ISSN: 0976-3031

Available Online at http://www.recentscientific.com

International Journal of Recent Scientific Research
Vol. 3, Issue, 7, pp. 644 - 646, July, 2012

International Journal of Recent Scientific Research

REVIEW ARTICLE

SIGFREE-EFFICIENT BUFFER OVERFLOW ATTACK PROTECTOR

***Vetrivendan, R., Renugadevi, N and Vignesh, B**

Department of CSE, As-salam college of Engineering and Technology-Tamilnadu

ARTICLE INFO

Article History:

Received 11th June, 2012
Received in revised form 20 th , June, 2012
Accepted 10th July, 2012
Published online 30th July, 2012

Key words:

Intrusion detection, buffer overflow attacks, code-injection attacks

ABSTRACT

Efficient 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. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. Unlike the previous code detection algorithms, SigFree uses a new data-flow analysis technique called code abstraction that is generic, fast, and hard for exploit code to evade. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is a transparent deployment to the servers being protected, it is good for economical Internet-wide deployment with very low deployment and maintenance cost. We implemented and tested SigFree; our experimental study shows that the dependency-degree-based SigFree could block all types of code-injection attack packets (above 750) tested in our experiments with very few false positives. Moreover, SigFree causes very small extra latency to normal client requests when some requests contain exploit code.

© Copy Right, IJRSR, 2012, Academic Journals. All rights reserved.

INTRODUCTION

The history of 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 bonnets. 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 [1]. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets), 1 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. Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly desired requirements: (R1) simplicity in maintenance; (R2) transparency to existing (legacy) server OS, application software, and hardware; (R3) resiliency to obfuscation; (R4) economical Internet-wide deployment. 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. To see how existing defenses are limited in meeting these four requirements, let us break down the existing buffer overflow defenses into six classes, which we will review shortly in Section 2: (1A) Finding bugs in source code. (1B) Compiler extensions. (1C) OS modifications. (1D) Hardware modifications. (1E) Defense-side obfuscation [3], [4]. (1F) Capturing code running symptoms of buffer overflow attacks [5], [6], [7], [8]. (Note that the above list does not include binary-code-analysis-based defenses, which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows: 1) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardware, thus they are not transparent. Moreover, Class 1E defenses 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 1F defenses 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 1F defenses have limited transparency and potential for economical deployment. 3) Class 1A defenses need source code, but source code is unavailable to many legacy applications. Besides buffer overflow defenses, 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 relies on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

MATERIALS AND METHODS

Experiments

To overcome the above limitations, in this paper, we propose SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that “the nature of communication to

* Corresponding author: + 91
E-mail address: vetriascet@gmail.com

and from network services is predominantly or exclusively data and not executable code" [12]. In particular, as summarized in [12], 1) on Windows platforms, most web servers (port 80) accept data only; remote access services

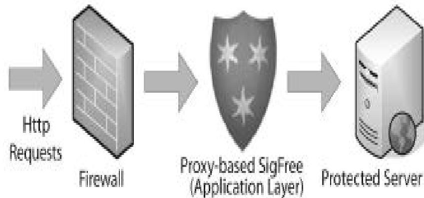

Fig. 1 Sig Free is an application layer blocker between the protected server and the corresponding firewall

(ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434), which are used to monitor Microsoft SQL Databases, accept data only. 2) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306, and 5432 accept data only. Since remote exploits are typically binary executable code, this observation indicates that if we can precisely distinguish (service requesting) messages containing binary code from those containing no binary code, we can protect most Internet services (which accept data only) from code injection buffer overflow attacks by blocking the messages that contain binary code. Accordingly, SigFree (Fig. 1) works as follows: SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new OðNÞ 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 (if any). However, in this phase, some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree 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 (Scheme 2) or dependence degree (Scheme 3) to a threshold to determine if this instruction sequence (distilled in phase 1) contains code. Unlike the existing code detection algorithms [12], [13], [14] that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade (Section 2 gives a more detailed comparison).

The merits of SigFree are summarized as follows: they show that SigFree has taken a main step forward in meeting the four requirements aforementioned:

- SigFree is signature free, thus it can block new and unknown buffer overflow attacks.
- Without relying on string matching, SigFree is immunized from most attack-side obfuscation methods.
- SigFree uses generic code-data separation criteria instead of limited rules. This feature separates SigFree from [12], an independent work that tries to detect code embedded packets.
- Transparency. SigFree is an out-of-the-box solution that requires no server side changes.
- SigFree is an economical deployment with very low maintenance cost, which can be well justified by the aforementioned features.

Sigfree Overview

Basic Definitions and Notations

This section provides the definitions that will be used in the rest of this paper. 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. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Fig. 2 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as si in Fig. 2, where 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. Below, 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. Definition 2 (instruction flow graph). An instruction flow graph (IFG) is a directed graph G ¼ ðV; EÞ where each node v 2 V corresponds to an instruction and each edge e ¼ ðvi; vjÞ 2 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.

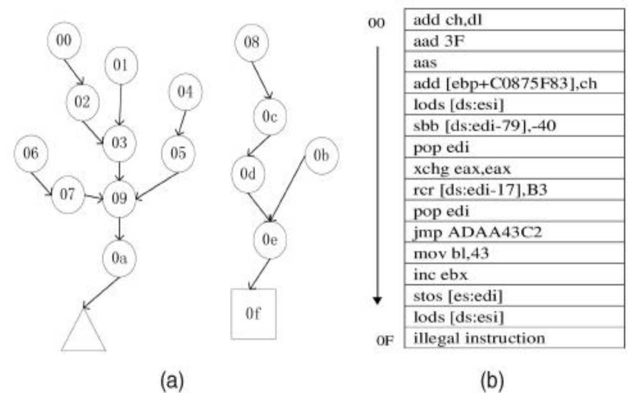

Fig. 3. Data structure for the instruction sequences distilled from the request in Fig. 2. (a) EIFG. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

Definition 3 (extended IFG). An extended IFG (EIFG) is a directed graph G ¼ ðV; EÞ, which satisfies the following properties: each node v 2 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 ¼

ðvi; vjÞ 2 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. The reason that we define IFG and EIFG is to model two special cases, which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s08 in Fig. 2, the transfer of control is from instruction "lods [ds:esi]" to an illegal instruction at address 0F. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s00 in Fig. 2 has an instruction "jmp ADAAC3C2," which jumps to external address ADAAC3C2.

Algorithm 1

Distill all instruction sequences from a request 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 <¼ the end address of the request do inst decode an instruction at I if inst is illegal then A½i_ illegal instruction inst set type of node i "illegal node" in G else A½i_ 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 eðnode i; node tÞ to G else add edge eðnode i; node i þ inst:lengthÞ to G

CONCLUSION

We have proposed SigFree, an online signature-free out-of - the- box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. SigFree does not require any signatures, thus it can block new unknown attacks. SigFree is immunized from most attack-side code obfuscation methods and good for economical Internet-wide deployment with little maintenance cost and low performance overhead

References

1. B.A. Kuperman, C.E. Brodley, H. Ozdoganoglu, T.N. Vijaykumar, and A. Jalote, "Detecting and Prevention of Stack Buffer OverflowAttacks," Comm. ACM, vol. 48, no. 11, 2005.
2. J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," IEEE Security and Privacy, vol. 2, no. 4, 2004.
3. G. Kc, A. Keromytis, and V. Prevelakis, "Countering Code- Injection Attacks with Instruction-Set Randomization," Proc. 10thACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.
4. 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.
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
7. Worms," Proc. 20thACMSymp. Operating Systems Principles (SOSP), 2005.
8. 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.
9. 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.
10. S. Singh, C. Estan, G. Varghese, and S. Savage, "The Earlybird System for Real-Time Detection of Unknown Worms," technical report, Univ. of California, San Diego, 2003.
11. H.-A. Kim and B. Karp, "Autograph: Toward Automated,
12. Distributed Worm Signature Detection," Proc. 13th USENIX Security Symp. (Security), 2004.
13. J. Newsome, B. Karp, and D. Song, "Polygraph: Automatic Signature Generation for Polymorphic Worms," Proc. IEEE Symp. Security and Privacy (S&P), 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 Intrusion Detection (RAID), 2005.
15. T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," Proc. Fifth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2002.
16. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna,
17. "Polymorphic Worm Detection Using Structural Information of Executables," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2005.

****