Securing Server System from Buffer Overflow vulnerability using Vel-Alagar Algorithm

*P.VADIVELMURUGAN #K.ALAGARSAMY

*Research Scholar, Department of Computer Center, Madurai Kamaraj University, Madurai, Tamil Nadu, India
#Associate Professor, Department of Computer Center, Madurai Kamaraj University, Madurai, Tamil Nadu, India

ABSTRACT-

Now a day's buffer overflows have take place the most common target for network based attacks. The main proliferation method used by worms, malicious codes and improper coding by developers. Many techniques have been developed to secure servers, but the conciliation of the vulnerable codes make buffer overflow attacks again. Buffer overflows occur it stop system functions and crash the memory. In the face of automatic function execution, cyclic attacks causes due to malicious codes, and it leads to repeated restarts of the victim application, and it makes service unavailable. In this research paper we have a hopeful new move towards to learn the characteristics of buffer overflow vulnerability, and we develop a suitable algorithm to avoid and prevent the buffer overflows.

KEYWORDS: Buffer overflows attack, malicious code, Securing stack, vulnerability, secure server.

I INTRODUCTION

Buffer overflows attacks accepted by computer security professionals to be one of the most vulnerable threats to the systems security. A rudiment of buffer overflow attack is stack and heap. In C, C++ code, application library components and dynamic components of application software. The coding practices that will avoid buffer overflow problems, and tools can use to detect buffer overflows, and make your system secure. Every time a software program request input (from a users, from a system file, over through a network, or by some other means), there is a potential to receive improper data. For example, the input data might be longer than what you have reserved space for in memory.

The given input data size is longer than will fit in the reserved memory space, if you do not curtail it, that data will overwrite other data in memory if allotted. When it happens, it is called a buffer overflow. If the memory overwritten contained data essential to the function of the program, this overflow causes a error that, being intermittent, might be very hard to find. If we entering new data in its place of existing, data include the address of other code to be executed and the user has complete this intentionally, the user can point out the malicious code that your program will then execute.

II STACK OVERFLOWS

All most all the operating systems have an application has a stack (multithreaded applications programs have one stack per thread). This stack contains memory storage for locally assigned data. The stack is divided up into small units called stack frames. Every stack frame holds all data exact to a particular call to a particular function. This type of data characteristically includes the function's parameters, the complete details of local variables within that function, and linkage information details - that is, the address of the function calls itself, where the execution continues when the function returns). Depending on the compiler flags segment, it may also contain the address of the top of the next stack frame. The exact data content and order of data on the stack determined by on the operating system and CPU architecture.

Every time a program function is called, a new fresh stack frame is added to the apex of the stack. Each time a program function returns, the vertex of stack frame is removed from the memory. If any specified point in execution, an application can directly access the data in the top most segment of stack frame. (Pointers can get around this, but it is not a good idea to do so). This type of design creates recursion likely possible because each nested loop is calling to a program function gets its own individual copy of local variables and parameters. In commonly, an application should check all kind of input data to make sure it is appropriate condition of the file. In many cases, programmers assuming that the user will not do anything difficult. This becomes a severe

problem when the application stores that data into a fixed-length of buffer. If the user is malicious (or opens a file that contains malicious code), they may provided data that is longer than the size of the buffer. Because the function holds only a confined amount of memory space on the stack for this data, the data should overwrite other data on the stack.

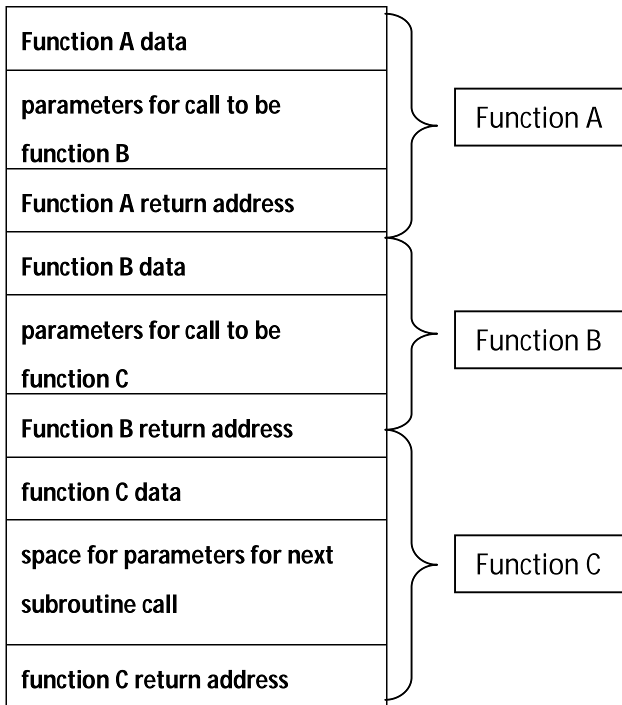

Figure 1: Schematic view of stack function in buffer

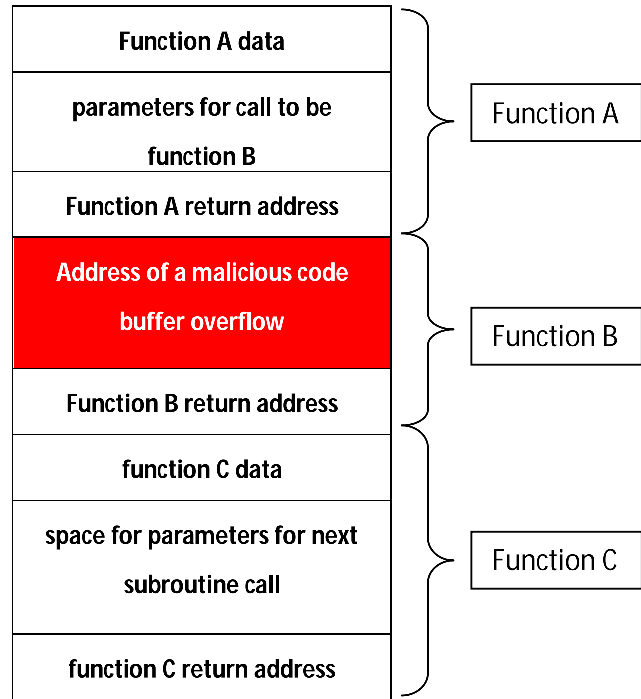

Figure 2: Stack after malicious buffer overflow crash in memory

III CALCULATING BUFFER SIZES

If a software running with fixed length buffers, it is always use to calculate the size of a buffer, and then if we supply more data into the buffer than it can hold. Even if it's really assigned a static size to the buffer, either the code in the future might be change the buffer size but it fail to change every case where the buffer is written to.

Table shows two ways of allotting a character buffer 1024 bytes in length, and checking the length of an input string value, and replication it to the buffer memory.[3]

Table 1: Calculating buffer size

Instead of this:	Do this:

char buf[1024];
...
if (size <= 1023) {
	#define BUF_SIZE 1024
	...
	char buf[BUF_SIZE];
	...
	if (size < BUF_SIZE) {
	...
	}

char buf[1024]; ... if (size < 1024) { ...}	char buf[1024]; ... if (size < sizeof(buf)) {… .}

The two oddments on the left side are safe as long as the original statement of the buffer size is never changed. However, if the buffer size gets altered in a later edition of the program devoid of changing the test, then a buffer overflow will result.
The two oddments on the right side confirm safer versions of this code. In the beginning version, the buffer memory size is set using a constant that is set in another place, and the verify uses the same constant. In the next version, the buffer is set to 1024 bytes, but the check calculates the real size of the buffer. In either of these oddments, varying the original size of the buffer does not invalidate the check.

IV BUFFER OVERFLOW ATTACKS HELD PLACES IN NETWORKING OPERATING SYSTEMS

Table 2: Different types of services and vulnerabilities obtainable on the network operating system [5]

Attack types	Attacking held is systems
iis_bof(*h*)	IIS web server has buffer overflow vulnerability on host *h*
exchange_ivv(*h*)	Exchange mail server has input validation vulnerability on host *h*
squid_conf(*h*)	Squid web proxy is misconfigured on host *h*
licq_ivv(*h*)	LICQ client has input validation vulnerability on host *h*
sshd_bof(*h*)	SSH server has buffer overflow vulnerability on host *h*
scripting(*h*)	HTML scripting is enabled on host *h*
ftp(*h*)	FTP service is running on host *h*
wdir(*h*)	FTP home directory is writable on host *h*
fshell(*h*)	FTP user has executable shell on host *h*
xterm_bof(*h*)	*xterm* program has buffer overflow vulnerability on host *h*
at_bof(*h*)	*at* program has buffer overflow vulnerability on host *h*

The intruder can use generic buffer overflow exploits, illustrate as follows:[5]

- *iis_r2r* - Buffer overflow vulnerability in the Microsoft IIS web server allows remote intruders to gain root shell on the target host.
- *ssh_r2 r* - Buffer overflow vulnerability in the SSH server allows remote intruders to gain root shell on the target host
- *xterm_u2r* -Buffer overflow vulnerability in the *xterm* program allows local users to gain root shell on the target host.
- *at_u2r* -Buffer overflow vulnerability in th*at* program allows local users to gain root shell on the target host.

Table 2: An experimental result exploits held conditions in networking operating systems

Exploit	Preconditions	Post conditions
iis_r2r(*hs, ht*)	*iis_bof*(*ht*) *C*(*hs*, *ht*, *http*) $plvl(hs) \geq user$ $plvl(ht) < root$	*iis*(*ht*) $plvl(ht) := root$
sshd_r2r(*hs, ht*)	*sshd_bof*(*ht*) *C*(*hs*, *ht*, *ssh*) $plvl(hs) \geq user$ $plvl(ht) < root$	*ssh*(*ht*) $plvl(ht) := root$
xterm_u2r(*ht, ht*)	*xterm_bof*(*ht*) $plvl(ht) = user$	$plvl(ht) := root$
at_u2r(*ht, ht*)	*at_bof*(*ht*) $plvl(ht) = user$	$plvl(ht) := root$

V LEMMA 1: VEL-ALAGAR ALGORITHM

Current position of stack is in memory is denoted as X_i and the Rate of change stack position is consider as V_i if stack pointer is grow. If the buffer overflow occurs in the stack pointer need to control or hold the position of stack is denoted as Best position Y_i. If spot 'i' the best position reached by that spot at a given time. Let "f" be the objective function to be maximized. The best position of a particle at iteration or time step "t" is updated as every 5 seconds

$$y_i(t)= y_i(t\text{-}1) \text{ if } f(x_i(t)) \leq f(y_i(t\text{-}1))$$
$$x_i(t) \quad \text{if } f(x_i(t)) \leq f(y_i(t\text{-}1)) \qquad \text{----------}(1)$$

The best is determined from the entire buffer by selecting the best control position. This position is denoted as ŷ.The equation that manipulates the update equation and is stated as

$$v_{ij}(t+1)= v_{ij}(t)+c_1r_1(t)(y_{ij}(t)\text{-}x_{ij}(t))+ c_2\, r_2(t)(\, \hat{y}_j(t)\text{-}x_{ij}(t)) \quad \text{----------}(2)$$

Where $v_{ij}(t + 1)$ is the position change for the jth dimension, j = 1, 2… n.

c1 and c2 are the stepping up constants, where the first control, controls the maximum step size towards the meticulous part of the buffer(process id), while the second control the maximum step size towards the buffer consuming level - one iteration. $r1_j$ (t) and $r2_j$ (t) are two random values in the range [0, 1] and given the algorithm to control the buffer overflow.

Memory position changes v_i updates on each dimension can be secured with a user defined changing speed up the vulnerable detection Vmax, which would prevent them from exploding, thereby causing premature convergence. Each portion of buffer updates its position using the following equation:

$$x_i(t+1)=x_i(t)+v_i(t+1) \qquad \text{------------}(3)$$

In the vel-alagar algorithm, the memory part 'I' show the best position is changing to its new position x_i (t + 1). After the new position is calculated for each part of the buffer memory position, the iteration counter increases and the new buffer memory positions are evaluated. This process is repeated until some convergence conditions are satisfied.

Therefore, the buffer memory remains continuous assessed. Since each Vij is a true value, a memory mapping needs to be defined from Vij to a probability in the range [0 to n]. This is done by using an algorithm function to control the congestion speeds into a [0 to n] range. The sigmoid function is defined as

$$Sig(v)=1/1+e^{-v} \qquad \text{--------------}(4)$$

The equation for updating the new positions is then swapped by the following probabilistic update equation:

$$X_{ij}(t+1)=0 \quad \text{if} \quad r_{3j}(t)\geq sig(v_{ij}(t+1)) \qquad \text{---------------}(5)$$
$$n \text{if} r_{3j}\,(t) < sig(v_{ij}(t+1))$$

Where $r3_j\,(t)$ is a random value in the range [0 to n].In the vel-alagar algorithm behavior it protect from buffer overflow attack. With the speed of deduced as a probability of change, speeding up the vulnerable detection V_{max}, sets the minimal probability of causing the buffer overflow attack.

VI CONCLUSION

Every attack circumstances are a series of exploits initiated by an intruder for a particular goal. To prevent an exploit, the security analyst must deploy an appropriate countermeasure such as the installing firewall or add patch the vulnerabilities that made this exploit possible. The group of possible attack scenarios in a computer system can be represented by malicious code developers. To overcome or minimization the buffer overflow vulnerability the lemma 1, vel-alagar algorithm is a new step of solution to control and prevent the buffer overflow vulnerability in the networking operating system.

REFERENCES

1. Eugen Leontie, Gedare Bloom, Olga Gelbart, Bhagirath Narahari and Rahul Simha Department of Computer Science,The George Washington University,Washington, DC 20052 A Compiler- Hardware Technique for Protecting Against Buffer Overflow Attacks

2. I. simon. “A comparative analysis of methods of defense against buffer overflows attackshttp://www.mcs.csuhayward.edu/~simon/security/boflo.html, January 2001.

3. https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html

4. J. McCarthy “Take Two Aspirin, and Patch That System – Now”, *SecurityWatch*, August 31, 2001

5. Mahdi Abadi and Saeed Jalili (2011). A Memetic Particle Swarm Optimization Algorithm for Network Vulnerability Analysis, Evolutionary Algorithms, Prof. Eisuke Kita (Ed.), ISBN: 978-953-307-171-8, InTech,http://www.intechopen.com/books/evolutionary-algorithms/a-memetic-particle-swarmoptimization- algorithm-for-network-vulnerability-analysis

6. Mathematical Symbol Table http://xaravve.trentu.ca/mascot/handbook/SEC_symbols.pdf

7. peter Silverman and Richard Johnson “A comparison buffer overflow prevention, implementation and weakness”
8. P. Vadivel Murugan, and K. Alagarsamy "Buffer Overflow Attack Vulnerability in Stack.", International Journal of Computer Applications 13.5 (2011): 1-2.

9. Seema Yadav, Khaleel Ahmad and Jayant Shekhar. “Classification and Prevention Techniques of Buffer Overflow Attacks” Proceedings of the 5th National Conference; INDIACom-2011 Computing For Nation Development, March 10 – 11, 2011

10. Tz-Rung Lee1, Kwo-Cheng Chiu1, and Da-Wei Chang A. Hua and S.-L. Chang **“**A Lightweight Buffer Overflow Protection Mechanism with Failure-Oblivious Capability”, (Eds.): ICA3PP 2009, LNCS 5574, pp. 661–672, 2009. Springer - Verlag Berlin Heidelberg 2009

11. Vadivel Murugan.P K.Alagarsamy “Averting Buffer Overflow Attack in Networking OS using – BOAT Controller”, *International* Journal of Computer Trends and Technology (IJCTT) – volume 4 Issue 7–July 2013

12. V.Selvi Dr.R.Umarani, “Comparative Analysis of Ant Colony and Particle Swarm Optimization Techniques” International Journal of Computer Applications (0975 – 8887) Volume 5– No.4, August 2010