IJournals: International Journal of Software & Hardware Research in Engineering
ISSN-2347-4890
Volume 3 Issue 4 April, 2015

Introduction to Buffer Overflow Attack

Author: Nilesh Kunhare[1]; Sanjay Kumar Tehariya[2]

Affiliation: Assistant Professor[1], MANIT Bhopal (CSE Department) [1]; Assistant Professor[2]; R.G.P.V. Bhopal (CSE Department) [2]

E-mail:nilesh954@gmail.com[1];sanjay_tehariya@rediffmail.com[2]

ABSTRACT

The paper focuses on what buffer overflow attack is, implementation of a buffer overflow attack, In order to get the remote root privileges, the effects of a buffer overflow attack, and ways to prevent buffer overflow attacks. Real world buffer overflow attack cases are presented.

Keywords: Vulnerabilities, buffers, stack, registers, Buffer Overflow Vulnerabilities and Attacks, current buffer over flow, Shell code, Buffer Overflow Issues, the Source of the Problem, prevention/detection of Buffer Overflow attacks

1. INTRODUCTION

A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows: "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." It uses input to a poorly implemented, but (in intention) completely harmless application, typically with root / administrator privileges. A buffer overflow attack is an exploit that takes advantage of a program that is waiting on a user's input. The input that is placed in the buffer is beyond the buffer's allocated size. This causes information to be overwritten, which can lead to a system crashing or an attacker implementing damaging code. Buffer overflows are one of the most common forms of software security vulnerability. The buffer overflow attack results from input that is longer than the implementer intended. To

understand its inner workings, we need to know a little bit about how computers use memory.

2. MEMORY ORGANIZATION

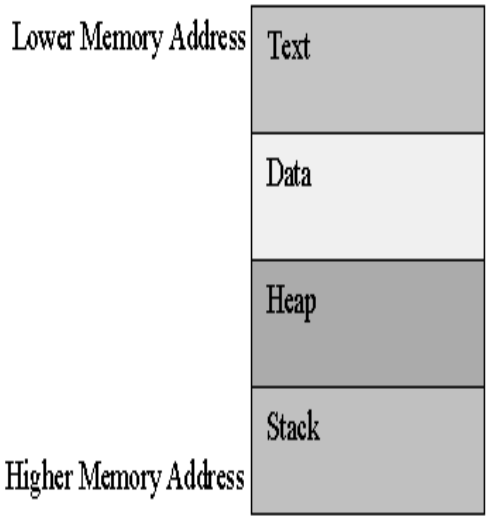


Fig 1: Process memory layout

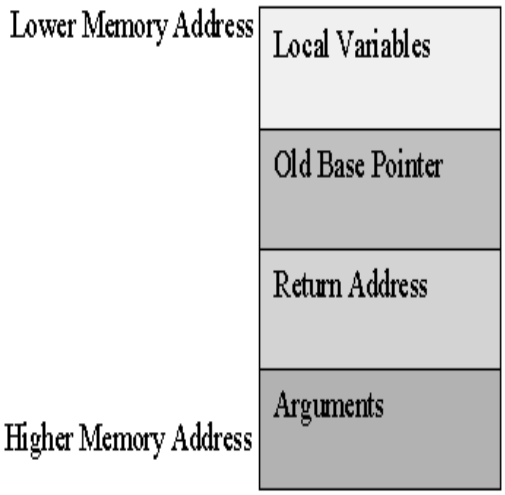


Fig 2: Stack layout

General idea of Buffer Overflow Attack

Target of Buffer Overflow Attacks Subvert the usual execution flow of a program by redirecting it to an Injected (malicious) code. The attack consists of:

1 Injecting new (malicious) code into some writable memory area,

2 Changing a code pointer (usually the return address) in such a way that it points to the injected malicious code.

The operating principle of a buffer overflow is closely related to the architecture of the processor on which the vulnerable application is executed. Data entered in an application are stored in random access memory in a region called a buffer. A correctly designed program should stipulate a maximum size for input data and make sure the input data do not exceed this value. The instructions and data of a running program are temporarily stored adjacently in memory in a region called a stack). The data located after the buffer contain a return address (called an instruction pointer) that lets the program continue its run-time. If the size of the data is greater than the size of the buffer, the return address is overwritten and the program will read an invalid memory address generating a segmentation fault in the application. A hacker with strong technical knowledge can make sure the overwritten memory address corresponds to an actual address, for example located in the buffer itself. As such, by writing instructions in the buffer (arbitrary code), it is easy for him to execute it. It is therefore possible to include instructions in the buffer that open a command interpreter (a shell) and make it possible for the hacker to take control of the system. This arbitrary code that makes it possible to execute the shell is called a shellcode. Buffer overflow is the root cause for most of the cyber-attacks like worms, zombies, botnets and server breaking.

Worms - Self-replicating malware computer program which uses a computer network to send copies of itself to other nodes and it may do so without any user intervention.

Zombies - a computer connected to the Internet that has been compromised by a computer virus or cracker and can be used to perform malicious tasks even in remote directions.

Botnets - a large number of compromised computers that are used to create and send viruses or flood a network with messages as a denial of service attack.

Return to libc attacks – a computer security attack usually starting with a buffer overflow in which the return address on the stack is replaced by the address of another instruction. This allows attackers to call malicious code without the need to inject malicious code into a program.

Code Injection Code can be injected by overflowing a local buffer allocated on the stack. The target of the injected code is usually to launch a shell to the adversary Therefore the injected code is often referred to as **shell code.**

Shell code A common element in all buffer overflow exploits is the *shell code*. Shell code is the attacker's code which is triggered by a exploiting a vulnerability. It is typically planted in an input buffer of a vulnerable program which is then tricked into running it. Shell code has to be compiled and assembled before it can be planted. Often, it is also necessary to character encode it. For example, when the target program expects character input from the user, the user would supply characters whose binary encodings (in ASCII) represent the shell code. This presents its own problems since the code must not use certain bytes. For example, end-of-file (^D) or newline characters which would terminate the input. The shell code usually contains instructions to launch a shell or a remote xterm. If the target program is a server daemon running as root then the shell will run as root too. Through a root shell the attacker has unrestricted access to the target machine. These days shell-code is much more than that and what it can do is only limited by a hacker's creativity. Because of this, some experts in the field have suggested the name shell-code is insufficient. As it is It is common practice to write shell-code in assembly and use an assembler such as NASM http://www.nasm.us/ which converts assembly instructions to OPCODEs and does some low level memory management, such as creation of stack frames (unless the user opts to do that themselves). This OPCODE is then modified to run as shell code. Note that the shell-code used in this paper was written by Steve Hanna.

Current generation Buffer Overflow Attack Format String Attacks

Format string vulnerabilities occur due to sloppy coding by software engineers. A variety of C language functions allow printing the characters to files, buffers, and the screen. These functions not only place values on the screen, but can format them as well. The following list contains common ANSI format functions:

Printf: print formatted output to the standard output stream.
Wprintf: wide character version of printf
F_printf: print formatted data to a stream usually a file).
Fwprintf: wide character version of fprintf
Sprintf: writes formatted data to a string.
Swprintf: wide character version of sprint.

Vprintf: writes formatted output using pointers to a list of arguments.
Vwprintf: wide character version of vprintf.
Vfprintf: writes formatted output to a stream using pointers to a list of arguments.
Vwfprintf: wide character version of Vfprintf.

3. History of Buffer overflow attack

In November 1988, the famous Internet Worm managed to bring down an estimated 10 percent of the Internet. One part of this worm exploited buffer overflow vulnerability in the TCP *finger* service that existed on some VAX computers running versions 4.2 or 4.3 of BSD UNIX. *Finger* is a simple program. It does nothing more than report information about users on a particular computer, like their name, office location, or phone number. The *finger* program in turn calls the C library function *gets ()*, a function used to handle string input data. While *gets()* had a buffer 512 bytes long, the Internet Worm passed a string 536 bytes long to the *finger* program that was in turn passed to *gets()*. This caused the buffer overflow that gave the Internet Worm access to the target computer. More recently, buffer overflow vulnerabilities have been responsible for many high-profile worms, including the Code Red worm of July 2001, the Slapper worm of September 2002, and the Slammer worm of January 2003. The Code Red worm infected over 359,000 computers in less than 14 hours and caused an estimated $2.6 billion in damage3 by exploiting buffer overflow vulnerability in Microsoft Internet Information Services (IIS). Information about this vulnerability was first released on June 12 2001. Figure 5 shows the history of some buffer overflow attacks.

3.1 A simple demonstration of a buffer over flow attack

```
void function( int a, int b , int c){
char buffer1[5];
char buffer2[10];
}
void main(){
function(1,2,3);
}
```

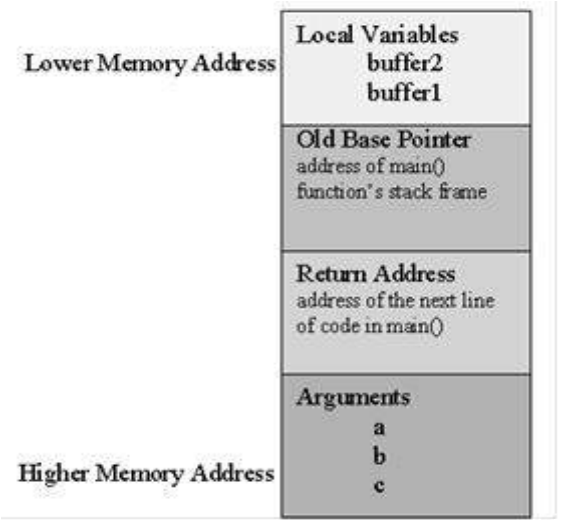


Fig 3. Memory Layout of function (a,b,c)

```
void function( int a, int b, int c) {
char buffer1[5];
char buffer2[10];
int *ret;
ret = buffer1 + 12;
(*ret) += 8;
}
void main() {
 int x;
x = 0;
function(1,2,3);
x = 1;
printf( "%d\n",x);
}
```

This function jumps over the x=1 assignment directly to the printf()
and prints the value as 0. The offsets (12, 8 used above) are
machine-dependent.

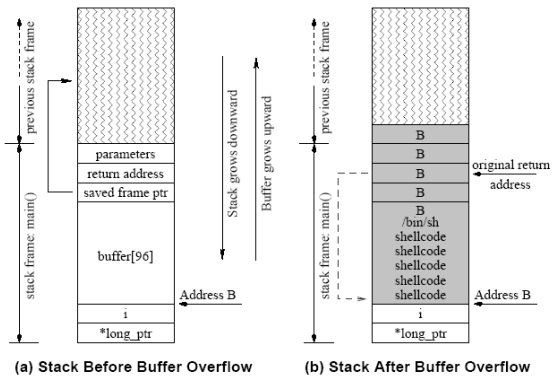
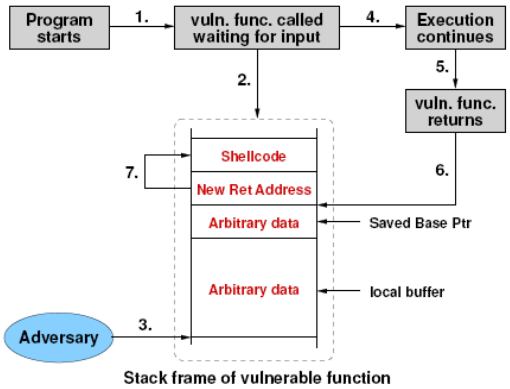


Fig 4. Execution of attack in memory



Stack frame of vulnerable function

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).

Fig 5: A brief history of some buffer overflow attacks

4. Prevention Techniques

4.1 Main problem: strcpy(), strcat(), sprintf() have no range checking. "Safe" versions strncpy(), strncat() are misleading – strncpy() may leave buffer unterminated. – strncpy(), strncat() encourage off by 1 bugs.

4.2 Defenses

• Type safe languages (Java, ML). Legacy code?

• Mark stack as non-execute. Random stack location.

• Static source code analysis. • Run time checking: StackGuard, Libsafe, SafeC, (Purify).

• Black box testing (e.g. eEye Retina, ISIC).

Use of Stack guard:

DETECTING RETURN ADDRESS
CHANGE: CANARY. Place a Canary word before the return address. When the function returns, it first checks to see that the "**CANARY WORD**" is intact before jumping to the address
pointed to by the return address.

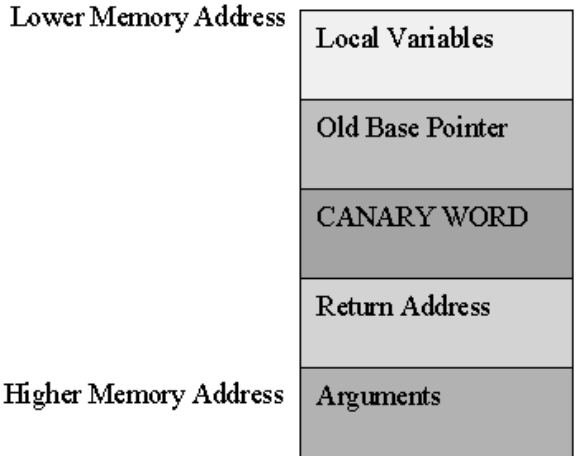


Fig 6. Use of STACKGUARD

5. REFERENCES

[1] C. Cowan et al, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, in Proceedings of the 7th USENIX Security Symposium, pp. 63-78, San Antonio, TX, January, 1998.

[2] Buffer Over Flow Attacks, Ataques por desbordamiento de búfer Angriffe durch Pufferüberlauf (buffer overflow) Attaques par débordement de tampon (buffer overflow) Attacchi buffer overflow Ataques por profusão de tampão (buffer overflow)
[3] C. Cowan et al, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, in Proceedings of the 7th USENIX Security Symposium, pp. 63-78, San Antonio, TX, January, 1998.
[4] E.Malekian, "Penetration in network and the methods of confrontation", Nas publishment,2002.
[5] Crosscheck Networks,"Vulnerability Assessment of SAP Web Services",website: http://www.crosschecknet.com.
[6] Andreas Wiegenstein, Frederik Weidemann, Dr. Markus Schumacher, Sebastian Schinzel," Web Application Vulnerability Scanners – a Benchmark", Version 1.0 - 2006-10-04.
[7] OWASP, the Open Web Application Security Project," The Ten Most Critical Web Application Security Vulnerability",January,27th,2004,website:http://www.owasp.org.
[8] Kenneth F.Belva,CISSP Franklin Technologies United,INC, "Case Studies in Discovering Previously Unknown Web Application Vulnerabilities", Website: http://www.ftusecurity.com.
[9] CENZIC, Securing Enterprise Application, "Continuous Testing of Production Web Applications", website: http://www.cenzic.com/.
[10] "Secure Programming with Static Analysis (Addison-Wesley Software Security Series)", Brian Chess, Jacob West.
[11] Michael Howard, David LeBlanc, and John Viega, 19 Deadly Sins of Software Security "Programming Flaws and How to Fix Them",
Osborne McGraw-Hill, 2005.
[12] Jeffrey Rubin, "Review: Web Vulnerability Scanners", Secure Enterprise Magazine, Sept. 2006.
[13] Mark Curphey, Rudolph Araujo, "Web Application Security Assessment Tools", IEEE Security & Privacy (Vol. 4, No.4), July / August 2006.
[14] H.Q.Nguyen, B.Johnson, M.Hackett, "Testing Applications on the Web, Wiley Publishing", 2003.
[15] Arian Evans, "Software Security Quality: Testing Taxonomy andTesting Tool Classification", 2nd Annual OWASP Conference, Washington DC, Oct. 2005.
[16] http://www.cert.org/advisories/CA-2001-19.html
[17] http://www.cert.org/advisories/CA-2003- 4.html
[18]http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html
[19]DilDog, *The Tao of Windows Buffer Overflows*, http://www.newhackcity.net/win_buff_overflow/
[20] Crispin Cowan, et al., StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overfl ow Attacks.
[21]http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/
[22] Aleph One, Smashing the Stack for Fun and Profit. Originally published in Phrack 49-14.1996
[23]http://www.newscientist.com/news/news.jsp?id=ns99994696
[24] J. Wilander, M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, in Proceedings of the 10th Network and Distributed System Security Symposium, pages 149--162, San Diego, CA, Feb 2003
[25] Libsafe: Protecting Critical Elements of Stacks Timothy K. Tsai, Navjot Singh
http://citeseer.nj.nec.com/baratloo99libsafe.html
[26]. Transparent Run-Time Defense Against Stack Smashing Attacks Arash Baratloo, Navjot Singh, Timothy Tsai Proceedings of the USENIX Annual Technical Conference
http://citeseer.nj.nec.com/baratloo00transparent.html
[27] Architecture Support for Defending Against Buffer Overflow Attacks Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel. Ravishankar K. Iyer