MCAIDS- Machine Code Analysis Intrusion Detection System

Sachin B. Jadhav
Department of CSE,
SIRT Bhopal,RGPV,India,

Abstract

MCAIDS - Machine Code Analysis Intrusion Detection System for blocking code-injection buffer overflow attack messages targeting at various Internet services such as web service. With the increasing access of Internet,the Internet threat takes a form of attack, targetting individuals users to gain control over network and data. Buffer overlow is one of the most occuring security vulnerability in computer's world. Buffer overflow attack typically contains executables where as legitmate client request never contains executables in most Internet services. MCAIDS blocks attack by detecting the presense of code. MCAIDS uses new data - flow analysis technique called code abstraction. MCAIDS is signature free , thus it can block new and unknown buffer overflow attack. This MCAIDS simulate by using Network Simulater NS2 on the linux platform to analyze the expected results.

Keywords - Buffer overflow, Buffer overflow attack, Intrusion detection, Computer security, signature free.

"1 . INTRODUCTION"

Buffer overflow violates the boundary of Computer security. Buffer overflow which is one of the threats which occurs due to writing large amount of data to fixed sized buffer and the data which overruns is being adjusted to another memory region. Although tons of research has been done to tackle buffer overflow attacks, existing defences are still quite limited in meeting four highly desired requirements: (1) simplicity in maintenance; (2) transparency to existing (legacy) server OS, application software, and hardware; (3) resiliency to obfuscation; (4) economical Internet-wide deployment [1]. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

Existing defences are limited in meeting these four requirements. Existing buffer overflow defences are categorized into six classes. (A) Finding bugs in source code. (B) Compiler extensions. (C) OS modifications. (D) Hardware modifications. (E) Defence-side obfuscation. (F) Capturing code running symptoms of buffer overflow attacks [5], [6], [7], [8]. We may briefly summarize the limitations of these defences in terms of the four requirements as follows: 1) Class B, C, D, and E defences may cause substantial changes to existing (legacy) server Operating Systems, application software, and hardware, thus they are not transparent. Moreover, Class E defences generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination overhead as economical deployment. 2) Class F defences can be very secure, but they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. As a result, Class F defences have limited transparency and potential for economical deployment. 3) Class A defences need source code, but source code is unavailable to many legacy applications. Besides buffer overflow defences, worm signatures can be generated and used to block buffer overflow attack packets [9], [10], [11]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.
To overcome above limitations the buffer overflow attack blocker systems implementation will be demonstrated in this paper.

"2.NEED AND SIGNIFICANCE OF THE WORK"

This section demonstrate the need and significance of the work. Here some basic definitions have been provided about buffer overflow. This section also demonstrate why buffer overflow occurs and what is the impact of buffer overflow on the Internet services.

2.1 Buffer

In computer science, a buffer is usually a contiguous computer memory area or block of fixed size to store data or to hold some inputs or outputs. This data can be integers, floating points, characters, or even user defined data types.

2.2 Buffer overflow

In most computer languages, a buffer is represented as an array. If programs don't check the size of the user input for a buffer array and the size of the input data is larger than the size of the buffer array, then areas adjacent to the array will be overwritten by the extra data. The lack of such ``bound checks'' creates the breeding ground for buffer overflow attacks. The general idea is to give servers very large strings that will overflow a buffer. It is a phenomenon to overflow a buffer so that it overwrites the return address. When the function is done it will jump to whatever address is on the stack. We put some code in the buffer and set the return address to point to it! This is a called Smashing the stack. There are two ways to detect Buffer overflows: one way is to look at the source code and another way is to feed the application with huge amount of data and check the abnormal behaviour.

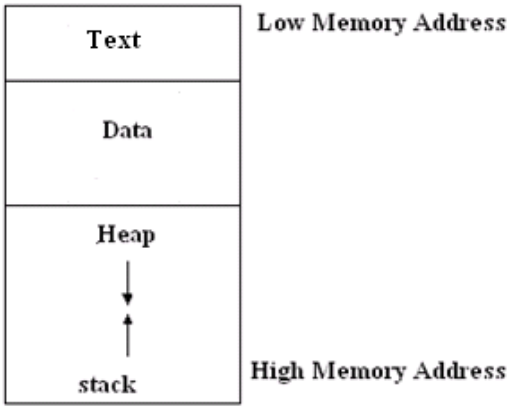
2.3 Buffer-overflow attack

A buffer-overflow attack is an attack that uses memory-manipulating operations to overflow a buffer which results in the modification of an address to point to malicious or unexpected code [2].

2.4 Buffer overflow on stack

In computer memory, a process is organized into four regions: text, data, heap and stack. These regions are located in different places and have different functionalities. Figure 1 shows the organization of a process in memory. Although all of them are important, we only give a brief introduction of the text, data and heap regions. We focus on the stack region, which is the key region related to the buffer overflow vulnerability discussed in this project. The text region stores instructions and read-only data. The data region consists of initialized and uninitialized data.

A stack is a widely used abstract data type in computer science. A stack has the unique property of last in, first out (LIFO), which means that the element that is placed in last will be moved out first.



"Figure.1 Organization of Process in Memory"

There are many operations associated with a stack, of which the most important are PUSH and POP. PUSH puts an element on the top of the stack and POP takes an element from the top of the stack. The kernel dynamically adjusts the stack size at run time.

Modern computer languages are high-level. Such languages apply functions or procedures to change a program's execution flow. In a low-level language (such as assembly), a jump statement changes program flow. Unlike jump instruction, which jumps to another place and never go back, functions and procedures will return control to the appropriate location in order to continue the execution. The stack is used to achieve this effect. More precisely, in memory, a stack is a consecutive block that contains data, which can be used to allocate the function's local variables, pass function's parameters, and return a function's result.

In memory, the stack boundary is represented by the Extended Stack Pointer (ESP) register. The ESP points to the top of the stack. In most of architectures, including Intel Architecture 32bit (IA32), the ESP points to the most recently used stack address. In other architectures, the ESP points to the first address that is free. When a PUSH or POP instruction is used to add or remove data, respectively, the ESP moves to indicate where the

new top of the stack is in memory. Based on the different implementations, the stack can either grow down toward the lower memory addresses or up toward the higher memory addresses.

Figure 2 demonstrates a simple C program, bo.c, which contains a function called foo that illustrates what happens in the stack when a function is called Here, when foo is called, there are four steps that need to be done in the stack.

(1)PUSH the parameters x and y of the function foo backwards onto the stack.

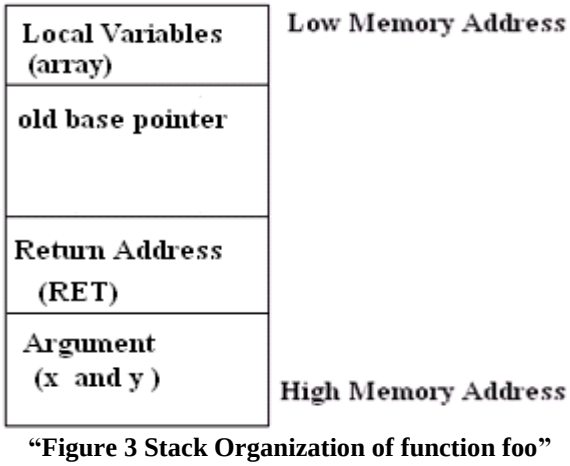
(2)The return address, RET, is put into the stack, Which contains the address of next machine instruction to be executed. Once the execution of the foo function has completed, execution returns to the instruction address stored in RET. In our example, the instruction address of *return 0* is stored in RET.

(3)Finally, any local variable in foo (in the example in Figure 2, *array*) is pushed onto the stack. Figure 3 shows the stack organization when foo is called.

```
#include <stdio.h>
#include <string.h>
void foo (int x, int y)
{
  char array[8];
 printf("Enter Serial Number\n");
   scanf("%s", array);
}
      if(!strncmp(array, "S123", 4))

 printf("\nSerial number is correct.\n\n");
}
int main(){
  foo(3, 4);
  return 0;
}
```

"Figure 2. A Program Example with A Function"



"Figure 3 Stack Organization of function foo"

In languages like C and C++, there is no build-in mechanism for buffer boundary checking and consequently, the task of bounds-checking falls to the programmer. If a C language developer allows more data to be copied to an array than it can hold, the data will fill the array and overwrite the contents following the array. The program will compile but might crash or otherwise behave badly when it is executed.

An attacker can sometimes take advantage of a buffer overflow flaw. Since buffer overflow is the most serious vulnerability in computer security, many computer scientists have been working on it trying to solve the problem. Presently, there are three common ways to defend against buffer overflow vulnerabilities and exploits: writing correct code, non-executable buffers, and array bounds checking by the compiler.

"3. LITERATURE SURVEY"

In November 1988, the "Morris worm", attacked VAX and Sun machines and prevented a great number of users from accessing machines via the Internet. In July 2001, the "Code Red" worm successfully exploited more than 300,000 computers that used Microsoft's IIS Web Server. In January 2003, another worm, "Slammer", attacked Microsoft SQL Server 2000 Slammer crashed parts of the Internet in South Korea and Japan, interrupted Finnish phone system, and slowed down the U.S. networks for airline reservation as well as credit card transaction [17]. All these attacks exploited buffer overflow vulnerabilities. Although computer technology has advanced, the buffer overflow vulnerability remains a major problem. Moreover, with the increasing number of computers used in daily life, buffer overflow attacks have the potential to do even greater damage.

Therefore many persons have been written about this. Several people have done precious work in this area including Xinran wang, chi-chun pan, peng Liu, and Sencun Zhu, They propose SigFree, an online signature-free out-of-the-box application-layer method for blocking code-injection buffer overflow attack messages targeting at various Internet services such as web service [1]. Krerk Piromsopa and Richard J. Enbody propose a framework for protecting against buffer overflow attack and many more [2].

"4.OBJECTIVE"

Buffer overflow problems are well known. Several studies have been conducted to detect and prevent buffer overflows. The purpose of this work is

to study security problems and to detect buffer overflow attack using various techniques.
The following are the major objective:
1] To study the vulnerabilities and attacks such as buffer overflow.
2] To study some of the existing techniques used for detecting and preventing buffer Overflow.
3] To develop a new efficient techniques for detecting and preventing the buffer overflow.
4] To analyze this techniques.

"5. PROPOSE METHODOLOGY"

Existing prevention/detection techniques of buffer overflows can be roughly broken down into six classes: Class A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools have been developed [1] ,[9] ,[10] ,[11]. The bug-finding techniques used in these tools, which in general belong to static analysis, include but are not limited to model checking and bugs-as-deviant-behavior. Class A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, MCAIDS will handle machine code embedded in a request (message). Class B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler" [1] , [3]. Class B techniques require the availability of source code. In contrast, MCAIDS does not need to know any source code. Class C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such. Class 1C techniques need to modify the OS. In contrast, MCAIDS does not need any modification of the OS. Class D: Hardware modifications. A main idea of hardware modification is to store all return addresses on the processor [1] ,[13]. In this way, no input can change any return address. Class E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX [1], [14]. Address-space randomization can detect exploitation of all memory errors [1], [15],[16]. Instruction set randomization can detect all code-injection attacks, whereas MCAIDS cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. Class F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflow is a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class B, Class C, and Class E techniques can capture some-but not all-of the running symptoms of buffer overflows. For example, accessing non executable stack segments can be captured by OS modifications; compiler modifications can detect return address rewriting; and process crash is a symptom captured by defense-side obfuscation [1] .

Class F techniques can block both the attack requests that contain code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. MCAIDS is signature free and does not need any changes to real-world services.

5.1 Machine code analysis

Although source code analysis has been extensively studied , in many real-world scenarios, source code is not available and the ability to analyze binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyze Obfuscated binaries and (P3) to identify and analyze the code contained in buffer overflow attack packets [1].

The purpose of MCAIDS is to see if a message contains code or not, not to determine if a piece of code has malicious intent or not. MCAIDS disassemble binary code. MCAIDS differs from P1 and P2 techniques.

5.2 Basic Definitions

Definition 1 (Instruction sequence). An instruction sequence is a sequence of CPU instructions, which has one and only one entry instruction and there exists at least one execution path from the entry instruction to any other instruction in this sequence. A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program Those instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. we call them random instruction sequences, whereas use the term binary executable code to refer to a fragment of a real program in machine language [1].

Definition 2 (Instruction flow graph).
An instruction flow graph (IFG) is a directed graph G = (V, E) where each node v Є V corresponds to an instruction and each edge e = (vi, vj) Є E

corresponds to a possible transfer of control from instruction vi to instruction vj [1].

Definition 3 (extended IFG). An extended IFG (EIFG) is a directed graph G = (V, E) which satisfies the following properties: each node v Є V corresponds to an instruction, an illegal instruction (an "instruction" that cannot be recognized by CPU), or an external address (a location that is beyond the address scope of all instructions in this graph); each edge e = (vi , vj) Є E corresponds to a possible transfer of control from instruction vi to instruction vj, to illegal instruction vj, or to an external address vj [1].

5.3 Instruction sequence Distiller

This section first describes an effective algorithm to distill instruction sequences from requests.

To distill an instruction sequence, we first assign an address (starting from zero) to every byte of a request, where address is an identifier for each location in the request. We use the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from an N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences[1].

One drawback of the recursive traversal algorithm is that the same instructions are decoded many times .The main aim of this paper to implement the memorization algorithm by using some tool command language with proper data structure to reduce running time.

5.3.1 Excluding Instruction Sequences

Distilling instruction sequence may output many instruction sequences at different entry points. Next, we exclude some of them based on several heuristics. Here, excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any) [1].

Step 1: If instruction sequence S_a is a subsequence of instruction sequence S_b, we exclude S_a. The logic behind for excluding S_a is that if S_a satisfies some characteristics of programs, S_b also satisfies these characteristics with a high probability.

Step 2: If instruction sequence S_a merges to instruction sequence S_b after a few instructions and S_a is no longer than S_b, we exclude S_a. It is reasonable to expect that S_b will preserve S_a's characteristics. Many distilled instruction sequences are observed to merge to other instruction sequences after a few instructions.

Step 3: For some instruction sequences, when they are executed, whichever execution path is taken, an illegal instruction is inevitably reached. We say an instruction is inevitably reached if two conditions hold. One is that there are no cycles (loops) in the EIFG of the instruction sequence; the other is that there are no external address nodes in the EIFG of the instruction sequence.

5.4 Instruction Sequence Analyzer

Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may easily obfuscate his program by introducing enough data flow anomalies. Here, we use the detection of data flow anomaly in a different way called code abstraction. We observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path has a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.

Here we use the algorithm to check if the number of useful instructions in an execution path exceeds a threshold [1].

5.5 System Requirements

Operating System: Linux
Language : Tool command Language.
Development Tools: NS 2

"6. EXPECTED OUTCOME"

Here, we first tune the parameter for MCAIDS method based on some training data ,then evaluate and compare the performance of these methods in checking messages collected from various sources. We use the threshold value to determine if a request contains code or not. Here we set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. Here, we choose HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies contain more binaries.

"7. CONCLUSION"

We propose MCAIDS, an signature-free blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. MCAIDS does not require any signatures, thus it can block new unknown attacks. MCAIDS is good for economical Internet-wide deployment with little maintenance cost and low performance overhead.

REFERENCES

[1] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu , "SigFree: A Signature-Free Buffer Overflow Attack Blocker", *IEEE Transactions on dependable and secure computing*, vol. 7, No. 1, January-March 2010

[2] Krerk Piromsopa, Member, IEEE, and Richard J. Enbody, *"Buffer-Overflow Protection: The Theory ,"*Member, IEEE

[3] B.A. Kuperman, C.E. Brodley, H.Ozdoganoglu, T.N. Vijaykumar,and A. Jalote, *"Detecting and Prevention of Stack Buffer Overflow Attacks,"* Comm. ACM, vol. 48, no. 11, 2005.

[4] Bindu Madhavi , Padmanabhuni and Hee Beng Kuan Tan,*"Defending against Buffer – Overflow ulnerabilities",*Nanyang Technological University,Singapore,

[5] J.Newsome and D. Song, *"Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,"* Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang,and P. Barham, *"Vigilante: End-to-End Containment of Internet Worms,"* Proc. 20thACMSymp. Operating Systems Principles (SOSP),2005.

[7] Z.Liang and R. Sekar, *"Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers,"* Proc.12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[8] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, *"Automatic Diagnosis and Response to Memory Corruption Vulnerabilities,"* Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[9] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, *"A First Step towards Automated Detection of Buffer Overrun Vulnerabilities,"*Proc. Seventh Ann. Network and Distributed System Security Symp.(NDSS '00), Feb. 2000.

[10] D. Evans and D. Larochelle, *"Improving Security Using Extensible Lightweight Static Analysis," IEEE Software,* vol. 19, no. 1, 2002.

[11] H. Chen, D. Dean, and D. Wagner, *"Model Checking One Million Lines of C Code,"* Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS), 2004

[12] Zhimin Gu Jiandong Yao Jun Qin*,"Buffer Overflow Attacks on Linux Principles Analyzing and Protection",* Department of Computer Science, Beijing Institute of Technology (Beijing 100081)

[13] J. McGregor, D. Karig, Z. Shi, and R. Lee, *"A Processor Architecture Defense against Buffer Overflow Attacks," Proc. Int'l Conf. Information Technology: Research and Education (ITRE '03)*,pp. 243-250, 2003

[14] Pax Documentation, http : // pax.grsecurity.net/docs/ pax.txt, Nov. 2003.

[15] G. Kc, A. Keromytis, and V. Prevelakis, *"Countering Code- Injection Attacks with Instruction-Set Randomization,"* Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[16] E.Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi*,"Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks,"* Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[17] Wheeler, D. *"Preventing Today's Top Vulnerability."* http://www-106.ibm.com /developerworks/linux/library/l-sp4.html