Buffer Overflow : Anomaly in Application Security

Anwar Ahamed Shaikh,Anurag Srivastava,Ariq Ahmad,Ashutosh Singh and Adnan Abdul Rashid
Department of Computer Science and Engineering,
Integral University,
Lucknow, India.

***Abstract*— One of the most common attacks of applications include Buffer Overflow. A buffer overflow, or buffer overrun, is an anomaly where a program, in which data is being written to buffer, goes over the buffer's boundary and overwrites side memory locations. This Violation is perhaps a special case of memory safety. Software security vulnerability's best known form is buffer overflow. Even though software developers are aware of what a buffer overflow vulnerability is, the buffer overflow attacks for both newly-developed and legacy applications are very common. Part of this problem lies in the prevention techniques which are highly error-prone ones, and part of the problem is because of the nature of buffer overflows as it can occur in numerous ways. It is the nature of Buffer Overflows that makes it difficult to be discovered and even if it is discovered, it can't be exploited easily on general conditions. Nonetheless, buffer overflows have been identified by attackers in a surprising array of components and products. In a classic exploit of buffer overflow, data is stored in undersized stack buffer that is sent by the attacker in the first place. As a result, the function's return pointer and all the other information on the call stack is overwritten. The transfer of control to malicious code being contained in the attacker's data is occurred when the function returns as the data has set the value of the return pointer to it.**

I. INTRODUCTION

In computer science, when data is being moved from one place to another, it is temporarily stored in a dedicated region of a physical memory known as the data buffer (or just buffer) [1].

The buffer is mainly used as an intermediate temporary storage in the physical memory where the reading and writing speed of source and destination respectively differ for example a video streaming website. The buffer acts as a queue thus the writing and reading up of data is varied accordingly so as to adjust the timing.

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. Such a contravention of memory safety is a special case [2].

Consider an empty glass of water as a buffer. When this glass is filled with water then the buffer is being filled up, eventually the glass cannot accommodate more water at this point the buffer is full. If more water is being poured, then the water will overflow and the extra water dripping down is now writing now reaching another surface hence writing on another memory location which is outside the given bounds of the buffer hence violates the safety of the memory. This vulnerability is present in many software' and acts as a pearl in the ocean for hackers to exploit.

The exploitation techniques may vary according to the architecture, memory area and operating system. The unfair utilization of memory may either be Heap-based or Stack-based.

II. DISSECTION OF BUFFER OVERFLOW

A. Technical Introduction

Buffer Overflow attacks are common application attacks which are not so easy to find but happen due to procrastination of developers in implementing memory management and security strategies. Most of the Buffer Overflow anomalies happen in most languages when an array is accessed and its bounds are not checked, and also when an attacker enters some data going out of array size, he can write to any available writable memory address. The real application of Buffer Overflow is to insert a malicious code by overwriting function calls and modifying the flow of the program. Common exploits of Buffer Overflow attacks usually work on stacks; the exploit overwrites the return address to some other place to execute its own code in the application [3].

To explain this, we should already know that in real life scenarios, stack increases to lower memory addresses, whenever program calls some function, the address of function call instruction is saved in stack as a return for the function. When the function executes, it allocates local variables, including buffers to stack and they are given a lower address than the return address. So, in this scenario the return address is a certain level above the base address for buffers and if the buffer is overflowing, then it is most likely that an attacker can change return address as well. If the return address is changed to some random value, then it will cause segmentation fault, but if the return address is changed to a certain address where some executable code is present, then that may complete attackers intended tasks with the application. The attack code should be in such a way so that no NULL pointer occurs in the code. Since, the majority of buffer overflow exploits is dependent upon string operations, there are generally two methods of injecting the code. The first method is to put the attack code in the buffer that is being overflowed, then setting return address to the address of the buffer. The second method involves filling the buffer with random memory address and shell codes, and after the return address, the malicious code is place on the stack. Now, in order to jump control to the pointer of the stack, which would actually be pointing to the location just following the return address, an instruction in either system call or normal code is

to be developed which can overwrite the return address, which will perform the above function of jumping control. The interesting part here is that, the byte sequence of machine code which is equivalent to actual command to perform the jump control event will do the work even if it is present without the actual command code. This implies that the return address is actually disguised to be as the correct jump instruction when in reality it has been overwritten by an address which points to the middle of an actual command in the code [4].

It is a very tough process to know what will be the base address of the buffer or what is the return address, so it requires attackers to run exploit locally on the attacker machine to guess the right address.

Usually there are two types of buffer overflow attacks. They are: -

1.Real Stack Overflow: In a real attack, the return address will be replaced by the address of the top of stack, and terrible lines of assembly code will follow it, like invoking another tool. If high privileges are offered to the running corrupt program, the same privilege level will be provided to the running tool. The attacker is at more favorable position since transmission of a little script program is all that it takes for the whole process to complete.

2.Heap Overflow: Programs implement stack and dynamically allocated memory too. The input is copied into the buffer allocated on the heap by the vulnerable program using a call to function similar to strcpy. The data on the heap will be overwritten by the correct type of input, which should be longer than the buffer. The program will neither work as advertised nor as will it crash. The stack is then corrupted by the hacker, who notices this behavior, by trying various inputs until the stack is corrupted. The arbitrary code snippets can be executed by the attacker after the stack is corrupted [5].

III. Prevention And Mitigation

A. Array Bounds Checking

In Computer Programming, any method of detecting if a variable is under specific bounds before it is used is known as Bounds Checking. It is normally used to ensure that the bounds of the array are larger than the variable that is being used as an array index (index checking), or a given type can hold a particular number (range checking).

A great number of languages (Python, Java, C#) by default obviate restrict the programmer from going beyond the end of an array. When this process is performed at runtime, it is known as Bounds Checking.

The commonly associated Programming languages with Buffer Overflows include C++ and C, where overwriting or accessing data protection is not provided by default and no automatic check is provided for checking that the boundaries of that array can hold the data written to it. Buffer Overflows can easily be prevented by bounds checking.

The complete elimination of buffer overflow vulnerabilities is the biggest advantage of array bounds checking. Since, every array and pointer operation is supposed to be checked especially for array- and pointer- intensive programs, these become the most expensive solution and therefore they are not preferred for a production system.

B. Return Address Defender

Return Address Defender or RAD is a user friendly compiler patch which protects programs from buffer overflow attacks by automatically storing a copy of return addresses in a protected area and also when it compiles the applications, it automatically adds protection code to it [6]. The protected programs' source code does not need to be modified provided when RAD is being used to protect a program. The generated binary code is compatible with object files and existing libraries, since the layout of stack frames remains unaltered by RAD.As per experimental performance measurements, RAD prototype which are completely operational show that RAD protected programs experience a small factor of 1.01 to1.31 slow-down.

C. Validating Source of System Calls

Rabek, et al. have proposed a method [7] called Detection of Malicious Executables (DOME) that will be able to detect whether or not malicious code is executing on the system.

This is backboned on the fact that most malware codes will have to make system calls so as to obtain access to resources of systems such as files etc. This technique is bifurcated into two parts: preprocessing and monitoring. In former, the various locations are found out from where all possible system calls are made by preprocessing of the executable file as for each system call the address of the corresponding instruction is stored. This is the address that would be placed on the stack when the system call is called by the program. In latter i.e. monitoring step, a check is made on the return address of the stack with the list of the return addresses retrieved in the former step. If there is a match the execution continues else a malicious code detection is made. Since this will intercept any calls to access any resources protected by the operating system, it will be able to detect the malicious code before it is able to access these resources, thereby preventing damage to the system, or in the case of worm, self-propagation.

The major limitation of this method is that it relies on the malicious code to make system calls. The fact is, much more damage can still be done even if the malicious code doesn't make the system call. Another drawback is that it has no method to deal with wrappers of system calls. If the code makes system calls using wrapper functions, then the attack will avoid detection.

D.StackGuard

Another method that has been proposed is called StackGuard [8]. It is an extension that sandwiches a "canary" between return address on the stack and the local variables. The 4-byte canary is a number that is generated randomly when the programs begins to run. When the execution of the function completes, the canary is first cross-checked with its value before transfer of control to the return address on which the stack was. If it doesn't match, then it may be deduced that

the attack overflowed the buffer in the function which was recently executed and then the program terminated. Thus, this method effectively detects the occurrence of the buffer overflow and eventually kills the process before the attack code can be executed.

This scheme is not completely foolproof however, as outlined by Bulba and Kil3r [9]. If the program is vulnerable to an attack that overwrites a pointer, the attacker can overwrite that pointer with the address of where the return address is located. Then a subsequent string copy operation would overwrite the return address, bypassing the canary altogether.

IV. Conclusion

Buffer Overflow vulnerabilities are a major problem, and will remain to be unless necessary action is taken. Practices such as secure coding can be helpful to any future code that is produced, but still it will not be enough. Mistakes in coding can even be occurred by careful programmers. Reduction of the problems by the insecure code can be achieved by static analysis methods, though it can't completely solve the problem. The act of providing protection at a much lower level, i.e. instruction set randomization and address obfuscation is perhaps the best way to obviate insecurities caused by buffer overflow. The attacking of the most vulnerable code can be made extremely difficult by the combination is the above two methods.

References

[1] Wikipedia,"Databuffer",Available: https://en.wikipedia.org/wiki/Data_buffer

[2] Wikipedia,"BufferOverflow",Available: https://en.wikipedia.org/wiki/Buffer_overflow

[3] M. Shaneck, "An Overview of Buffer Overflow Vulnerabilities and Internet Worms,".

[4] Aleph One, "Smashing the Stack for Fun and Profit", Phrack, Volume 7, Issue 49J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[5] T. Schwarz, "Buffer Overflow attack", SCU,2004, Available: http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html

[6] Tzi-cker Chiueh, Fu-Hau Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks,"International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April 2001,Available: http://www.ecsl.cs.sunysb.edu/RAD/

[7] J. Rabek, R. Khazan, S. Lewandowski, R. Cunningham, "Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code," In Proceedings of the 2003 ACM workshop on Rapid Malcode, October 2003.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. "Stackguard: Automatic dectection and prevention of buffer-overflow attacks." In Proceedings of the 7th USENIX Security Symposium, January 1998.

[9] Bulba and Kil3r, "Bypassing StackGuard and StackShield," Phrack, Volume 5, Issue 56.