A Checklist Based Approach for the Mitigation of Buffer Overflow Attacks

S. K. Pandey, K. Mustafa, S. I. Ahson
Department of Computer Science
Jamia Millia Islamia (Central University), New Delhi-110025, INDIA
santo.panday@yahoo.co.in, kmfarooki@yahoo.com,drsiahson@yahoo.com

***Abstract*- Buffer overflows has appear to be one of the most common problems in the area of software security. Many of the buffer overflow problems are probably the result of careless programming, which might have been found and corrected by the developers, before releasing the software. The work presented in this paper is intended to detect and prevent such buffer overflow vulnerabilities. In this paper, two separate checklists are proposed to both programmers as well as test engineers for verifying the software during coding and testing phases respectively for building secure software.**

Keywords: Software Security, Buffer Overflow, Attacks, Checklist

I. INTRODUCTION

Buffer overflow attacks aim to alter the execution of a vulnerable program by copying data to a variable in such a way that the original storage capacity is exceeded [1]. This may cause excess data to spill over the unallocated address space and overwrite the pointer to the next instruction after a function call. However, in order to deploy the attack successfully, execution must be accurately diverted to attacker's arbitrary code. To do so, the attacker might develop a program, which can assemble the different components of the malicious buffer. Moreover, because the location of the vulnerable program in address space is determined at runtime, certain characteristics of the malicious buffer should be approximated in the code.

The principal objective of evolving overflow attacks is to access critical applications against buffer overflow vulnerabilities. However, recent work on vulnerability testing indicates that intrusion detection systems can detect a particular instance of an attack, but are generally unable to 'generalize' to the class of overflow attacks [2, 3, 4, 5, 6, 7, 8, 9].

The main contribution of this work is to prepare checklists for programmers as well as test engineers, to take care of these attacks at the time of coding. And if any thing is missed during coding phase by mistake then test engineer must catch it during the testing phase by using another checklist. Such practices may facilitate the mitigation of the attacks, exploiting the inherent buffer overflow vulnerabilities.

The remainder of this paper is organized as follows. Section II describes the buffer overflow concepts. Checklists methodology is discussed in Section III. Results after using the checklists are discussed in Section IV and Conclusions are drawn in Section V.

II. BUFFER OVERFLOW ATTACKS

Buffer overflow attacks exploit a lack of bounds checking on the size of input being stored in a buffer array. By writing data *past* the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. By far, the most common data structure to corrupt in this fashion is the stack, called a "stack smashing attack," which we briefly describe here, and is described at length elsewhere [10, 11, 12].

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible [13, 14, 16]. For instance, many of the standard C library functions such as gets and strcpy do not do bounds checking by default [10].

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

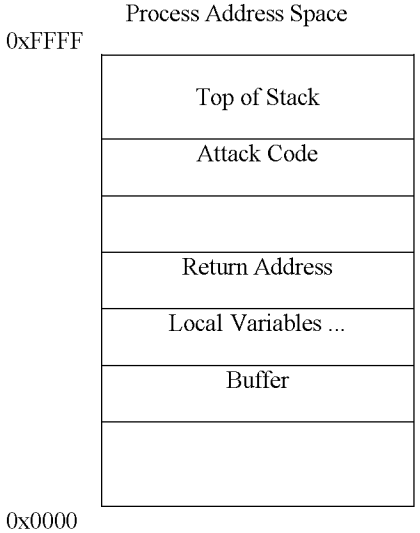

Figure 1: Buffer Overflow Scenario

Inject Attack Code: The attacker provides an input string that is actually executable, binary code native to the machine being attacked. Typically this code is simple, and

does something similar to exec ("sh") to produce a root shell.

Change the Return Address: There is a stack frame for a currently active function above the buffer being attacked on the stack. The buffer overflow changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps to the attack code.

The programs that are attacked using this technique are usually privileged daemons; the programs that run under the user-ID of root to perform some service. The injected attack code is usually a short sequence of instructions that spawns a shell, also under the user-ID of root. The effect is to give the attacker a shell with root's privileges.

If the input to the program is provided from a locally running process, then this class of vulnerability may allow any user with a local account to become root. More distressing, if the program input comes from a network connection, this class of vulnerability may allow any user anywhere on the network the ability to become root on the local host. Thus while new instances of this class of attack are not intellectually interesting, they are none the less critical to practical system security.

Engineering such an attack from scratch is surely non-trivial. Often, the attacks are based on reverse-engineering the attacked program, so as to determine the exact offset from the buffer to the return address in the stack frame, and the offset from the return address to the injected attack code. However, it is possible to soften these exacting requirements [11]:

1. The location of the return address can be approximated by simply repeating the desired return address several times in the approximate region of the return address.

2. The offset to the attack code can be approximated by prepending the attack code with an arbitrary number of NOP instructions. The overwritten return address need only jump into the middle of the field of NOPs to hit the target.

The cook-book descriptions of stack smashing attacks [10, 11, 12] have made construction of buffer overflow exploits quite easy. The only remaining work for a would-be attacker to do is to find a poorly protected buffer in a privileged program, and construct an exploit. Hundreds of such exploits have been reported in recent years [4].

III. CHECKLIST METHODOLOGY

Some general guidelines of programming practices are proposed by experts. Some of them are given as follows [1]:

- Programmers should avoid gets(). They should use fgets() with stdin as the file instead. fgets() will ask for specifying a string size, but it should be in mind that the size of the string should not be greater than the size of the buffer otherwise an off-by-one error may occur.
- Programmers should avoid strcpy() and strcat().They should use strncpy() and strncat() instead. strcpy() and strcat() are with constant strings for the source, but it's probably not safe to assume that a string will always be constant.
- Programmers should use precision specifiers with the scanf() family of functions (scanf(), fscanf(), sscanf(), etc.). Otherwise they will not do any bounds checking for the software.
- Programmers should never use variable format strings with the printf() family of functions.
- Programmers should watch for off-by-one errors with otherwise safe functions such as memcpy().
- When using streadd() or strecpy(), it should make sure that the destination buffer is four times the size of the source buffer.
- There should be proper care of every library function that takes a pointer (especially string pointers) as input for a place to store its output. If there is not another parameter for the buffer size or string size then that function does not do bounds checking.
- Programmers should also be careful and do bounds checking when using functions like getc() and read() in a loop.
- Functions like syslog() and getopt() take strings as input but, depending on the implementation, might not check the size of the string before using it. Programmers should truncate strings to a reasonable length before passing pointers to them as input to any library function.
- When using library functions that take buffer size as a parameter, programmers should make sure that buffer is as big as they say it is.

Here two checklists are proposed, one for Programmers, and other for Test Engineers:

A. Checklist for Programmers

1. Is there bounds checking on arrays?
2. Is there bounds checking on pointer arithmetic?
3. Is there any provision for checking before copy to, format, or send input to a buffer, for making sure that it is big enough to hold whatever might be thrown at it?
4. Is there any provision for checking the input because input itself might cause a buffer overflow and not the size of the input?
5. Is there correct use of unsafe library functions?
6. Is there any provision for checking of off-by-one errors?
7. Is this clear that an array declared in C as int A[100] can only be accessed with indices A[0] through A[99]?
8. Is there any provision for checking old code?
9. Are all the assumptions taken with attacker's point of

view?
10. Is it considered that all buffer overflows are a security risk?

B. Checklist for Test Engineers

1. Is there a checking of all string and buffer inputs by entering a very long string or too much data?
2. Is there any checking for one number in the partition?
3. Is there any checking for each partition of value or size on any input?
4. Is there any provision for testing of old code when programmers are using it for new things, even if it is tested before?
5. Is the code tested on all platforms it was meant to run on?
6. Are all the assumptions taken with attacker's point of view?
7. Is it considered that all buffer overflows are a security risk?

IV. RESULTS AFTER USING THE CHECKLISTS

These checklists were tested in a development project of an organization and results given by them show that risk of such attacks is reduced up to 80%. On the request of the organization, we are not disclosing the name and details of the project.

V. CONCLUSION & FUTURE WORK

A checklist based solution is proposed for prevention and detection of buffer overflow attacks. If programmers and test engineers keep these points in mind, then the prevention may be done at the time of coding and testing phase respectively. In future, we are planning to make a security assessment framework in which we will use these checklist based mitigation mechanisms of defects. This work will help to software programmers and security experts for building secure software which is the main objective of this project.

ACKNOWLEDGEMENT

The research work is funded by DIT, Ministry of Communications and Information Technology, Govt. of India, under grant no. 12(51)/05-ESD.

REFERENCES

[1] Foster J.C., Osipov V., Bhalla N., Heinen N., "Buffer Overflow Attacks: Detect, Exploit, Prevent", Syngress Publishing, ISBN 1-932266-67-4, Ch.5, 2005.

[2] Christodorescu M., Jha S., "Static analysis of executables to detect malicious patterns", Proceedings of the USENIX Security Symposium, 2003.

[3] Detristan T., Ulenspiegel T., Malcom Y., Underduk M. S., "Polymorphic shellcode engine using spectrum analysis", Phrack Online Magazine, 61, 2003.

[4] Dozier, G., Brown, D., Cain, K., Hurley, J., "Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers," Proceedings of the Genetic and Evolutionary Computation Conference, Lecture Notes in Computer Science, LNCS 3102, pp 263-274, 2004.

[5] Marti R., "THOR: A tool to test intrusion detection systems by variations of attacks", Master's Thesis, Swiss Federal Institute of Technology, March 2002.

[6] Rubin S., Jha S., Miller B.P., "Automatic Generation and Analysis of NIDS Attacks", 20th Annual Computer Security Applications Conference - ACSAC, pp 28-38, 2004.

[7] Tan, K.M.C., Killourhy, K.S., Maxion, R.A., "Undermining an Anomaly-based Intrusion Detection System using Common Exploits", 5th International Symposium on Recent Advances in Intrusion Detection - RAID, Lecture Notes in Computer Science, LNCS 2516, pp 54-73, 2002.

[8] Vigna, G., Robertson, W., Balzarotti D., "Testing Network Based Intrusion Detection Signatures Using Mutant Exploits", ACM Conference on Computer Security, 2004.

[9] Wagner, D., Soto, P., "Mimicry Attacks on Host-based Intrusion Detection Systems", ACM Conference on Computer Security, pp 255-264, 2002.

[10] "Mudge". How to Write Buffer Overflows. . http://l0pht.com/advisories/bufero.html, 1997

[11] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[12] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. http://millcomm.com/~nate/ machines/security/stack-smashing/ nate-buffer.ps, 1997.

[13] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A reexamination of the Reliabilityof UNIX Utilities and Services. Report, University of Wisconsin, 1995.

[14] B.P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12):33–44, December 1990.

[15] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, *The SANS Network Security Digest*. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard.

[16] S. K. Pandey, S. I. Ahson: "A Taxonomy of Software Security Vulnerabilities", Proceedings of the National Conference on Security Issues in e-commerce, Aligarh, India, Mar 10, 2007.