International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 4, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

Code-injection Buffer Overflow Attack Blocker

Ms. Mayura A. Kathwate, Mr. D. K. Chitre

***Abstract* — Code injection buffer overflow attack blocker propose a real time, application layer blocker for preventing buffer overflow attacks and all types of code injection message. It can filter out code-injection and buffer overflow attack messages targeting at various Internet services such as web service. This paper is specifically works on the observation that buffer overflow attacks typically contain executables whereas lawful client requests never contain executables in most internet services, it blocks attacks by detecting the presence of code. System first simply dissembles and extracts instruction sequences from a request, then applies a technique called code abstraction, which uses data flow anomaly to remove useless instructions in an instruction sequence. Finally it compares the number of useful instructions to a threshold to determine if this instruction sequence contains code. Code injection buffer overflow attack blocker does not work on any pre-known pattern, thus it can block any new and unknown buffer overflow attacks. As there is no need to do any modifications in software or hardware at server so blocker is transparent to the servers being protected. Its deployment and maintenance cost is also very less so it is good for deployment in internet services. We proposed code injection buffer overflow attack blocker; could block all types of code injection attack packets, with less throughput degradation to normal client requests.**

***Keywords* — Security, Intrusion Detection, Buffer overflow attacks, code injection attacks.**

I. INTRODUCTION

One of the most serious vulnerabilities in cyber security is buffer overflow. The root cause for most of the cyber attacks such as server security destroyed in worms, zombies are Buffer overflow vulnerability [1]. When a fixed size buffer has too much data copied into it during a program execution buffer overflow occurs. This allows the data to overwrite into adjacent memory locations, which affects the program execution depending on what is stored over there. By considering different buffer overflow attacks it is observed that buffer overflow attacks do not always carry binary code in attacking request packets but code-injection buffer overflow attacks like stack smashing contents binary code which is mostly occurred in the real world. The sensitive information of the client is connected to a backend database of web servers which provide services. As demand of web services increased by the customers, deployment of web applications also increased. Because of that there has been increase in the number of attacks targeting such applications. It is observed that most of the cyber attacks occur at the application layer and the sites that they visited were vulnerable to web attacks.

Buffer overflow vulnerability causes overwriting the contents to the adjacent memory locations while writing data to a program buffer exceeding the allocated size. The overwriting might corrupt the sensitive information or variables of the buffer like return address of a function or the stack frame pointer. Buffer overflow can occur due to vulnerable ANSI C, C++ library function calls, lack of null characters at the end of buffers, accessing buffer through pointers and aliases, logical errors, placement of new operator and insufficient checks before accessing buffers in program code.

An example of buffer overflows as shown in Figure1 in a C code snippet. The function foo has a buffer named *buf* that is located inside the stack region. The valid location of this buffer is between *buf[0]* and *buf[15]*. The variable *var1* is located immediately after the ending location of the buffer followed by the stack frame pointer (*sfp*) and the return address (*ret*) of the function *foo* as shown in Figure2. The return address indicates the memory location where the next instruction is stored and is read immediately after the function is executed.

```
1. void foo (int a) {
2.     int var1;
3.     char buf [16];
       ...
   }
```

Figure 1 C code snippet of *foo* function

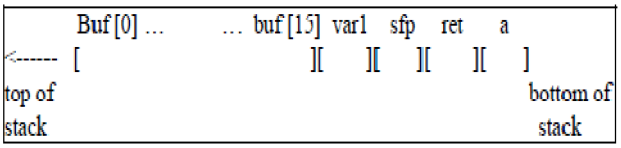

Fig.2. Stack layout of *foo* function

A buffer overflow might happen during reading or writing operations. Writing past the *buf* by at least one byte corrupts the value of *var1* assuming no padding performed by a compiler. If overwriting spans more than one byte in stack, it might modify the return address (*ret*) of the function *foo*. As a result, when the function tries to retrieve the next instruction after its execution, the modified location might not fall within the valid address space of the program. This might result a segmentation fault and the program crashes.

II. LITERATURE SURVEY

Classification of Buffer Overflow Vulnerability: Existing prevention or detection techniques of buffer overflows can be roughly broken down into following classes:

I) Finding bugs in source code: Mainly programming bug's causes Buffer overflow. Depending on programming bugs, various bug-finding tools have been developed. The bug-finding techniques used in these tools, which in general belong to static analysis, they can check bugs. Class I techniques are designed to handle source code

only, and they do not ensure completeness in bug finding. In contrast, Code injection buffer overflow attack blocker handles machine code embedded in a request (message).

II) Compiler extensions: "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler"[4]. Three such compilers are StackGuard[15], ProPolice[16], and Return Address Defender (RAD)[17]. DIRA[7] is another compiler that can detect control hijacking attacks, identify the malicious input, and repair the compromised program. Class II techniques require the availability of source code. In contrast, Code injection buffer overflow attack blocker does not need to know any source code.

III) OS modifications: Modifying some aspects of the operating system may prevent buffer overflows such as Pax[18] , LibSafe[19], and e-NeXsh[20] .In contrast, Code injection buffer overflow attack blocker does not need any modification of the OS.

IV) Hardware modifications: A main idea of hardware modification is to store all return addresses on the processor. In this way, no input can change any return address.

V) Capturing code running symptoms of buffer overflow attacks: Basically, buffer overflows are a code running symptoms. If such unique symptoms can be precisely captured, all buffer overflows can be detected.

All above techniques can detect some of the running symptoms of buffer overflows but not all. For example OS modifications can capture accessing non executable stack segments, detecting return address rewriting can be done by the compiler modifications and process crash is a symptom captured by defense-side obfuscation. To achieve all the symptoms of the buffer overflow, dynamic data flow analysis techniques where proposed in TaintCheck[5]. They can detect buffer overflows during running time but it may cause significant runtime overhead.

III. PROPOSED METHOD

There is static or dynamic method to detect data flow anomalies in the software reliability and testing field. Static methods are not suitable in our case due to its slow speed; dynamic methods are not suitable because it requires real execution of a program with some inputs. Existence scheme is rule-based, where as Code injection buffer overflow attack blocker is a *generic* approach which does not require any pre-known patterns. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to see if the packet really contains code. Although, they used a special rule to detect polymorphic exploit code which contains a loop and they mentioned that the above rules are initial sets and may require updating with time, it is always possible for attackers to bypass those pre-known rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, code injection buffer overflow attack blocker exploits a different data flow analysis technique. We proposed Code injection buffer overflow attack blocker, a real time application layer blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats, to various Internet services. Code injection buffer overflow attack blocker does not require any signatures, thus it can block new, unknown attacks.

Buffer overflow vulnerability of a web server can be occurred by sending a request, which contains a malicious payload. There are several HTTP request methods among which GET and POST are most often used by attackers. Some web server such as Microsoft IIS read a request body according to the request headers instructions though HTTP 1.1 does not allow GET to have a request body. The vulnerability determines the position of a malicious payload. A malicious payload may be included in the Request-URI field as a query parameter. The maximum length of Request-URI is limited; the size of a malicious payload, therefore the behavior of such a buffer overflow attack is also limited. It is more common that a buffer overflow attack payload is embedded in Request-Body of a POST method request. Technically, a malicious payload may also be embedded in Request-Header, although this kind of attacks has not been observed yet. In this work, we assume an attacker can use any request method and embed the malicious code in any field. Figure 3 depicts the architecture of Code injection buffer overflow attack blocker.

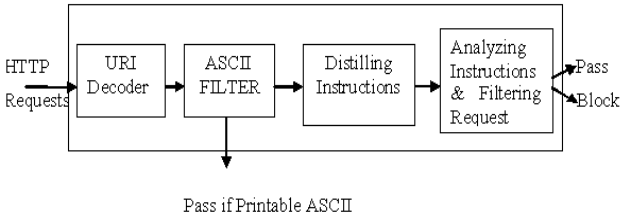

Fig.3. Working of Buffer overflow attack blocker

URI decoder: - The URLs specification restricts the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded. The first step of Code injection buffer overflow attack blocker is to decode the request-URI because a malicious payload maybe embedded in the request-URI as a request parameter.

ASCII Filter: - In order to provide throughput and response time of the protected web system, if the query parameters of the request-URI and request-body of a request are both printable ASCII ranging from 20-7E in hex then Code injection buffer overflow attack blocker allows the request to pass. A malicious executable code is normally binary string.

Distilling Instructions:- This module distills all possible instruction sequences from the query parameters of Request-URI and Request-Body. This section first proposes an executive algorithm to distill instruction sequences from http requests, followed by several excluding techniques to reduce the processing overhead of instruction sequences analyzer.

International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 4, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

Analyzing Instructions and Filtering Request: - Using all the instruction sequences distilled from the instruction sequences distiller as the inputs, this module analyzes these instruction sequences to determine whether one of them is part of a program.

IV. PROCESSING INSTRUCTION SEQUENCES

For distilling instruction sequences, first assign an address to every byte of a request. After assigning addresses we disassemble the request from certain address unless end of the request is reached or an illegal op-code is obtained. There are two traditional disassembly algorithms: linear sweep and recursive traversal. The linear sweep algorithm begins disassembly at a certain address, and proceeds by decoding each encountered instruction. The recursive traversal algorithm also begins disassembly at a certain address, but it follows the control flow of instructions and as per the requirements of our project we want control flow information of instructions. In this, we apply the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. In order to get all possible instruction sequences from N-byte request we execute disassembly algorithm N times, and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time complexity is also very less.

Definitions:

Definition1 (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.[2]

An instruction sequence may be a fragment of a program but an instruction sequence is not necessarily a fragment of a program. We can distill instruction sequences from any binary strings so it becomes challenge. Fig. 4 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as si in Fig. 4, where i is the entry location of the instruction sequence in the string. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. We can call these instruction sequences as random instruction sequences and fragment of a program as binary executable code.

Definition 2 (instruction flow graph): - An instruction flow graph (IFG) is a directed graph G= (V, E) where each node v ∈ V corresponds to an instruction and each edge e = (vi, vj) ∈ E corresponds to a possible transfer of control from instruction vi to instruction vj.

Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model the control flow of an instruction sequence, we further extend the above definition.

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

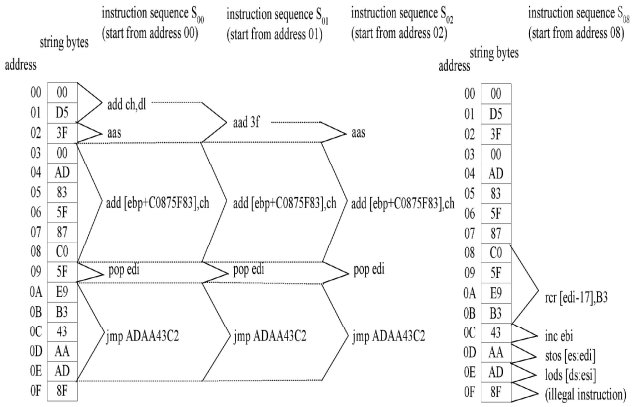

Fig.4. Assigning addresses to instruction sequences

As shown in above Figure4 Instruction sequences distilled from a substring of a GIF file. We assign an address to every byte of the string. Instruction sequences s00, s01, s02, and s08 are distilled by disassembling the string from addresses 00, 01, 02, and 08, respectively.

In recursive traversal algorithm the same instructions are decoded many times is the main drawback of the algorithm. For example, instruction “pop edi” in Figure. 4 is decoded many times by this algorithm. To reduce the running time, we use a data structure, which is an EIFG, to represent the instruction sequences. An EIFG is created to distill all possible instruction sequences from a request. We use an instruction array to represent all possible instructions in a request. An EIFG is used to represent all possible transfers of control among these instructions. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruction array.

Excluding Instruction sequences

Distilling instruction sequences generate many instruction sequences at different entry points. Hence we remove some of them based on several methods. Instruction sequences are excluded means that the entry of this sequence is not considered as the real entry for the embedded code. In removing instruction sequences the basic rule is that it should not affect the decision whether a request contains code or not. This rule can be defined into the technical requirements: if a request contains a fragment of a program, the program must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from the remaining sequence only by few instructions.

International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 4, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

Analyzing Instruction Sequences:

After distilling instruction sequence we get either sequence of random instructions or a fragment of a program in machine language. To analyze the instruction sequences whether it is fragment of the program or not different methods can be used. We used a method which exploits the data flow characteristics of a program to detect the obfuscated buffer overflow attacks. A real program has few or no data flow anomalies and a random instruction sequence is full of data flow anomalies. An attacker may obfuscate his program easily by introducing enough data flow anomalies so the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence. Here, we use the detection of data flow anomaly in a different way. 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 have 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.

A data flow anomaly is caused by an improper sequence of actions performed on a variable. There are three data flow anomalies:

a) define-define,
b) define-undefine,
c) undefine-reference

V. RESULTS

System first accepts HTTP Request for that user has to first upload file and send it to the server as a request. The uploaded file is stored in database. If request contains executable file then system does not allow sending request to the server and system block the request as shown in figure5.2. If request does not contain executable file then it is stored in database and given to the next stage that is URI decoder.

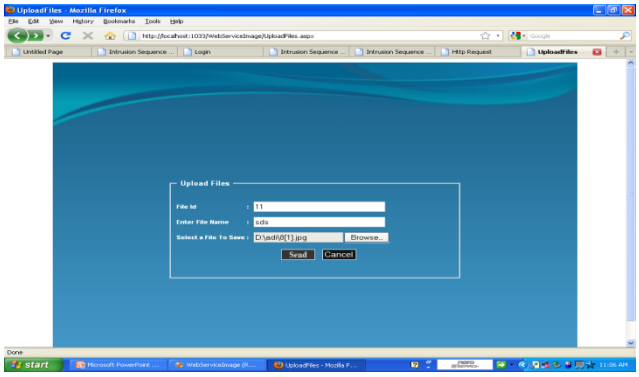

Fig.5.1. Snapshot for uploading file

In URI decoder system decodes the URI request because code may be embedded in request only. After checking decoded request is send to ASCII filter.

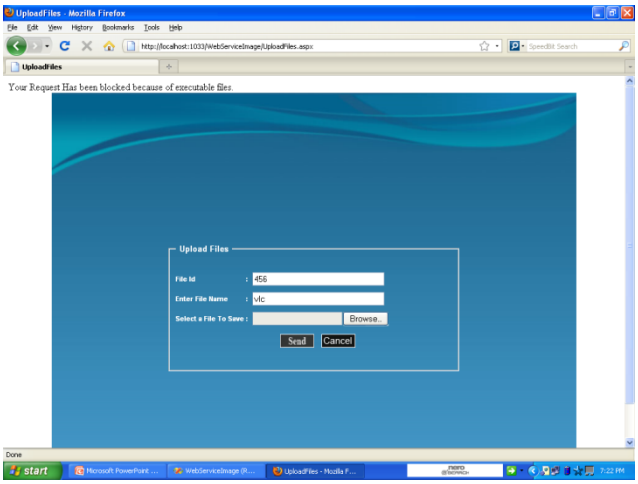

Fig.5.2. Snapshot for blocking request on executable files.

While analyzing instruction sequences we can use different methods to detect dependence degree of instructions. Normally, a random instruction sequence is full of data flow anomalies but a program has very less or no data flow anomalies. We use a threshold value of useful instructions, as a random instruction sequence has very less useful instructions then we can pass that instruction sequences and if it crosses threshold value then discards that request.

Another method based on a specific operating systems, a program in machine language has certain characteristics depending on operating system on which it is running, like calls to operating system libraries. A random instruction sequence does not carry this kind of instructions. We can easily differentiate a real program from a random instruction sequence by identifying call patterns in instruction sequence.

Second method is faster than data flow anomaly techniques but first method is more robust to obfuscation.

Limitation of this proposed system is that it cannot fully handle the self modifying code, which dynamically modifies itself at runtime. It also does not detect attacks such as return-to-libc attacks that just corrupt control flow without injecting code.

Whenever any suspicious packet comes to system and that is analyzed by analyzer then system will send message over mobile and email to the server administrator.

VI. CONCLUSION

A Code injection buffer overflow attack blocker is a web application that can filter code-injection buffer overflow attack messages, one of the worst cyber security threats. It works on the observation that buffer overflow attack usually contains executables in the messages where as legitimate client has data in the message. Code injection buffer overflow attack blocker blocks attacks by detecting the presence of code. This approach does not use any pre-known patter to detect the code so it can detect any unknown buffer overflow attack. For detection of code it uses a new data-flow analysis technique called code abstraction that is generic and fast. For deployment this method does not require any changes in software or

hardware of the server so it's economical, and can also handle encrypted SSL messages.

REFERENCES

[1] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan, Nanyang Technological University, Singapore," Defending against Buffer Overflow Vulnerabilities", published by IEEE computer society, 018-9162, November 2011.

[2] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu," SigFree: A Signature-Free Buffer Overflow Attack Blocker", in ieee transaction on Dependable and secure computing, vol. 7, no. 1, Jan-March 2010.

[3] Hossain Shahriar and Mohammad Zulkernine,S chool of Computing Queen's University, Kingston, Canada," Classification of Buffer Overflow Vulnerability Monitors", International Conference on Availability, Reliability and Security, 2010.

[4] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, "Detecting and prevention of stack buffer overflow attacks," Communications of the ACM, vol. 48, no.11 pp.51-56, 2005.

[5] J.Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in NDSS, 2005.

[6] Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in Proc. 12th ACM Conference on Computer and Communications Security, 2005.

[7] A. Smirnov and T. cker Chiueh, "Dira: Automatic Detection,Identification, and Repair of Control-Hijacking Attacks," Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[8] R. Chinchani and E. V. D. Berg, "A fast static analysis approach to detect exploit code inside network flows," in RAID, 2005.

[9] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in RAID, 2005

[10] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in Proc. 12th ACM Conference on Computer and Communications Security, 2005.

[11] S. Singh, C. Estan, G. Varghese, and S. Savage, "The early bird system for real-time detection of unknown worms," tech. rep., University of California at San Diego, 2003.

[12] H.A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in Proceedings of the 13th Usenix Security Symposium, August 2004.

[13] G. Kc, A. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in Proceedings of the 10th ACM conference on Computer and communications security, pp. 272-280, October 2003.

[14] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in Proceedings of the 10th ACM conference on Computer and communications security, 2003.

[15] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," Proc. Seventh USENIX Security Symp. (Security '98), Jan. 1998.

[16] GCC Extension for Protecting Applications from Stack-SmashingAttacks, http://www.research.ibm.com/trl/projects/security/ssp, 2007.

[17] T. cker Chiueh and F. H. Hsu, "Rad: A Compile-Time Solution toBuffer Overflow Attacks," Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS), 2001.

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

[19] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense against Stack Smashing Attacks," Proc. USENIX Ann. Technical Conf. (USENIX '00), June 2000.

[20] G.S. Kc and A.D. Keromytis, "E-NEXSH: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," Proc. 21st Ann. Computer Security Applications Conf. (ACSAC), 2005.

AUTHOR'S PROFILE

Ms. Mayura A. Kathwate

received the B.E. degree form Sanjeevani Rural Education Society College of Engineering from Pune University in 2007, worked as a lecturer for two years in K J Somaiya Institute of Information Technology, Sion. Currently, M.E. (Computer) Student in Terna Engineering College, Nerul, Navi Mumbai from Mumbai university, working in Datta Meghe College of Engineering, Airoli, Navi Mumbai.

Mr. D. K. Chitre

is an Associate Professor of Computer Engineering in Terna Engineering College, Nerul, Navi Mumbai, currently working as a Head of Department of Computer Engineering in same institute from Mumbai University.