**Buffer Overflow**

CPSC 329 Project about buffer overflow, including how it works and techniques attackers use to successfully exploit these vulnerabilities.

**What is a Buffer?**

A buffer is a sequential section of memory allocated to hold input data. It can hold anything from a character string to an array of integers.

**What is a Buffer Overflow?**

A Buffer Overflow is an attack against a common software caused by coding mistake that happens when more data is put into a fixed-length buffer than the buffer can handle. The extra information can overflow into adjacent memory space, corrupting or overwriting the data held in that space on memory. This overflow usually results in a system crash and may change the execution path of the program. This can result in a response that damages files or exposes private information or even give root access to malicious users.

**Programming Languages Most Vulnerable to Buffer Overflow**

Many programming languages are prone to buffer overflow attacks. Languages like Java is generally not susceptible to buffer overflows as they would throw an exception and crash the program. However, programs written in C, C++, Fortran or Assembly could allow the attacker to fully compromise the targeted system.

The most common language for systems programming – C/C++, since they often do not provide any built-in protection against accessing or overwriting data in any part of memory. Bounds checking can prevent buffer overflows, but requires additional code and processing time.

**Types of Buffer Overflow attacks**

The techniques to exploit a buffer overflow vulnerability vary by architecture, operating system and by memory region. The two types of buffer overflow attacks:

1. **Stack-based buffer overflows**

This is the more common type of buffer overflow that attack stack memory which only exists during the execution of a function. It can be done in several ways:

* By overwriting a local variable that is located near the vulnerable buffer on the stack, in order to change the behavior of the program.
* By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker.
* By overwriting a function pointer or exception handler, which is subsequently executed.
* By overwriting a local variable (or pointer) of a different stack frame, which will be used by the function which owns that frame later.

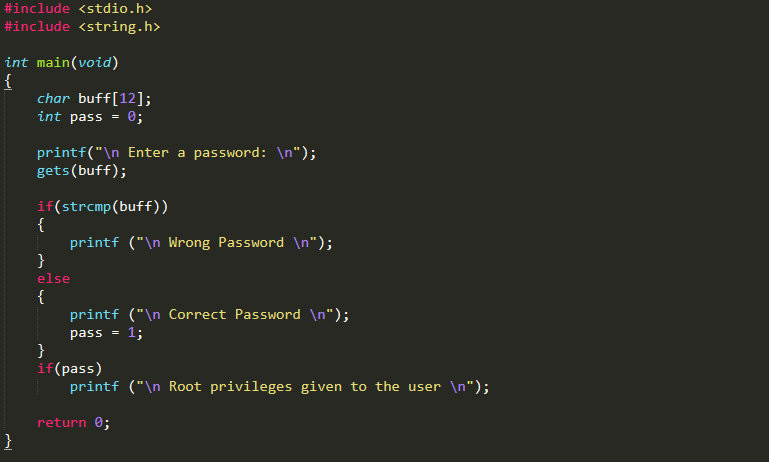


1. **Heap-based exploitation**

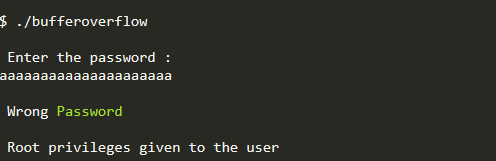
A buffer overflow occurring in the heap data area is referred to as a heap overflow and is exploitable in a different way compared to that of stack-based overflows. Memory on the heap is allocated dynamically by the application during run-time and typically contains program data. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc meta data) and uses the resulting pointer exchange to overwrite a program function pointer.

**How Buffer Overflow Works**

The method for exploiting a stack-based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data. This can be shown using the following example:



The example code lets a user input a password from the user and if the password is correct, then it grants root privileges to the user. However, this code is vulnerable to buffer overflow. The code allocates 12 bytes to store the password typed in by the user. However the gets() function does not check the how many characters has been inputted by the user. So, an attacker can write a string of length greater than the size of the buffer and overflow it, causing the ‘pass’ variable to store a value not equal to 0. This will result in giving the attacker root access even though they did not know the password.



**Stack Canaries**

Stack canaries are used to detect a stack buffer overflow before execution of malicious code can occur. This method works by placing a small integer which is randomly chosen in memory just before the return instruction pointer on stack. In order to overwrite the return pointer and take control of the process, the canary value must be overwritten first. Right before the function return, the small integer value (canary) is checked to make sure it has not been altered. This technique can greatly increase the difficulty of exploiting a stack buffer overflow because it forces the attacker to guess the random canary placed before the saved instruction pointer.



**Format String Attack**

The format String exploit happens when the attacker inputs a string that is evaluated as some by the program. This can potentially leak values on the stack or cause segmentation fault. For example, if a Format String parameter like %x is inserted into the posted data, the string is parsed by the Format Function such as ‘printf’ and parameters are executed. However, the format function is expecting more arguments as input and if these arguments are not provided, the function could read or write the stack.

**Common Parameters used in a Format String Attack**

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Output** | **Passed as** |
| %% | % character (literal) | Reference |
| %p | External representation of a pointer to void | Reference |
| %d | Decimal | Value |
| %c | Character |  |
| %u | Unsigned decimal | Value |
| %x | Hexadecimal | Value |
| %s | String | Reference |
| %n | Writes the number of characters into a pointer | Reference |

ByAhmed Farazi & Mushfik Rahman

**References and image credits**

* <https://www.veracode.com/security/buffer-overflow>
* <https://en.wikipedia.org/wiki/Buffer_overflow>
* <https://www.imperva.com/learn/application-security/buffer-overflow/>
* <https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/>
* <https://en.wikipedia.org/wiki/Buffer_overflow>
* <https://www.thegeekstuff.com/2013/06/buffer-overflow/>
* <https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries>