A Study of Buffer Overflow Attacks

M.Rupasri, P.Rajyalakshmi, V.Sangeeta

Abstract— A computer buffer is an area of memory used for temporary storage of data when a program or hardware device needs an uninterrupted flow of information. A buffer overflow occurs when a program or process tries to store more data in a buffer than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information - which has to go somewhere - can overflow into adjacent buffers, corrupting or overwriting the valid data held in them. In buffer overflow attacks, the extra data may contain codes designed to trigger specific actions, in effect sending new instructions to the attacked computer that could, for example, damage the user's files, change data, or disclose confidential information. This paper presents an overview of the buffer overflow attack and the countermeasures to defend that attack.

Index Terms—Buffer overflow, Function pointers, Heap overflow, Stack overflow.

I. INTRODUCTION

Buffer overflows make up one of the largest collections of vulnerabilities in existence; And a large percentage of possible remote exploits are of the overflow variety. Almost all of the most devastating computer attacks to hit the Internet in recent years including SQL Slammer, Blaster, and I Love You attacks. If executed properly, an overflow vulnerability will allow an attacker to run arbitrary code on the victims machine with the equivalent rights of whichever process was overflowed. This is often used to provide a remote shell onto the victim machine, which can be used for further exploitation[1]. A buffer overflow[2][5] is a term used by programmers to describe the act of writing data past the end of a buffer. A buffer is best thought of as being similar to a bucket. Buffers, like a bucket, can come in a variety of sizes and can contain various quantities of items. However, just like a bucket, buffers have a maximum storage capacity. If this storage capacity is exceeded, the buffer is said to have been overflowed. When such an overflow occurs data seeps past the boundary of the buffer and corrupts its surroundings. Under these conditions it is often possible for an attacker to take control of the data adjacent to the buffer and thus leverage control of the execution path that a program takes. When such a scenario occurs an attacker exploits a buffer overflow by taking advantage of the data seepage. The following illustrates an example of a buffer overflow attack
Let a program has defined two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte integer, B. Initially, A contains nothing but zero bytes, and B contains the number 1979. Characters are one byte wide.

Variable Name	A								B	
Value	NULL String								1979	
Hex value	00	00	00	00	00	00	00	00	07	BB

Now, the program attempts to store the null-terminated string "excessive" in the A buffer. By failing to check the length of the string, it overwrites the value of B:

Variable Name	A								B	
Value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
Hex value	65	78	53	65	73	73	69	75	65	00

Although the programmer did not intend to change B at all, B's value has now been replaced by a number formed from part of the character string. In this example, on a big-endian system that uses ASCII, "e" followed by a zero byte would become the number 25856. If B was the only other variable data item defined by the program, writing an even longer string that went past the end of B could cause an error such as a segmentation fault, terminating the process.

This paper presents a study of how a buffer overflow causes vulnerabilities to the security of information and how it can be defended. Examples used in this paper are written in C programming language.

II. GENERATIONS OF BUFFER OVERFLOW ATTACK

A. First Generation

First generation buffer overflows involve overflowing a buffer that is located on the stack.

Strictly speaking, a stack overflow[3] is what happens when the amount of memory allocated to a program for its call stack is over-filled, causing the program to crash. In general use, particularly in security issues, the term 'stack overflow' refers to a stack buffer overflow, a type of buffer overflow taking place on the call stack. A stack buffer overflow attack is an attempt to exploit a coding vulnerability in software, where the boundaries for data being passed onto the stack are inadequately controlled. This allows malicious data to be written to the stack in such a way that it overwrites other areas, which can then in turn lead to the data written to those areas being executed as program code.

Many major malware attacks have exploited such vulnerabilities, including the hugely widespread Slammer and Blaster worms.

Manuscript received on March 27, 2012.

M.Rupasri, MCA (M.Tech) Pydah Engineering College, A.P., India. (E-mail:m.rupasri@gmail.com).

P. Rajyalakshmi, Asst. Professor, Dept. of CSE, Pydah Engineering College , A.P., India. (E-mail: rajyam_ped2003@yahoo.co.in).

V. Sangeeta. Assoc. Professor & Head, Dept. of CSE, Pydah Engineering College, A.P., India.
(E-mail:sangeetaviswanadham@yahoo.com).

For example, the following program(C language program) declares a buffer that is 256 bytes long. However, the program attempts to fill it with 512 bytes of the letter 'A' (0x41).

```
int i;
void function(void)
{char buffer[256]; // create a buffer
for(i=0;i<512;i++) // iterate 512 times
buffer[i]='A'; // copy the letter A}
```

B. Second Generation

The second generation of buffer overflow attack include off- by- one overflows, heap overflows and function pointers.

Off- by- one Overflow: Exceeding the buffer size by just one byte is called off- by- one overflow attack [4] . Usually off-by-one errors can do no more than crash the program. Consider the following program where the programmer has mistakenly utilized 'less than or equal to' rather than simply 'less than'.

```
#include <stdio.h>
int i;
void vuln(char *foobar)
{char buffer[512];
for (i=0;i<=512;i++)
buffer[i]=foobar[i];}
void main(int argc, char *argv[])
{if (argc==2)
vuln(argv[1]);}
```

Heap Overflow: Heap is an area of memory used for dynamic memory allocation. A heap overflow[5] 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().

The following is an example for heap overflow. The program dynamically allocates memory for two buffers. One buffer is filled with 'A's. The other one is taken in from the command line. If one types too many characters on the command line an overflow will occur.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(int argc, char **argv)
{char *buffer = (char *) malloc(16);
char *input = (char *) malloc(16);
strcpy(buffer,"AAAAAAAAAAAAAAA");
// Use a non-bounds checked function
strcpy(input,argv[1]);
printf("%s",buffer);}
```

Function Pointers: A function pointer occurs mainly when callbacks occur. If in memory, a function pointer follows a buffer, there is the possibility to overwrite the function pointer if the buffer is unchecked. The following is an example that an attacker may use to take control by replacing the function pointer with the address of attack code.

```
void bar( void( *func1)( ) )
{Void (*func2) ( );
    Char buf[128];
    ……
    Strcpy (buf, getenv( "HOME"));
    (*func1) ( );
    (*func2) ( );}
```

C. Third Generation

Third generation buffer overflow attacks include format string vulnerabilities. Format String Attacks[6] are caused from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as printf(). A malicious user may use the %s and %x format tokens, among others, to print data from the stack or possibly other locations in memory. One may also write arbitrary data to arbitrary locations using the %n format token, which commands printf() and similar functions to write back the number of bytes formatted to an argument of type int *. By manipulating the stack by using spurious format tokens, this argument can be faked as part of the format string. The following is an example that illustrates format string attack.

```
int main(int argc, char *argv[])
{char user_input[100];
... ... /* other variable definitions and statements */
scanf("%s", user_input); /* getting a string from user */
printf(user_input); /* Vulnerable place */
return 0;}
```

III. BUFFER OVERFLOW ATTACK PREVENTION

revention of buffer overflow attacks is an easy one if you follow these:

(a) Use trusted libraries when running code

(b) Use of good programming techniques.

(c) Do not write code that is susceptible to overflow attacks.

(d) Check user input for validity and make sure it is in the correct format and within the range of function.

(e) Use trusted releases of programs. Alpha and beta versions have bugs that tend to not prevent these attacks.

IV. CONCLUSION

Buffer overflow occurs anytime the program writes more information into the buffer than the space it has allocated in the memory. This allows an attacker to overwrite data that controls the program execution path and hijack the control of the program to execute the attacker's code instead the process code. So in order to defend the buffer overflow attack input validation, safe libraries, good programming language, stack examination are required.

REFERENCES

1. Buffer Overflow Attacks by James C. Foster, Vitaly Osipov, Nish Bhalla, Niels Heinen and Dave Aitel , Syngress, 2005.
2. http://en.wikipedia.org/wiki/Buffer_overflow
3. Stack based overflows: detect and exploit by Morton Christiansen , SANS institute 2007.
4. Buffer overflows for dummies, by Josef Neliben, SANS institute, 2002.
5. A Buffer overflow study- attacks and defenses, by Pierre-Alain FAYOLLE, Vincent GLAUME, ENSEIRB, Networks and Distributed Systems, 2002.
6. Format string attacks, by Tim Newsham, Guardent, Inc., September 2000.