Protecting Web Services from Remote Exploit Code: A Static Analysis Approach

Xinran Wang, Yoon-Chan Jhi, Sencun Zhu
Dept. of Computer Science and Engineering, Pennsylvania State University, State College, PA
xinrwang, jhi, szhu@cse.psu.edu

Peng Liu
College of Information Sciences and Technology, Pennsylvania State University, State College, PA
pliu@ist.psu.edu

ABSTRACT

We propose *STILL*, a signature-free remote exploit binary code injection attack blocker to protect web servers and web applications. STILL is robust to almost all anti-signature, anti-static-analysis and anti-emulation obfuscation.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General - Security and protection.

General Terms: Security.

Keywords: HTTP, Code Injection Attack, Static Analysis.

1. INTRODUCTION

A great number of remote binary execution vulnerabilities including buffer overflow and format string vulnerabilities have been found in web servers and web applications [1]. This type of vulnerabilities allow attackers to use a crafted HTTP request to inject a piece of exploit binary code into the "body" of the web servers and applications. Once such exploit binary code injection attacks succeed, the attacker may gain full control of the victim machine. In different attacks, exploit code may be either a piece of shellcode to break into web servers or an infection vector for worms.

We propose *STILL*, a real-time, out-of-the-box, signature-free, remote exploit binary code injection attack blocker to protect web servers. STILL is motivated by an important observation that the request messages to web servers are exclusively data and not binary executable code. Since remote exploits are typically binary executable code, this observation indicates that if we can precisely distinguish (service requesting) messages that contain binary code from those that do not contain any binary code, we can protect web servers as well as other Internet services (which accept data only) from binary code-injection attacks by blocking the messages that contain binary code. Figure 1 shows that an application layer proxy-based STILL is deployed between the web server and the corresponding firewall to protect web servers.

STILL (including *static taint analysis* and *initialization analysis*) detect not only unobfuscated exploit code, traditional polymorphic and metamorphic exploit code, but also self-modifying and indirect jump obfuscation code that could easily defeat previous static analysis approaches. Indeed, STILL is robust to almost all anti-signature, anti-static-analysis and anti-emulation obfuscation. STILL is signature free, thus it can block new and unknown remote code injection attacks such as zero-day exploit code. STILL is also good for economical Internet wide deployment with very low deployment cost.

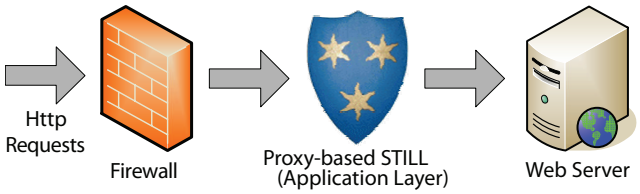

Figure 1: Deployment of STILL.

2. RELATED WORK

This paper is mainly relevant to the previous static analysis exploit code detection approaches [3, 4, 6]. One benefit of these static analysis approaches is that they can detect both foreseen exploit code exploiting known vulnerabilities and zero-day exploit code exploiting unknown vulnerabilities. In addition, they are in general more resilient to polymorphism and metamorphism (than string-matching signatures). However, Polychronakis et al. [5] demonstrated that some anti-static-analysis techniques such as self-modifying can easily thwart these existing static analysis techniques.

Polychronakis et al. [5] firstly proposed a CPU emulator to detect polymorphic shellcode. The emulators, being a dynamic analyzer, are immune to most anti-static-analysis techniques. However, dynamic analysis is vulnerable to several anti-emulation techniques, which have existed in virus writer community for many years. Motivated by [5], we proposed STILL, which is robust to both anti-static-analysis and anti-emulation techniques.

3. PROPOSED METHOD

Figure 2 depicts how STILL works. We next briefly describe its working flow. It works as a proxy-based blocker in the application layer. When it captures a data stream, it disassembles the data stream and generates a control flow graph. It analyzes the disassembled result in two stages. First, STILL detects self-modifying and indirect jump obfuscation code. Although the real exploit code may be hidden by self-modifying and indirect jump, the obfuscation code itself provides some strong evidences of self-modifying and/or

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

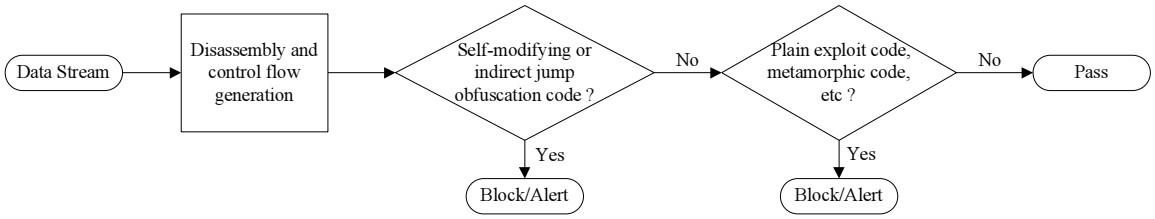

Figure 2: The activity diagram of STILL system

indirect jump behaviors. STILL detects these behaviors by static taint analysis and initialization analysis. Since polymorphism is a kind of self-modifying, STILL can also detect polymorphic code in this stage. However, attackers may use neither self-modifying nor indirect jump obfuscation. In the second stage, STILL detects the plain exploit code based on system calls and/or function calls that could even have been obfuscated by metamorphism. STILL also exploits static analysis and initialization analysis in this stage to combat other obfuscation techniques. Below we will describe the mechanisms in greater details.

3.1 Disassembly and Control Flow Graph Generation

We exploit the O(N) disassembly algorithm used in SigFree [6] to disassemble the input data stream and generate a control flow graph. Here N is the length of the data stream. It first decodes all possible instructions and finds all possible transfer of control in a data stream, and then creates a control flow graph based on these instructions and transfers of control. We note that in the presence of indirect jump and self-modifying obfuscation, it is impossible to completely and statically disassemble the entire body of the exploit code embedded in a data stream using the recursive traversal algorithm. Fortunately, the partially disassembled result may already provide some strong evidences of self-modifying and/or indirect jump behavior.

3.2 Detection of Self-modifying and Indirect Jump Obfuscation Code

The new techniques we propose to detect self-modifying and indirect jump exploit code are called *static taint analysis* and *initialization analysis*. We observe that self-modifying and indirect jump exploit code first need acquire the absolute address of payload. Accordingly, we first try to find the piece of code which acquires the absolute address of payload at runtime from an instruction sequence. The variable which holds the absolute address will be marked *tainted*.

Then, we use the static taint analysis approach to track the tainted values and detect whether tainted data are used in the ways that could indicate the presence of self-modifying and indirect jump exploit code. A tainted variable is propagated to a new tainted variable by data transfer instructions that move data (e.g., push, pop, move) and data operation instructions that perform arithmetic or bit-logic operations on data (e.g., add, sub, xor). For data transfer instructions, the destination operand will be tainted if and only if the source operand is tainted. For data operation instructions, the destination operand will be tainted if and only if either source or destination operand is tainted.

Finally, we use initialization analysis to reduce false positives. We observed that the operands of self-modifying and indirect jump code must be initialized. Specifically, the jump target of indirect jump should be initialized; the operands of memory updating or writing instructions in self-modifying code should be initialized. If they are uninitialized, we will not consider them as attacks.

4. EXPERIMENTAL RESULTS

To evaluate the detection effectiveness of STILL, we collected 12,000 polymorphic attack messages from 10 publicly available polymorphic engines, all of which encrypt the original shellcode. Among these ten, seven engines are from the Metasploit framework [2], including Countdown, Alpha2, JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai. The other three engines are CLET , ADMmutate, and JempiScodes . ShikataGaNai, CLET, ADMmutate, and JempiScodes are advanced polymorphic engines, which also obfuscate the decryption routine by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to defeat data mining methods.

We generated 1,000 different attack messages per each of ADMmutate and CLET. For JempiScodes, we generated 3,000 different attack messages, 1,000 per each of its three obfuscation algorithms. We also generated 7,000 different attack messages using the Metasploit Framework, 1,000 per each of the following engines, Alpha2, JumpCallAdditive, Countdown, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai. We tested the stand-alone prototype of STILL using these 12,000 attack messages. All of these messages are successfully detected.

5. CONCLUSION

We proposed STILL, a novel static taint and initialization analysis approach, to protect web servers from binary code-injection attacks. Our experiments show that STILL detect self-modifying code or indirect jumps with a high accuracy.

Acknowledgments This research was supported by the National Science Foundation (CAREER NSF-0643906).

6. REFERENCES

[1] Computer emergency response team (cert). http://www.cert.org.
[2] The metasploit project. http://www.metasploit.com.
[3] Ramkumar Chinchani and Eric Van Den Berg. A fast static analysis approach to detect exploit code inside network flows. In *RAID*, 2005.
[4] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.
[5] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *DIMVA*, 2006.
[6] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *15th Usenix Security Symposium*, July 2006.