Buffer Overrun: Techniques of Attack and Its Prevention

Mahtab Alam[1], Prashant Johri[2], Ritesh Rastogi[3]
1: Asst. Prof. & Head of Dept. Computer Science, Aryabhatt College of Engineering and Technology, Baghpat
Email: alam_mahtab@rediffmail.com
2: Asst. Prof, Dept. of Computer Sc., NIET, Gr. Noida, Email: johri.prashant@gmail.com
3: Asst. Prof, Dept. of Computer Sc., NIET, Gr. Noida, Email: rit_raj@hotmail.com

Abstract: Buffer Overflow attack has been considered as one of the important security breaches in modern software systems that has proven difficult to mitigate. This attack allows the attacker to get the administrative control of the root-privilege by using the buffer overflow techniques by overwriting on the address of a returned function, function pointer stored on the memory and overflow a buffer on the heap. In this paper, we present the different buffer overflow techniques used by the exploiters and the methodologies applied to mitigate the buffer overflow.

Keywords**:** Buffer Overrun, Heap Smashing, Pointer Subterfuge, Arc Injection

1. Introduction

The complexity and opportunity of software systems vulnerabilities are regularly growing with the use of computer system. Almost every software system is insecure because of the high growth rate of expertise of the malicious users. Software system is considered insecure because of its existing security holes. Buffer Overflow attacks are the most common security intrusion attack [3,5] Software security holes related to buffer overflow accounts the largest share of CERT advisories. David Wagner from University of California at Berkeley shows that buffer overflows stand for about 50% of the vulnerabilities reported by CERT [3]. In the memory allocation table, variables with similar properties are assigned into the same buffer area, and their locations are adjacent to each other. A buffer overflow condition occurs when a program attempts to read or write outside the bounds of a block of allocated memory or when a program attempts to put data in a memory area past a buffer [1]. A buffer overflow may happen accidentally during the execution of a program [2]. Buffer overflow is best known for software security vulnerability, as buffer overflow attack can be performed in legacy as well as newly developed application. Buffer overflows are applicable to most operating systems [2]. In particular the attacks are quite successful in Windows NT and Windows 2000 system [4,6,7,8,9,10]

A buffer is a sequential section of computer memory that holds more than one instances of the same data type. It is allocated to contain anything from a character string to an array of integers. An extremely common kind of buffer is simply an array of character type. Overflow occurs when data is added to the buffer outside the block of memory allocated to the buffer. Buffer overflow can be conducted either by locally or remotely. In a local attack the attacker already has access to the machine and acquires the access privileges. On the other hand in remote attack the attacker deliver commands through network port, and simultaneously gains the unauthorized access privilege.

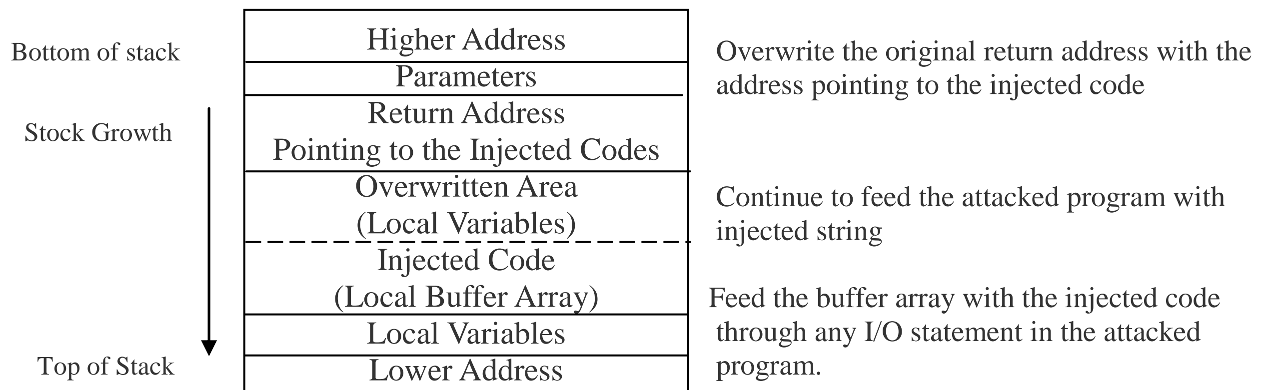

Figure -1 Fragment of a Stack

Buffer overrun is characterized as a stack overrun or heap overrun depending on what memory gets overrun. Stack memory is used in C and C++ compilers when local variables as well as parameters have been used. Heap memory in this context refers to the dynamically allocated memory uses new/delete or *alloc*(), *malloc*().

Buffer overrun mainly consist the following three steps [2]: Planting the attack code into the program, copying into the buffer which overflows it and corrupts adjacent data structures, and hijacking the program to execute code.

Commonly buffer overflow can be executed by using the stack smashing: modifying the return address saved on the stack to point to code the attacker supplies that resides in stack buffer overrun exploitation as shown in the Figure-1 [11].

2. Buffer Overrun Methods

In recent years, hackers have developed some other approaches of buffer overrun to exploit software such as Arc Injection, Pointer Subterfuge and Heap Smashing.

2.1 Arc Injection

Arc Injection sometimes also called as return–into-libc transfer control of the code that already exists in the memory space. These types of injection insert a new arc using the installation of an existing functions such *system*(), *execl*() or *printf()* as into the program's control flow graph and create a shell on the compromised machine with the permission of the compromised program. An exploiter uses the arc injection to invoke a number of functions in a small program that includes chained functions in sequence with arguments that are supplied by them.

Example: Following are the main functions used in arc injection buffer overrun vulnerability [23].

2.1.1 system():

system takes a single argument and executes that argument with /bin/sh.

2.1.2 execl():

execl() requires an argument list that is null terminated. This will end our string early, so there is a need to chain multiple calls to libc.

2.1.3 printf():

printf is very popular output function used in C language, but it can be used for exploitation of a program using following techniques:

- The %n parameter prints how many characters have been written so far to a location specified by the argument.
- By using n$ inside a parameter, one can read the value of the nth argument.
- Combining these, %3$n will write the number of characters printed so far to the address specified in the 3rd argument.

2.2 Pointer Subterfuge

Pointer Subterfuge is a general expression for exploitation by using modification of pointer address. This approach used by an attacker to divert the control flow of a program by using function pointers (a variable whose value is used as an address) as an alternative to the saved return address, or modify the program flow by subverting data pointers [1]. A pointer subterfuge software exploitation is illustrated as below [24]:

```
void SomeFunc() {
	// do something
}
typedef void (*FUNC_PTR )(void);

int MalFunc(char *ptString) {
	char buf[32];
	strcpy(buf,szString);
	FUNC_PTR fp = (FUNC_PTR)(&SomeFunc);
	// Other code
	(*fp)();
	return 0;
}
```

If the malicious user uses the ptString argument in function MalFunc, then the buffer in the stack buf is ready to overpower. If the attacker overwrites the function pointer fp, then this pointer points to another address and exploits code and the function (*fp)() is invoked. To overcome the problem caused by pointer subterfuge we have to protect the function pointer.

2.3 Heap Smashing

Heap Smashing attack overruns a heap buffer to change the control flow of a program. Such overflow could overwrite

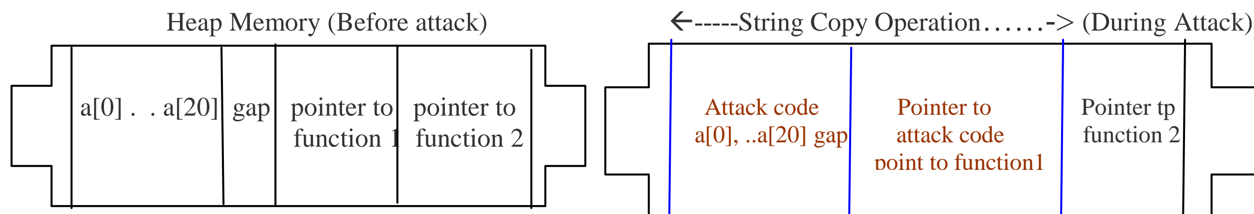

Figure-2 Heap smashing attack

function pointers stored on the heap to redirect the control flow. Heap Smashing allows an attacker to exploit the software by implementing some assumed variants in dynamically allocated memory. Although this type of attack is less common in practice but can be dangerous. Attacker typically is not aware the heap block's location ahead of time and the standard

trampoline approaches are not effective. A typical example of heap smashing is shown in the Figure-2 [25]

3. Mitigation Techniques

A user may overwrite the input buffer by providing more data for storage within the buffer than the programmer has expected. Errors in string manipulation have long been recognized as a leading source of buffer overflows in C and C++. A number of mitigation strategies have been devised. These include mitigation strategies at requirement and design levels to prevent buffer overflows from occurring and strategies that are Static analysis techniques should be employing to find the common coding problems that could expose buffer overrun. A through interface testing will further reduce the risk by providing the existence of buffer overruns and allowing the development team to fix them as they are found.

By using all the above mentioned techniques in a layered approach at secure software requirement analysis phase, it may be possible to reduce the risk of buffer overruns at some extent

3.1 Layered Approach

Buffer overruns are generally caused by introducing bugs during application implementation. These bugs can be mitigated by using following three techniques [12]:

3.1.1 Using Interpreted language:

Developed an application using a interpreted language that reduces the potential for buffer overruns, such as C# or Java. The interpreted code eventually calls into support code that is written into a compiled language such as C/C++ that could contain buffer overrun.

3.1.2 High Quality Code:

Buffer overrun to some extent can be mitigated by ensuring development in an environment that encourages a high-quality code that requires developers to participate in code review, running unit test, and educating them about buffer overruns. Buffer overrun sneak into the code either through inexperience or

designed to detect buffer overflows and securely recover without allowing the failure to be exploited. Rather than completely relying on a given mitigation strategy, it is often advantageous to follow a defense-in-depth strategy of combining multiple strategies. Some approaches to prevent the buffer overrun in a program are described in this sequence.

3.1.3 Testing Public Interfaces:

Static analysis techniques should be employing to find the common coding problems that could expose buffer overrun. A through interface testing will further reduce the risk by providing the existence of buffer overruns and allowing the development team to fix them as they are found.

By using all the above mentioned techniques in a layered approach at secure software requirement analysis phase, it may be possible to reduce the risk of buffer overruns at some extent

misunderstanding on the part of the developer regarding how the code works within the large application. Unfortunately, there are a large number of dangerous functions that come with C and C++. Any place a program uses them is a warning signal, because unless they are used carefully, they become vulnerable [26].

3.2 Traditional Approaches

Traditionally, buffer overruns caused by unsafe functions in the C library, like *strcpy*() have been identified and replace them with safe function like *strncpy*(). In this approach the static intrusion prevention method in which the software bugs can be eliminated by examining the large number of program carefully is applied. Removing all security bugs from a program is considered infeasible [17] which makes the static prevention incomplete. There are some tools available that one can use to automate the search for the vulnerability [13, 14, 15], but still manual auditing of the code which makes this massive and very expensive approach [2]. While the value of this systematically auditing code has been successfully executed, the approach is not guaranteed to produce buffer-overrun-free code [16].

3.3 Compiler Approaches

Almost all the buffer overruns problem take place in the compiler-based programming languages. Range checking indices are very effective against the buffer overrun attacks. Buffer overrun attack is not possible in Java programming language because Java automatically checks that an array index within the proper bounds. In C language it is not possible because of the dichotomy

between arrays and pointers [2]. When a compiler compiles a function *strcpy*(char* a, char* b) the two arguments are pointers and it is impossible for a complier to know the length of the corresponding array, and complier cannot generate code for range checking inside the function. Compiler Approach is a kind of dynamic intrusion prevention techniques which allow changing the run-time environment or system functionality making program at some extent less vulnerable.
C compiler allocates memory space for a local variable and a function return address in the same stack frame and adjacent to each other as shown in Figure-1. To mitigate the possibility of this type of problem some types of safe compilers are invented and implemented which are as follows [18}:

3.3.1 StackGuard:

The StackGuard complier was invented and implemented by Crispin Cowan [18]. The main objective of the StackGuard is to prevent the dynamic intrusion prevention by detecting and stopping stack based buffer overflow and return address. The overhead for StackGuard can reportedly be as high as 40% [19, 20].

In buffer overrun arrack, the stack is target to fill the higher address area and then overwrite the other local variable below the area specified for local variables The key ideas to mitigate this technique is to place a dummy value known as canary, in between the return address and local variables as shown in Figure -3. If the attacker try to overrun the buffer area, the canary intact the changing the return address, either by overwriting the canary with its correct value and thus not changing the actual one, or by overwriting the return address through a pointer.

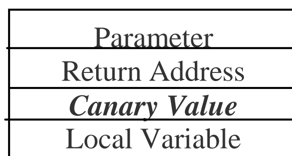

Figure-3 StackGuard Frame

Although these techniques only stop the buffer overflow attack that generally attack against the return address, but attacker still have potential to abuse the pointers variables, making it point at the return address and writing a new address to the memory position.

3.3.2 Stack Shield

Stack Shield is a tool for adding protection to programs from this kind of attacks at compile time without changing a line of code [21]. Stack Shield is also a compiler extension mechanism that protects the return address. Stack shield is more secure protection system than tool like Stack Guard. In the latest version 0.7 of stack shield there are two techniques which protect against writing of the return address and one against overwriting of function pointers.

(a) Global Ret Stack

In this mechanism the return address upon calling a function has been copied to Global Ret Stack array of 32-bit entries. Whenever a malicious user alters the address of the function, it has no effect since the original return address is remembered. In this method only prevention not attack detection is possible in this technique.

(b) Ret Range Check:

In this mechanism the value of the return address of a current function is store in the global variable. While calling the function the return address on the stack is compared with the value copied in the global variable. If there is any difference the program execution is halted. It can detect the attack too.

(c) Protection of Function Pointer:

Function pointer normally points to the text segment of the process memory. If the process ensure that no pointer is allowed to point the other parts of the memory except text segment, it is impossible for an attacker to inject a payload (Combination of data and code) into the process. Protection of function pointer can be possible by declaring a global variable in the data segment and its address is used as a boundary value. If the function points above or below the boundary the process is terminated.

4. Future Work

The results obtained from our work shows that buffer overrun can be easily conduct by smashing heap and stack memory or by overrunning the bytes available in the memory. Our motive of future work is to reduce these types of problem by using pointer encryption and resolve the techniques to mitigate the very small size even 'one- byte' of buffer overrun.

5. Conclusion

Buffer overrun is the most important software security breach. There are several techniques for stopping the common security buffer overrun. But we have presented some mitigation techniques related to requirement and design level of software development life cycle. Applying the above mentioned approaches one can mitigate the buffer overrun problem at some extent.

Reference

[1]. Jonathan Pincus, Brandon Baker, Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", IEEE Coputer Society, 2004, pp.20-27.

[2]. Istvan Simon, "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks", January 31, 2001.

[3]. D. wagner, J. S. Foster, E.A. Brewer, and A Aiken, "A First Step towards automated detection of Buffer Overrun Vulnerabilities. In Proceedings of the IEEE Symposium on Security and Privacy, pp.-3-17, February 2000.

[4]. DilDog, The TAO of Windows Buffer Overflow, http://www.cultdeadcow.com/cDc_files/cDc-351/, 1998

[5]. D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis", IEEE Software, 19(1), pp.-42-51, February 2002

[6]. Nishad Herath, (Joey_) Advanced Windows NT Security, in Black Hat Asia Conference, 2000. URL: http://www.blackhat.com/html/bh-asia-00/bh-europe-00-speakers.html#Joey

[7]. Barnaby Jack, (dark spyrit), Win32 Buffer Overflows (Location, Exploitation, and Prevention), Phrack Magazine 55,(15), URL: http://phrack.infonexus.com/search.phtml?view&article=p55- 15, 1999.

[8]. Mudge, How to Write Buffer Overflows, http://l0pht.com/advisories/buffero.html, 1997

[9]. Ryan Russel, Rain Forest Puppy, Elias Levy, Blue Boar, Dan Kaminsky, Oliver Friedrichs, Riley Eller, Greg Hoglund, Jeremy Rauch, and Georgi Guninski, "Hack Proffing Your Network Internet Tradecraft", Syngress, 2000.

[10]. Joel Scambray, Stuart McClure, George Kurtz, Hacking Exposed, Network Security Secrets & Solutions , Second Edition, Osborne/McGrawHill , 2001

[11]. Fu-Hau-Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks", Department of Computer Science, State University of New York at Stony Brook

[12]. Jason Taylor, "Webservices Risk Assessment and Recommendation",Security Innovation Commercial Tools Division 1318 S, Babcock Street, Melbourne, 2003/2004

[13]. HalVar Flake, "Auditing Binaries for Security Vulnerabilities", in Black Hat Europe Conference, 2000, URL: http://www.blackhat.com/html/bh-europe-00/bh-europe-00-speakers.html

[14]. Nishad Herath (Joey_) "Advanced Windows NT Security", In Black Hat Asia Conference, 2000

[15]. David Wagner, Jeffrey S Foster, Eric A. Brewer, and Alexander Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", in Proceedings 7th Network and Distributed System Security Symposium 2000

[16]. Chris Evans, Nasty Security Hole in lprm, posted in BugTraq, April 18, 1998

[17]. D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities", in Proceedings of the 2001 USENIX Security Symposium, Washington DC, USA, August 2001.

[18]. C. Cowan, C. Pu, D. Maier, J. Walpote, P. Bokke, S. Beatie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "STachGuard: Automatic adaptive detection and prevention of buffer-overrun attacks", in Proceedings of the 7th USENIX Security Conference, pp-63-78, San Antonio, Texas, Jannuary 1998.

[19]. David Llewellyn-Jones, Madjid Merabti, Qi Shi, Bob Askwith, "Buffer Overrun Prevention Through Component Composition Analysis", Proceedings of the 29th Annual International Computer Software and Application Conference (COMPSAC'05), IEEE Transaction 2005.

[20]. C. Cowan, S. Beattie, R. Finnin Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard", In Linux Expo., May 1999.

[21]. Stack Shield A "stack smashing" technique protection tool for Linux: URL: http://www.angelfire.com/sk/stackshield/

[22].http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/arc_injection/arc_injection.html

[23]. John Wilander, Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", Published at 10th Network and Distributed System Security Symposium (NDSS), 2003.

[24].http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx

[25]. Christof Petzer, Zhen Xiao, "Detecting Heap Smashing Attacks Through Fault Containment Wrappers.

[26]. David Wheeler, "Secure Programmer: Countering Buffer Overflows", 27 Jan 2004

Author Biographies

Mahtab Alam: Mr. Mahtab Alam is working as an Assistant Professor and Head of Dept. of Computer Science in Aryabhatt College of Engineering and Technology, Baghpat. He has having 12 years experience in the field of teaching and6 years in software industry. He has published Books in Cryptography and Information Security and currently pursuing Ph. D. in the field of Information Security. His areas of interest are Software Engineering, Information Security, Database Management System and computer graphics.

Prashant Johri : Dr. Prashant Johri is working as as Assistant Professor, Dept. of Computer Sc. In Noida Institute of Engineering and Technology, greater Noida, He has more than 10 years experience in teaching. He has completed his research work in software reliability and his areas of interest are Software Engineering, Design and Analysis of Algorithm, Simulation and Modeling.

Ritesh Rastogi: Mr. RItesh Rastogi is working as a Assistant Professor and Head of Department (MCA) at NIET Gr.Noida for last Eight years. He is having an experience of around ten years in the field of teaching. He has published number of Books and is continuously involved in the field of publication. His main areas of concern are Software Engineering, Information Security, Database and Information System.