Blocker: A Blocker Of Signature Free Buffer Overflow Attack

L.Raghavendar Raju
Associate Professor, Dept of CSE
JPNCE – Mahabubnagar, A. P.

Prof. D. Jamuna
Professor & HOD, Dept of CSE
JPNCE – Mahabubnagar, A. P.

M.Janardhan Reddy
M Tech. Research Scholar,
JPNCE Mahabubnagar,A.P

ABSTRACT

We propose Sig Free, 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, Sig Free blocks attacks by detecting the presence of code. Unlike the previous code detection algorithms, Sig Free uses a new data-flow analysis technique called code abstraction that is generic, fast, and hard for exploit code to evade. Sig Free is signature free, thus it can block new and unknown buffer overflow attacks; Sig Free is also immunized from most attack-side code obfuscation methods. Since Sig Free 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 Sig Free; our experimental study Shows that the dependency-degree-based Sig Free could block all types of code-injection attack packets (above 750) tested in our experiments with very few false positives. Moreover, Sig Free causes very small extra latency to normal client requests when some requests contain exploite code.

Index Terms—Intrusion detection, buffer overflow attacks, code-injection attacks.

1. INTRODUCTION

Throughout 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 bot nets. 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 (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.(1F) Capturing code running symptoms of buffer overflow attacks . (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,and1E 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) Class1F 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. 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.

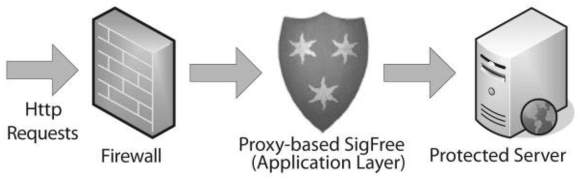

Fig.1. Sig Free is an application layer blocker between the protected server and the corresponding firewall.

To overcome the above limitations, in this paper, we propose Sig Free, an online buffer overflow attack blocker, to protect Internet services. The idea of Sig Free is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code" . In particular, as summarized in 1)on Windows platforms, most web servers (port 80) accept data only; remote access services (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, Postgre SQL)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, Sig Free (Fig. 1) works as follows: Sig Free is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at Sig Free, Sig Free first uses anew O(N)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,Sig Free 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 that are based on signatures, rules, or control flow detection, Sig Free is generic and hard for exploit code to evade.

2. RELATED WORK

2.1 Prevention/Detection of Buffer Overflows

Existing prevention/detection techniques of buffer overflows can be roughly broken down into six classes:

Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools 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. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, Sig Free handles machine code embedded in a request (message).

Class 1B: 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]. Three such compilers are Stack Guard [22], Pro Police [23], and Return Address Defender (RAD) [24]. DIRA [25] is another compiler that can detect control hijacking attacks, identify the malicious input, and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, Sig Free does not need to know any source code.

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax, LibSafe , and e-NeXsh . Class 1Ctechniques need to modify the OS. In contrast, Sig Free does not need any modification of the OS.

Class 1D: 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.

Class 1E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX

[26].Address-space randomization, in its general form [30], can detect exploitation of all memory errors. Instruction set randomization [3], [4] can detect all code-injection attacks, whereas Sig Free 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”.

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows are a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class 1B, Class 1C, and Class 1E 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.

2.2)Worm Detection and Signature Generation

Because buffer overflow is a key target of worms when they propagate from one host to another, Sig Free is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [Class 2A] techniques use such macro symptoms as Internet background radiation (observed by network telescopes) to raise early warnings of Internet-wide worm infection [33]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence, and address dispersion to generate worm signatures and/or block worms.

2.3) Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), 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 .

2.4) URI Decoder.

The specification for URLs [12] limits 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 [12]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of Sig Free is to decode the request-URI.

2.5) ASCII Filter.

Malicious executable code are normally binary strings. In order to guarantee the 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, Sig Free allows the request to pass (In Section 7.2,we will discuss a special type of executable codes called alphanumeric shell codes [45] that actually use printable ASCII) .

2.6) Instruction sequences distiller (ISD*)*
This module distills all possible instruction sequences from the query parameters of Request-URI and Request-Body (if the request as one).

2.7) Instruction sequences analyzer (ISA).
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 a program.

3. DISCUSSION

3.1 Robustness to Obfuscation

Most malware detection schemes include a two-stage analysis. The first stage is disassembling binary code, and the second stage is analyzing the disassembly results. There are obfuscation techniques to attack each stage and attackers may use them to evade detection. Sig Free is robust to most of these obfuscation techniques. Obfuscation in the first stage. Junk byte insertion is one of the simplest obfuscation against disassembly. Here, junk bytes are inserted at locations that are not reachable at runtime. This insertion however can mislead a linear sweep algorithm but cannot mislead a recursive traversal algorithm, on which our algorithm bases.

3.2 Limitations

Sig Free also has several limitations. First, Sig Free cannot fully handle the branch-function-based obfuscation, as indicated in Table 1. Branch function is a function f(x) that, whenever called from x, causes control to be transferred to the corresponding location f(x). By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art. Second, Sig Free cannot fully handle self-modifying code. Self-modifying code is a piece of code that dynamically modifies itself at runtime and could make Sig Free mistakenly exclude all its instruction sequences. Third, the executable shell codes could be written in alphanumeric form . Such shell codes will be treated as printable ASCII data and thus bypass our analyzer. By turning off the ASCII filter, Scheme 2 and Scheme 3 can successfully detect alphanumeric shell codes; however, it will increase computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

Fourth, Sig Free does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code. However, these attacks can be handled by some simple methods. For example, return-to-libc attacks can be defeated by mapping the addresses of shared libraries so that the addresses contain null bytes.

3.3 Application-Specific Encryption Handling

The proxy-based Sig Free 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 Stunnel may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install Sig Free 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. On the other hand, in some web server applications, SSL is implemented as a server module In this case, Sig Free will need to be implemented as a server module(though not shown in Fig. 13), located between the SSL module and the 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.

3.4 Applicability

So far, we only discussed using Sig Free to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to bufferOver flow attacks. For example, the proxy-based Sig Free can be used to protect all internet services that do not permit executable binaries to be carried in requests. Sig Free should not directly be used to protect some Internet services that do accept binary code such as FTP servers; otherwise, Sig Free will generate many false positives. To apply Sig Free for protecting these Internet services, other mechanisms such as white listing need to be used.

In addition to protecting servers, Sig Free can also provide file system real-time protection. Buffer overflow vulnerabilities Have been found in some famous applications such as Adobe Acrobat and Adobe Reader , Microsoft JPEGProcessing , and Win Amp.

4. CONCLUSION

We have proposed Sig Free, 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. Sig Free does not require any signatures, thus it can block new unknown attacks. Sig Free is immunized from most attack-side code obfuscation methods and good for economical Internet-wide deployment with little maintenance cost and low performance overhead.

5. ACKNOWLEDGEMENT

The authors express their deep gratitude to the Principal and the Management members of JPNCE for their encouragement and extensive support in preparing and publishing of this paper.

6. REFERENCES

[1] 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.

[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. 10th ACM 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 Worms," Proc. 20thACMSymp. Operating Systems Principles (SOSP),2005.

[7] Z. Liang and R. Shekar, "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,"AutomaticDiagnosisandResponse toMemory Corruption Vulnerabilities,"Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[9] S. Singh, C. Estan, G. Varghese, and S. Savage, "The EarlybirdSystem for Real-Time Detection of Unknown Worms," technical report, Univ. of California, San Diego, 2003.

[10] H.-A. Kim and B. Karp, "Autograph: Toward Automated,Distributed Worm Signature Detection," Proc. 13th USENIX Security Symp. (Security), 2004.

[11] J. Newsome, B. Karp, and D. Song, "Polygraph: AutomaticSignature Generation for Polymorphic Worms," Proc. IEEE Symp.Security and Privacy (S&P), 2005.

[12] R. Chinchani and E.V.D. Berg, "A Fast Static Analysis Approachto Detect Exploit Code inside Network Flows," Proc. Eighth Int'lSymp. Recent Advances in Intrusion Detection (RAID), 2005.

[13] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection viaAbstract Payload Execution," Proc. Fifth Int'l Symp. RecentAdvances in Intrusion Detection (RAID), 2002.

[14] 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 Intrusion Detection (RAID), 2005.

[15] The Meta sploit Project, http://www.metasploit.com, 2007.

[16]JempiscodesAPolymorphicShellcodeGenerator,http://www.shellcode.com.ar/en/proyectos.html, 2007.

[17] S. Macaulay, Admmutate: Polymorphic ShellcodeEngine,http://www.ktwo.ca /security.html, 2007.

[18]T.Detristan, T. Ulenspiegel, Y. Malcom, and M.S.V. Underduk, Polymorphic Shell codeEngineUsingSpectrumAnalysis,http://www.phrack.org/show.php?p=61&a=9, 2007.

[19] 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.

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

[21] 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.

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

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

Mr.L. Raghavendar Raju, Working as Associate Professor in CSE Dept. Jaya prakash Narayan College of Engineering, Mahabubnagar. His areas of Interest are in Mobile Adhoc Networks, Data Mining, Networking and guided M. Tech and B. Tech Students IEEE Projects.

Prof.D.Jamuna, Working as Professor & Head of CSE Dept. Jayaprakash Narayan College of Engineering, Mahabubnagar, M.Tech(SE) from School of Information Technology, JNTUH, Hyderabad. BE(CSE) from Vijayanagara Engineering College, Bellary. Experience 15 Years in Teaching Profession. Her areas of Interest are in Wireless Sensor Networks, Data Mining, Networking and guided M. Tech and B. Tech Students IEEE Projects. She is a Member of CSI.

M.JanardhanReddy,IIYear-M. Tech(CSE) Research Scholar at CSE Dept. Jayaprakash NarayanCollegeofEngineering, Mahabubnaar His areas of Interest are in Networking, Web Technologies, and Distributed System.