Buffer Overflow Attack – Vulnerability in Stack

P.Vadivel Murugan
Research Scholar
Madurai Kamaraj University
Madurai-Tamil Nadu- India

Dr.K.Alagarsamy
Associate Professor
Madurai Kamaraj University
Madurai-Tamil Nadu-India

ABSTRACT
Most of the vulnerability based on buffer overflows aim at forcing the execution of malicious code, mainly in order to give a root shell to the user. The malicious instructions are stored in a buffer, which is overflowed to allow an unexpected use of the process, by changing various memory sections.

Buffer overflow attacks exploit a need of bounds checking on the size of input being stored in a buffer array. By writing the data into the memory assigned to array, the attacker can make arbitrary changes to program state stored an adjacent to the array.

A buffer overflow is an inconsistent, where a process attempts to store data beyond the boundaries of a fixed length buffer. So that the additional data overwrites next memory the techniques to exploit buffer overflow vulnerability vary per architecture, operating system and memory region locations. The overwritten data may include other buffers, variables and program flow data a technically inclined and malicious user may exploit stack-based buffer overflows to manipulate the program[9,10].

Keywords: Buffer overflow exploit, stack allocation, heap function, memory allocation

1. INTRODUCTION
In C and C++ and other programs have buffer overflow vulnerabilities, both because the C language lacks array boundary checking, and because the method of C programmers promote a performance oriented style that avoids error checking where possible. For example, many of the standard C library functions such as gets and strcpy do not do bounds checking by default.

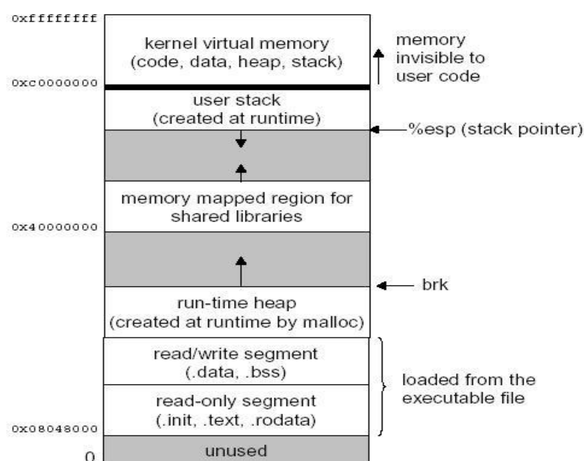

Figure 1: Stack collision Buffer Overflow Attack

The general form of buffer overflow exploitation is to attack buffers allocated on the stack. Stack collision attacks strive to achieve two mutually dependent goals, illustrated in Figure 1:

1.1 The Source of the Problem
The local variables are allocated on the stack, along with parameters and linkage information's. The accurate content and order of data on the stack depends on the operating system and processing unit architecture. When you use malloc, new, or the same functions to allocate a block of memory or instantiate an object, the memory is allocated on the heap.

Every time your program requests the input from a user, there is a potential for the user to enter inappropriate data. For example, they might enter the more data than you have reserved for in memory. If the user enters more data than will fit in the reserved memory space and you do not trim it, then that data will overwrite other data in memory. If the memory overwritten contained data vital to the operation of the program, this overflow will cause a bug that, being irregular, might be very hard to find. If the overwritten data includes the address of other code to be performed and the user has done this intentionally, the user can point to malicious code that your program will then executes.

In the case of data saved on the stack, such as a local variable, it is relatively simple for an attacker to overwrite the linkage information in order to execute malicious code. An attacker can also change local data and function parameters on the stack .The data on the heap changes in a no understandable way as a program runs; utilize a buffer overflow on the heap is more difficult. However, many exploits have involved heap overflows. Attacks on the heap might involve overwriting critical data, either to cause the program to crash, or to modify a value that can be exploited later (such as a program temporarily stores a user name and password on the heap and an attacker control to change them). In some cases, the heap contains pointers to executable code, so that by overwriting such a pointer an attacker can execute the malicious code. Although most programming languages check input against storage to prevent buffer overflows, C, and C++ do not. Because many programs link to C libraries, weakness in standard libraries can cause vulnerabilities even in programs written in "safe" languages. For this reason, even if you are confident that your code is free of buffer overflow problems, you should limit the exposure by running with least privileges possible.

2. DETECTING BUFFER OVERFLOWS
To test the buffer overflows, you should attempt to enter extra data than is asked for wherever your program accepts input. Also, if your program accepts data in a standard format, you should attempt to use malformed data. For example, if your program asks for a filename, you should attempt to enter a string longer than the maximum level filename. Or, if there is a data field that specifies the size of a block of data, attempt to

use a data block larger than the one you specify in the size field. If there are buffer overflows in your program, it will ultimately crash. (Regrettably, it might not crash until sometime later, when it tries to use the data that was overwritten.) .The crash log might provide some clues that the root of the crash was a buffer overflow attack.

```
Exception:  EXC_BAD_ACCESS (0x0001)
Codes:      KERN_INVALID_ADDRESS (0x0001) at 0x41414140

Thread 0 Crashed:

Thread 0 crashed with PPC Thread State 64:
  srr0: 0x0000000041414140 srr1: 0x000000004200f030                        vrsave: 0x0000000000000000
    cr: 0x48004242          xer: 0x0000000020000007   lr: 0x0000000041414141  ctr: 0x000000009077401c
    r0: 0x0000000041414141   r1: 0x00000000bfffe660   r2: 0x0000000000000000   r3: 0000000000000001
    r4: 0x0000000000000041   r5: 0x00000000bfffdd50   r6: 0x0000000000000052   r7: 0x00000000bfffe638
    r8: 0x0000000090774028   r9: 0x00000000bfffddd8  r10: 0x00000000bfffe380  r11: 0x0000000024004248
   r12: 0x000000009077401c  r13: 0x00000000a365c7c0  r14: 0x0000000000000100  r15: 0x0000000000000000
   r16: 0x00000000a364c75c  r17: 0x00000000a365c75c  r18: 0x00000000a365c75c  r19: 0x00000000a366c75c
   r20: 0x0000000000000000  r21: 0x0000000000000000  r22: 0x00000000a365c75c  r23: 0x000000000034f5b0
   r24: 0x00000000a3662aa4  r25: 0x000000000054c840  r26: 0x00000000a3662aa4  r27: 0x0000000000002f44
   r28: 0x000000000034c840  r29: 0x0000000041414141  r30: 0x0000000041414141  r31: 0x0000000041414141
```

Figure.2 Buffer overflow crash log

If there are many buffer overflows in your program, you should assume they are exploitable and fix them. It is much hard to prove that a buffer overflow is not exploitable than just to fix the bug.

3. CONCLUSIONS

Preventing buffer overflow exploits

Buffer overflow attack can be prevented. If the programmers were perfect in writing program coding, there would be no unchecked buffers, and consequently, no buffer overflow exploits. However, the programmers are not perfect, and unchecked buffers continue to abound. When unchecked buffers are found, vendors are often release patches that correct the problem. Unfortunately, keeping patches up to date on a large numbers of systems is difficult and many system administrators fail behind in patch deployments.

Calculating Buffer Sizes

You should always calculate the size of a buffer and then make sure you don't put excess data into the buffer than it can be hold. The reason you should not assume a static size for a

Don't use this style	Use this style instead

char buf[1024];
if (size <= 1023) {...}**or**
char buf[1024];...
if (size < 1024) {...}
	char buf[BUF_SIZE];
	if (size < BUF_SIZE) {...}
	or char buf[1024]; ...
	if (size < sizeof(buf)) { ... }

{char
file[MAX_PATH];...
addsfx(file); ...}
static *suffix = ".ext";
char *addsfx(char *buf)
{
return strcat(buf, suffix);
}
	{char
	file[MAX_PATH];...addsfx(file, sizeof(file));
	...}static *suffix = ".ext";char *addsfx(char *buf, uint size)
	{ return strlcat(buf, suffix, size);}

Table 1: C coding styles to use and avoid

buffer is because, even if you originally allocate a static size to the buffer, either you or someone else maintaining your programming code in the future might change the buffer size, but fail to change every case where the buffer is written to.

You should always use unspecified variables for calculating sizes of buffers and the data going into buffers. Because the negative numbers are stored as large positive numbers, if you use signed variables an attacker might able to alter in the size of the buffer or data by writing a large number of coding to your program.

4. REFERENCE

[1] Buffer Overflow Attacks on Linux Principles Analyzing and Protection Zhimin Gu Jiandong Yao Jun Qin Department of Computer Science, Beijing Institute of Technology (Beijing 100081)

[2] Computer emergency response team (cert). http://www.cert.org.The Meta sploit project. http://www. metasploit. com

[3] RamKumar ChincChani and Eric Van Den Berg. A fast staticanalysis approach to detect exploit code inside network flows.In RAID, 2005.

[4] C. Kruegel, E. KirDa, D. Mutz, W. Robertson, and G. Vigna.Polymorphic worm detection using structural information ofexecutables. In RAID, 2005.

[5] Michalis Polychranakis, Kostas G. Anagnostakis, andEvangelos P. Markatos. Network Level Polymorphic Shellcode Detection using Emulation. DIMVA , 2006.

[6] XinRan Wang, ChiChun Pan, Peng Liu, and Sencun Zhu. Sig free: A signature Free Buffer Overflow Attack Blocker. In 15 th Use nix Security Symposium, July 2006.

[7] Navjot Singh Libsafe: Protecting CriticalElement of Stacks White Paper December25, 1999Litchfield, D. (1999).

[8] Exploiting Windows NT for Buffer Overruns. Posted to Bugtraq mailing list in May1999. http://www.infowar.co.uk/mnemonix/ntbufferoverruns.htm.Mudge. (1995). How to write Buffer Overflows. http:// l0pht. com/ advisories/bufero.html.

[9] Smith, N.P. (1997). Stack smashing vulnerabilities in the UNIX operating system. Southern Connecticut State University. http:// destroy.net/ machines/ security/

[10] Summerfield, B. (1997) Re: Smashing the stack. From the Bugtraq mailing list. www. securityfocus. com / templates/ archive.pike 1997-01-21

[11] Spafford, E. H. (1988) The internet worm program: An analysis. ACM Computer Communication Review; 19(1), pp. 17-57. tp://www.cs.purdue.edu/homes/ spaf/techreps/823.ps.

[12] S. Alexander. Defeating compiler level buffer overflow protection. *The USENIX Magazine*, 30(3), June 2005.

[13] S. Nanda and T.C. Chiueh. Foreign code detection for Windows/X86 binaries. ECSL Technical report TR- 190, Computer Science Department,Stony Brook University, 2005.

[14] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflows vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[15] Z. Liang, R. Sekar, and D.DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self healing systems. In *USENIX Annual Technical Conference*, 2005.