**Module 2: Buffer Overflow**

Zane Milo Deso

Southern New Hampshire University

CS-405: Secure Coding

**Buffer Overflow Post-Mortem**

**Introduction**

As a Computer Science undergraduate attending Southern New Hampshire University (SNHU), I have been tasked with summarizing the completion of this assignment. In this paper, I will briefly outline buffer overflow, elaborate on my approach to the prevention of buffer overflow, why it works, problems encountered, and how I ameliorated the issues.

**Buffer Overflow**

Buffer overflow occurs when the range of the expected input falls out of the range of the container. More specifically, the input data exceeds the allocated memory for the container and erroneously overwrites into an adjacent memory address. This can lead to unexpected behavior, crashes, errors, etc.

**Approach to Detect and Prevent Buffer Overflow**

At first, I considered a few different angles that a buffer overflow may be prevented and detected. I opted to check the input length in a while loop, only allowing the correct amount of data to escape the while loop, continuing into the rest of the program.  
 Additionally, I identified that a length check and while loop did not fully prevent the buffer overflow. Therefore, after further research, I managed to adjust the code to include <limits> allowing me to leverage setw(). Along with carefully using .peek(), and .ignore(), the three were able to catch and prevent buffer overflow. The setw() function allows for only the specified amount (in this case 20) characters, including the null terminator to be read from the buffer. The .peek() function allows the program to safely look at, but not consume, the next character in the input, effectively checking if the input is larger than the maximum allowed. Lastly, the .ignore() function clears the buffer of any possible remaining characters it may hold.

**Why does the Approach work?**

The approach is actionable since the range of acceptable inputs has now been limited to the acceptable range of the allocated memory. This limited range disallows any input that can cause a buffer overflow.

**Problems Encountered & Overcoming the Obstacle**

The only problem that I encountered was when I attempted to use .size() on the input variable. This caused an error to be thrown, which was due to the variable not being a class object. This makes sense, so I fixed the problem by instead setting a new variable to capture the input length, accomplished via strlen().

Furthermore, I attempted to only use strlen() with a while loop, which caught when inputs were more than they should be for the initial loop, though the buffer was never cleared. Since the buffer was never cleared the additional loop inputs were truncated into the buffer still causing the buffer overflow.

Overcoming these obstacles involved additional research revealing a safer manner to handle the security vulnerability. The solution involved using a trifecta of functions, namely .setw(), .peek(), and .ignore(). Where .setw() allows for only a prescribed amount of characters to be read from the buffer, .peek() safely checks for stragglers, and .ignore() flushes the remaining characters from the buffer. Altogether, they work great to prevent the overflow of the buffer.

**Conclusion**

In conclusion, buffer overflows are a major threat that if left unchecked may alter the behavior of programs and even cause crashes. In production environments, this can be catastrophic and should be avoided at all costs. By leveraging the <limits> class template, and functions .setw(), .peek(), and .ignore(), developers and security professionals can mitigate risk of the fallout of buffer overflow.