Chapter 10: Buffer Overflow Attacks and Others

A buffer overflow, also known as a buffer overrun or buffer overwrite, is defined in the NIST *Glossary of Key Information Security Terms* as: "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

1. **Buffer Overflow Basics**
    a. **The basic concept** - A buffer overflow can occur as a result of a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer and consequently overwrites adjacent memory locations.
    b. **The location** - The buffer could be located on the stack, in the heap, or in the data section of the process.
    c. **The consequences** – Corruption of data used by the program, unexpected transfer of control in the program, possible memory access violations, and very likely eventual program termination.

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

2. **Buffer Overflow Example**

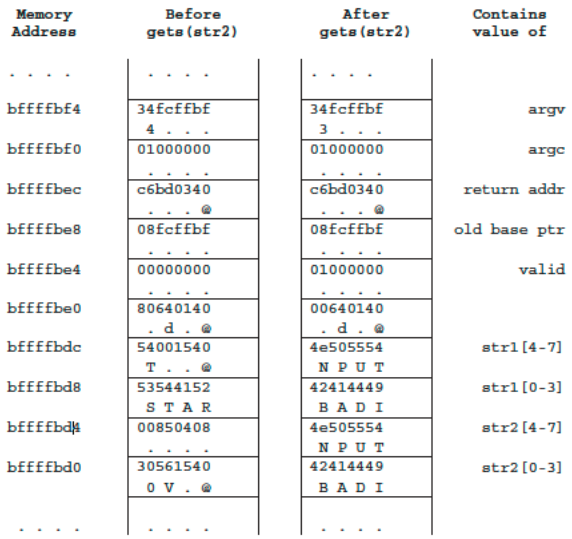
```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

This contains three variables (valid, str1, and str2), whose values will typically be saved in adjacent memory locations.

The order and location of these will depend on the type of variable (local or global), the language and compiler used, and the target machine architecture.

In this example it assumed that the variables are saved in consecutive memory locations, from highest to lowest. This will typically be the case for local variables in a C function on common processor architectures such as the Intel family.



Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

The purpose of the function next_tag (str1) to copy into str1 some expected tag value. Let's assume this will be the string START. It then reads the next line from the standard input for the program using the C library gets function and then compares the string read with the expected tag. If the next line did indeed contain just the string START, this comparison would succeed, and the variable VALID would be set to TRUE. Any other input tag would leave it with the value FALSE

The traditional C library gets() function does not include any checking on the amount of data copied. It will read the next line of text from the program's standard input up until the first newline character occurs and copy it into the supplied buffer followed by the NULL terminator used with C strings.

If more than seven characters are present on the input line, when read in they will (along with the terminating NULL character) require more room than is available in the str2 buffer. Consequently, the extra characters will proceed to overwrite the values of the adjacent variable, str1 in this case.

EVILINPUTVALUE, the result will be that str1 will be overwritten with the characters TVALUE, and str2 will use not only the eight characters allocated to it but seven more from str1 as well.

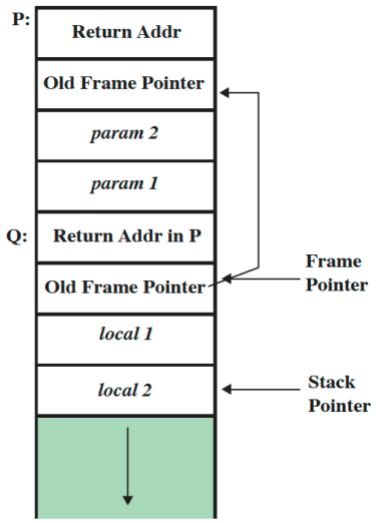
The overflow has resulted in corruption of a variable not directly used to save the input. Because these strings are not equal, valid also retains the value FALSE.

If 16 or more characters were input, additional memory locations would be overwritten.

3. **Buffer Overflow Exploitation Requirements**
    a. **The Attacker**
        i. Needs to identify a buffer overflow vulnerability in some program that can be triggered using external data.
        ii. Needs to understand how that buffer will be stored in the processes, in memory, and determine the corruption potential.
        iii. Needs to alter the flow of the program execution

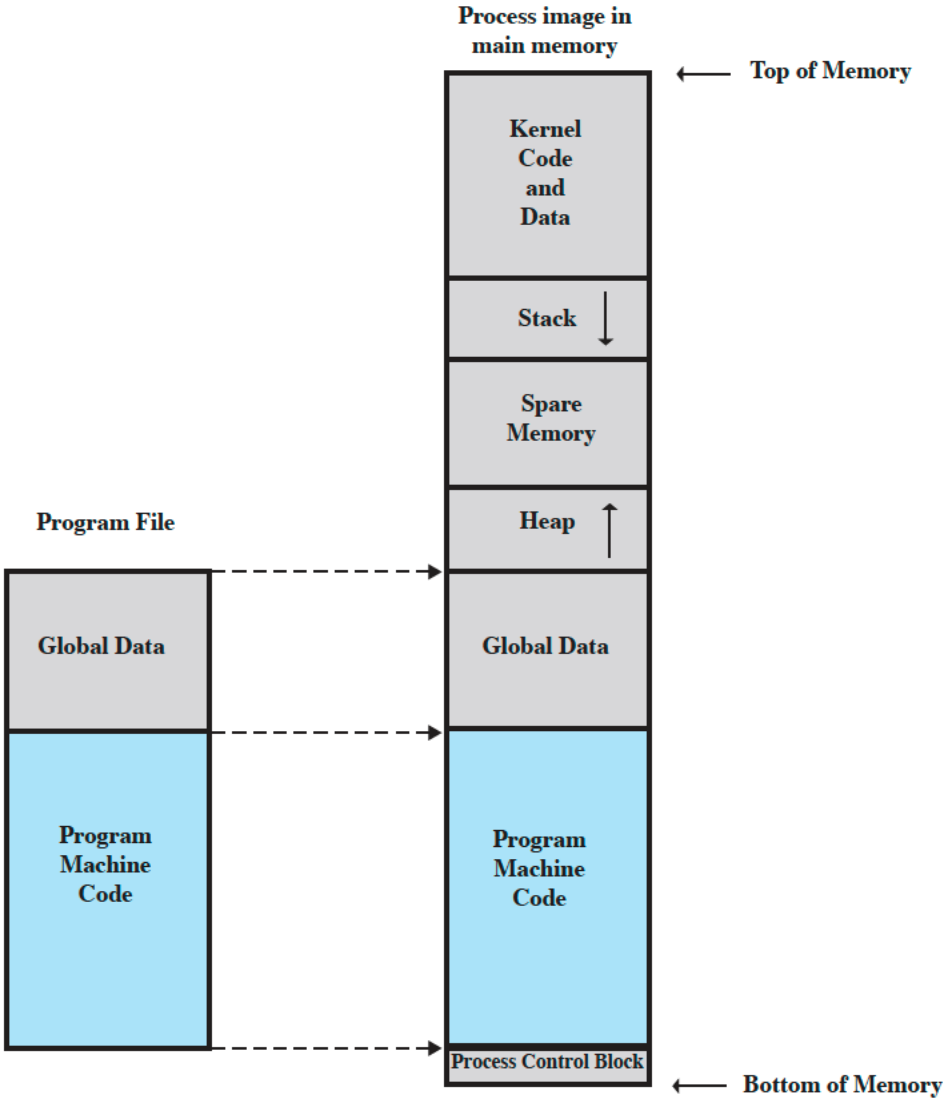
    b. **Identifying Vulnerable Programs**
        i. Inspect the program's source code
        ii. Trace the execution using oversized input
        iii. Use tools such as fuzzing to identify vulnerable programs
            1. This is a software testing technique that uses randomly generated data as inputs to a program.
            2. The intent is to determine whether the program or function correctly handles all such abnormal inputs or whether it crashes or otherwise fails to respond appropriately.

4. **Stack Buffer Overflows**
    a. **Stack Buffer Overflow Basics**
        i. A stack buffer overflow occurs when the targeted buffer is located on the stack.
            1. Usually as a local variable
        ii. Referred to as stack smashing
    b. **The Stack**
        i. When one function calls another, the stack needs to save the return address for when the function completes.
        ii. The program needs to return to the calling function,
        iii. Also needs to maintain the parameters passed in
        iv. Saved on the stack in a structure known as a stack frame.
        v. Ability to chaining these stack frames together, so that as a function is exiting it can restore the stack frame for the calling function before transferring control to the return address.



c. **The Process** - The general process of one function P calling another function Q can be summarized as:
    i. The calling function P
        1. Pushes the parameters for the called function onto the stack (typically in reverse order of declaration)
        2. Executes the call instruction to call the target function, which pushes the return address onto the stack
    ii. The called function Q
        1. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack

d. The Big Picture



i. The core of a stack overflow attack comes from the possibility of overwriting the saved frame pointer and return address.

ii. The local variables are placed below the saved frame pointer and return address, the possibility exists of overwriting the values of one or both of these key function linkage values.

iii. The local variables are usually allocated space in the stack frame in order of declaration, growing down in memory with the top of stack. Compiler optimization can potentially change this, so the actual layout will need to be determined for any specific program.

iv. When a program is run, the operating system creates a new process for it. The process is given its own virtual address space, with the general structure above
    1. This consists of the contents of the executable program file (including global data, relocation table, and actual program code segments) near the bottom of this address space.
    2. The space for the program heap to then grow upward from above the code.
    3. Room for the stack to grow down from near the middle or top.

5. Actual Windows Buffer Overflow Example