access an element, or use safer functions like '`at()`' that throw exceptions on out-of-range access. Not accessing elements outside the valid range of a string is compliant with the SEI CERT C++ CODING STANDARD STR53-CPP "**Range check element access**" (SEI External WIKI, n.d., STR53-CPP).

See the code examples below:

```
// Example 6: Bounds Checking with std::string (STR53-CPP)
void boundsCheckingIssue() {
    std::string s = "example";
    char c = s[10];  // Undefined behavior if index is out of bounds
}

void boundsCheckingSolution() {
    std::string s = "example";
    try {
        char c = s.at(10);  // Throws std::out_of_range if index is out of bounds
    } catch (const std::out_of_range &e) {
        std::cerr << "Out of range error: " << e.what() << std::endl;
    }
}
```

The table below shows a summary of the risk assessment of the C++ string-related coding rules from the SEI CERT C++ Coding Standard.

Table 1

String Rules Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR50-CPP	High	Likely	Medium	P18	L1
STR51-CPP	High	Likely	Medium	P18	L1
STR52-CPP	High	Probable	High	P6	L2
STR53-CPP	High	Unlikely	Medium	P6	L2

Note: From "Rule 05. Characters and Strings (STR). *SEI CERT C++ Coding Standard*" by SEI External WIKI. (n.d.)

Table Description:

- Rule(s) are the rules from rules from the SEI CERT C++ Coding Standard.
- Severity refers to the potential impact on security.
- Likelihood refers to how likely it is that violating the rule will result in a security vulnerability.
- Remediation Cost refers to the amount of effort needed to fix the violation.
- Priority (P) refers to the priority level at which a violation should be addressed, the lower the numbers the higher the priority.
- Level refers to the level of enforcement, that is how strictly the coding standard needs to be followed to not introduce vulnerabilities.

To summarize, working with strings is notoriously error-prone and may cause security vulnerabilities, especially in C++. C++ is a flexible programming language that allows manual memory management, which increases the risk of introducing errors and vulnerabilities. However, by following the SEI CERT C++ Coding Standards, C++ developers can generate secure and reliable code when manipulating strings.

-Alex

References:

SEI External WIKI. (n.d.) Rule 05. Characters and Strings (STR). *SEI CERT C++ coding standard*. Carnegie Mellon University. Software Engineering Institude. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046330

Ballman, A. (2016). Chapter-6 Characters and Strings (STR). *SEI CERT C++ coding standard: Rules for developing safe, reliable, and secure systems in C++*. Software Engineering Institute (Carnegie Mellon University). Hanscom, MA.