ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

Prevention of Buffer flow Vulnerability in Cyber Security using Stamp Free Analysis Techniques

G. Michael[1], G.Kavitha[2]

Assistant Professor, Department of Computer Science Engineering, Bharath University, Chennai, Tamil Nadu, India[1]

Assistant Professor, Department of Computer Science Engineering, Bharath University, Chennai, Tamil Nadu, India[2]

ABSTRACT*:* Stamp-free out-of-the-box has an application-layer for jamming the buffer overflow attack messages targeting at various Internet services. Focus on buffer overflow attacks whose payloads contain executable code in machine language. Unlike the previous code detection algorithms, StpFree uses a new data-flow analysis technique called code abstraction that is generic, fast, and hard for exploit code to evade. StpFree is stamp free, thus it can block new and unknown buffer overflow attacks; StpFree is also immunized from most attack-side code obfuscation methods. Since StpFree is a transparent deployment to the servers being protected, it is good for economical Internet-wide deployment with very low deployment and maintenance cost. StpFree could block all types of code-injection attack packets.

KEYWORDS: Stamp-Free (StpFree) malicious detection, buffer overflow attacks, code-injection attacks.

I. INTRODUCTION

The cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking in, worms, zombies, and botnets. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and depending on what is stored there, the behavior of the program itself might be affected. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests code-injection buffer overflow attacks such as stack smashing probably count for most of the buffer overflow attacks that have happened in the real world. StpFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at StpFree, StpFree first uses a new algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded However, in this phase, some data bytes may be mistakenly decoded as instructions. StpFree uses a novel technique called code abstraction. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions or dependence degree to a threshold to determine if this instruction sequence contains code. Unlike the existing code detection algorithms that are based on stamps, rules, or control flow detection, StpFree is generic and hard for exploit code to evade.

II. RELATED WORK

Various bug-finding tools [1], [2], [3] have been developed. 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. Techniques are designed to handle source code only, and they do not ensure completeness in bug finding. DIRA [4] is another compiler that can detect control hijacking attacks, identify StpFree is an application layer blocker between the protected server and the corresponding firewall. The malicious input, and repair the compromised program. Techniques require the availability of source code. In contrast, StpFree does not need to know any source code. Address-space

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

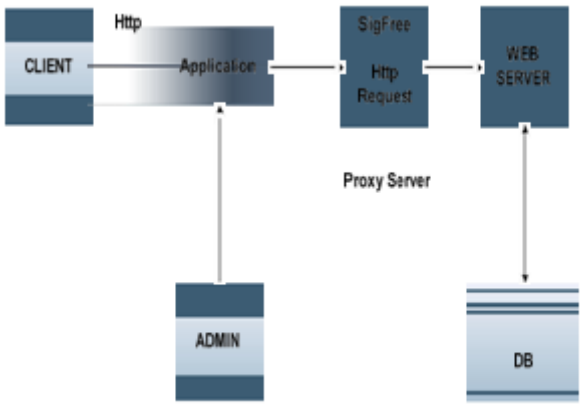
(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

randomization, in its general form [5], can detect exploitation of all memory errors. Instruction set randomization [6], [7] can detect all code-injection attacks, whereas StpFree cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable". In contrast, StpFree is an online attack blocker. As such, their techniques and StpFree are complementary to each other with different purposes. Moreover, unlike StpFree, their techniques[8] may not be suitable to block the code contained in every attack packet, because some buffer overflow code is so simple that very little control flow information can be exploited. Independent of our work, Chinchani and Berg [9] proposed a rule-based scheme to achieve the same goal as that of StpFree, that is, to detect exploit code in network flows. However, there is a fundamental difference between StpFree and their scheme. Their scheme is rule-based, whereas StpFree is a generic approach that does not require any preknown patterns. More specifically, their scheme first tries to find certain preknown instructions, instruction patterns, or control flow structures in a packet. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to check if the packet really contains code.

III. THE PROPOSED STAMP FREE BUFFER OVERFLOW ATTACK BLOCKER

This section first describes an effective algorithm to avoid buffer overflow attack instruction sequences from requests, followed by several pruning techniques to reduce the processing overhead of instruction sequence analyzer. There are two traditional disassembly algorithms: linear sweep and recursive traversal 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.



One drawback of the algorithm is that the same instructions are decoded many times. To reduce the running time, we design a memorization algorithm by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request. An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. 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.

Algorithm

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

Stamp-Free Buffer Overflow

Attack Blocker:

initialize EISG G and instruction array A to empty
for each address i of the request do add instruction node i to G
i ←the start address of the request
while i ← the end address of the request do inst ← decode an instruction at i
if inst is illegal then
A[i]← illegal instruction inst
set type of node i "illegal node" in G else
A[i]←instruction inst
if inst is a control transfer instruction then for each possible target t of inst do
if target t is an external address then add external address node t to G add edge e(node i; node t) to G else

add edge e(node i; node i + inst.length) to G i← i+1

II. INSTRUCTION SEQUENCES ANALYZER

A buffer overflow attack may be a sequence of random instructions or a fragment of a program in machine language. A program in machine language is dedicated to a specific operating system; hence, a program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. By identifying the call pattern in an instruction sequence, we can effectively differentiate a real program from a random instruction sequence. Where as a real program has few or no data flow anomalies[1]. 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. The detection of data flow anomaly in a different way called code abstraction[2]. 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[3].

IV. DATA FLOW ANOMALY

The term data flow anomaly was originally used to analyze programs written in higher level languages in the software reliability and testing field. During a program execution, an instruction may impact a variable (register, memory location, or stack) on three different ways: define, reference, and undefine. A variable is defined when it is set a value; it is referenced when its value is referred to; it is undefined when its value is not set or set by another undefined variable. Note that here the definition of undefined is different from that in a high level language. For example, in a C program, a local variable of a block becomes undefined when control leaves the block[4]. A data flow anomaly is caused by an improper sequence of actions performed on a variable[5]. There are three data flow anomalies: define-define, define-undefine, and undefinereference. The define-define anomaly means that a variable was defined and is defined again, but it has never been referenced between these two actions. The undefinereference anomaly indicates that a variable that was undefined receives a reference action[6]. The define-undefine anomaly means that a variable was defined, and before it is used it is undefined[7].

V. DETECTION OF DATA FLOW ANOMALIES

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 either due to the need for real execution of a program with some

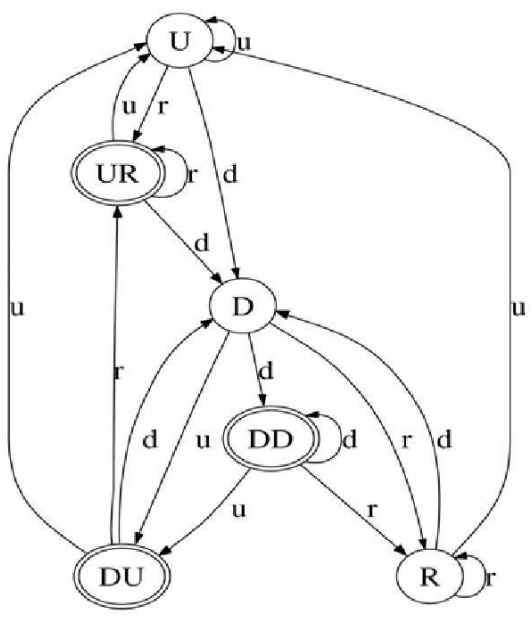
ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

inputs. As such, we propose a new method called code abstraction, which does not require real execution of code. As a result of the code abstraction of an instruction, a variable could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD: abnormal state definedefine; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine.



State U: undefined, state D: defined but not referenced, state R: defined and referenced, state DD: abnormal state define-define, state
UR: abnormal state undefine-reference, and state DU: abnormal state define-undefine[8].

Pruning useless instructions

Next, we leverage the detected data flow anomalies to remove useless instructions. A useless instruction of an execution path is an instruction that does not affect the results of the execution path; otherwise, it is called useful instructions. We may find a useless instruction from a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction that causes the "reference" is a useless instruction[9].

General purpose instruction

The instructions in the IA-32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instructions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we consider other groups of instructions as useless instructions[10].

Initial state of registers

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

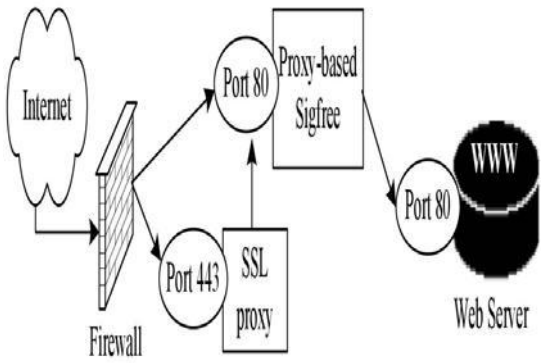
For registers, we set their initial states to "undefined" at the beginning of an execution path. The register "esp," however, is an exception since it is used to hold the stack pointer. Thus, we set register esp "defined" at the beginning of an execution path[11].

Indirect address

An indirect address is an address that serves as a reference point instead of an address to the direct memory location[12].

Application-specific encryption handling

The proxy-based StpFree could not handle encrypted or encoded data directly. A particular example is SSL-enabled web server. Enhancing security between web clients and web servers by encrypting HTTP messages, SSL also causes the difficulty for out-of-box malicious code detectors. To support SSL functionality, an SSL proxy such as S tunnel may be deployed to securely tunnel the traffic between clients and web servers[13].



In this case, we may simply install StpFree in the machine where the SSL proxy is located. It handles the web requests in clear text that have been decrypted by the SSL proxy[14. On the other hand, in some web server applications, SSL is implemented as a server module. In this case, StpFree will need to be implemented as a server module located between the SSL module and WWW server. We notice that most popular web servers allow us to write a server module to process requests and specify the order of server modules. Detailed study will be reported in our future work[15].

IV. CONCLUSION

A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. Many businesses do not view these changes and the process termination overhead as economical deployment. To overcome the above limitations, The StpFree is stamp free, thus it can block new and unknown buffer overflow attacks. Without relying on string-matching, StpFree is immunized from most attack-side obfuscation methods. StpFree uses generic code-data separation criteria instead of limited rules.

REFERENCES

1. 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.
2. Udayakumar R., Khanaa V., Kaliyamurthie K.P., "High data rate for coherent optical wired communication using DSP", Indian Journal of Science and Technology, ISSN : 0974-6846, 6(S6) (2013) 4772-4776.
3. D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," IEEE Software, vol. 19, no. 1, 2002.
4. Jaikumar S., Ramaswamy S., Asokan B.R., Mohan T., Gnanavel M., "Anti ulcer activity of methanolic extract of Jatropha curcas (Linn.) on Aspirin-induced gastric lesions in wistar strain rats", Research Journal of Pharmaceutical, Biological and Chemical Sciences, ISSN : 0975-

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 10, October 2014

8585, 1(4) (2010) PP.886-897.

5. 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.
6. Udayakumar R., Khanaa V., Kaliyamurthie K.P., "Optical ring architecture performance evaluation using ordinary receiver", Indian Journal of Science and Technology, ISSN : 0974-6846, 6(S6) (2013) pp. 4742-4747.
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. Kumar S.S., Rao M.R.K., Balasubramanian M.P., "Anticarcinogenic effects of indigofera aspalathoides on 20-methylcholanthrene induced fibrosarcoma in rats", Research Journal of Medicinal Plant, ISSN : 5(6) (2011) PP. 747-755.
9. S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," Proc. 14th USENIX Security Symp. (Security), 2005.
10. Udayakumar R., Khanaa V., Kaliyamurthie K.P., "Performance analysis of resilient ftth architecture with protection mechanism", Indian Journal of Science and Technology, ISSN : 0974-6846, 6(S6) (2013) pp. 4737-4741
11. 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.
12. 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.
13. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," Proc. Eighth Int'l Symp. Recent Advances in malicious Detection (RAID), 2005.
14. R. Chinchani and E.V.D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows," Proc. Eighth Int'l Symp. Recent Advances in malicious Detection (RAID), 2005.
15. A. Lakhotia and U. Eric, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries," Proc. Fourth IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM '04), Sept. 2004.
16. B.Vamsi Krishna, Significance of TSC on Reactive power Compensation, International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, ISSN (Online): 2278 – 8875,pp 7067-7078, Vol. 3, Issue 2, Febuary 2014
17. B.Vamsi Krishna, Realization of AC-AC Converter Using Matrix Converter, International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, ISSN (Online): 2278 – 8875,pp 6505-6512, Vol. 3, Issue 1, January 2014
18. D.Sridhar raja, Comparison of UWB Band pass filter and EBG embedded UWB Band pass filter, International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, ISSN 2278 – 8875,pp 253-257 ,Vol. 1, Issue 4, October 2012
19. D.Sridhar raja, Performances of Asymmetric Electromagnetic Band Gap Structure in UWB Band pass notch filter, International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, ISSN (Online): 2278 – 8875,pp 5492-5496, Vol. 2, Issue 11, November 2013
20. Dr.S.Senthil kumar, Geothermal Power Plant Design using PLC and SCADA, International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, ISSN 2278 – 8875,pp 30-34, Vol. 1, Issue 1, July 2012