**Buffer-overflow**

**Description 1**

Buffer overflow errors are characterized by the overwriting of memory fragments of the process, which should have never been modified intentionally or unintentionally. Overwriting values of the IP (Instruction Pointer), BP (Base Pointer) and other registers causes exceptions, segmentation faults, and other errors to occur. Usually these errors end execution of the application in an unexpected way. Buffer overflow errors occur when we operate on buffers of char type. Buffer overflows can consist of overflowing the stack (Stack overflow) or overflowing the heap (Heap overflow). [1]

Attackers generally use buffer overflows to **corrupt the execution stack of a web application**. By sending carefully crafted input to a web application, an **attacker can cause the web application to execute arbitrary code, possibly taking over the machine**. [2] (you can find more examples in [2])

**Description 2**

Buffer overflow attack uses input to a poorly implemented, but (in intention) completely harmless application, typically with root / administrator privileges. The buffer overflow attack results from input that is longer than the implementor intended. To understand its inner workings, we need to talk a little bit about how computers use memory.

The stack is a region in a program's memory space that is only accessible from the top. There are two operations, push and pop, to a stack. A push stores a new data item on top of the stack, a pop removes the top item. Every process has its own memory space, among them a stack region and a heap region. The stack is used heavily to store local variables and the return address of a function.

For example, assume that we have a function

**void foo(const char\* input) {**

**char buf[10];**

**printf("Hello World\n");**

**}**

When this function is called from another function, for example main:

**int main(int argc, char\* argv[])**

**{**

**foo(argv[1]);**

**return 0;**

**}**

then the following happens: The calling function pushes the return address, that is the address of the return statement onto the stack. Then the called function pushes zeroes on the stack to store its local variable. Since foo has a variable buf, this means there will be space for 10 characters allocated. The stack thus will look like depicted in the following Figure. [3]



A real attack would try to place the address of the top of the stack in lieu of the return address, followed by some horrible lines of assembly code, such as a call to another tool. If the subverted program runs with high privileges, then the tool will run with the same privilege level. Even better for the attacker, the whole process takes only the transmission of a little script program.

**Description 3**

There are two types of buffer overflow. Common Weakness Enumeration [4] explains each of the separately:

**Stack-based Buffer Overflow [4]**

A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

**Common Consequences [4]**

* Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
* Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.
* When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

**Heap-based Buffer Overflow [5]**

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().

**Common Consequences [5]**

* Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
* Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. Besides important user data, heap-based overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.
* When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

**Reference**

**[1]** https://www.owasp.org/index.php/Buffer\_Overflow

**[2]** https://www.owasp.org/index.php/Buffer\_Overflows

**[3]** <http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html>

**[4]** [**https://cwe.mitre.org/data/definitions/121.html**](https://cwe.mitre.org/data/definitions/121.html)

**[5]** [**https://cwe.mitre.org/data/definitions/122.html**](https://cwe.mitre.org/data/definitions/122.html)