International Journal of Computer and Communication Technology

Volume 6 | Issue 3 Article 2

July 2015

PREVENTING BUFFER OVERFLOW ATTACKS WITH RELIABLE AND LOW COST SYSTEM

B V S NAGARAJU
St.Anns College of Engineering & Technology, Chirala, AP, India, nag8888@gmail.com

Y. SRINIVASA RAO
St.Anns College of Engineering & Technology, Chirala, AP, India, tysr_cse@yahoo.co.in

P. HARINI
St.Anns College of Engineering & Technology, Chirala, AP, India, hpogadadanda@yahoo.co.in

PREVENTING BUFFER OVERFLOW ATTACKS WITH RELIABLE AND LOW COST SYSTEM

[1]B V S NAGARAJU, [2]Y. SRINIVASA RAO & [3]P. HARINI

St.Anns College of Engineering & Technology, Chirala, AP, India
Email: nag8888@gmail.com, tysr_cse@yahoo.co.in, hpogadadanda@yahoo.co.in

Abstract: Detection of Data Flow Anomalies There are static or dynamic methods 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 either due to the need for real execution of a program with some inputs. Their scheme is rule-based, whereas SigFree 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 Four rules (or cases) are discussed in this work.

Keywords: *Signature free, SigFree, Buffer Overflow, Worm, Security.*

1. INTRODUCTION

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. SigFree first blindly dissembles and extracts instruction sequences from a request. It then applies a novel technique called *code abstraction*, which uses data flow anomaly to prune 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. 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 transparent 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 showed that SigFree could block all types of code injection attack packets (above 250) tested in our experiments. Moreover, SigFree causes negligible throughput degradation to normal client requests.

We proposed SigFree, a real-time, signature free, out of- the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats, to various Internet services. SigFree does not require any signatures, thus it can block new, unknown attacks

We have implemented a SigFree prototype as a proxy to protect web servers. Our empirical study shows that there exists clean-cut "boundaries" between code embedded payloads and data payloads when our code data separation criteria are applied. We have identified the "boundaries" (or thresholds) and been able to detect/ block all 50 attack packets generated by Metasploit framework , all 200 polymorphic shellcode packets generated by two well-known polymorphic shellcode engine ADMmutate and CLET , and worm Slammer, CodeRed and a CodeRed variation, when they are well mixed with various types of data packets. Also, our experiment results show that the throughput degradation caused by SigFree is negligible.

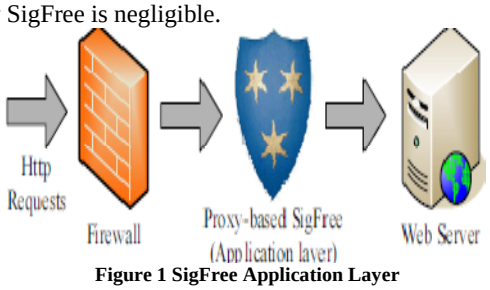

Figure 1 SigFree Application Layer

2. NETWORK ARCHITECTURE

1. Prevention/Detection of Buffer Overflows: 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 botnets. Buffer overflow attacks are the most popular choice in these attacks, as they provide substantial control over a victim.

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 not limited to model checking and bugs-asdeviant- behavior.

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."

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax , LibSafe and e-NeXsh .

Class 1C techniques need to modify the OS. In contrast, SigFree 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 [4]. 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 . Bhatkar and Sekar proposed a comprehensive address space randomization scheme. Addressspace randomization, in its general form , can detect exploitation of all memory errors.

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows area code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected.

2.Worm Detection and Signature Generation

The implementation of their approach is resilient to a number of code transformation techniques. Although their techniques also handle binary code, they perform offline analysis. In contrast, SigFree is an online attack blocker. As such, their techniques and SigFree are complementary to each other with different purposes. Moreover, unlike

SigFree, their techniques 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

3. SigFree Attack Model

An attacker exploits a buffer overflow vulnerability of a web server by sending a crafted request, which contains a malicious payload. Figure 3 shows the format of a HTTP request. There are several HTTP request methods among which GET and POST are most often used by attackers.

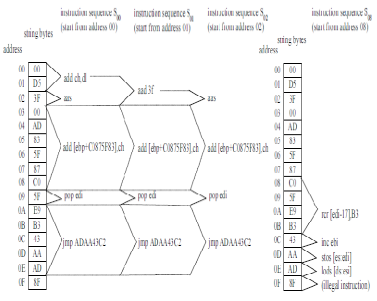

Figure 2 Instruction sequences distilled from a Substring of a GIF file

Although HTTP 1.1 does not allow GET to have a request body, some web servers such as Microsoft IIS still dutifully read the request-body according to the request-header's instructions (the CodeRed worm exploited this very problem). The position of a malicious payload is determined by the exploited vulnerability. A malicious payload may be embedded in the Request-URI field as a query parameter.

However, as the maximum length of Request-URI is limited, the size of a malicious payload, hence the behavior of such a buffer overflow attack, is constrained. 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 have not been observed yet. In this work, we assume an attacker can use any request method and embed the malicious code in any field.

*4. URI decoder***.**

The specification for URLs 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 . Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI.

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 <= 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
```

*5. ASCII Filter***:**

Malicious executable code is 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, SigFree allows the request to pass we will discuss a special type of executable codes called alphanumeric shellcodes that actually use printable ASCII) .

6. Instruction sequences distiller:

This module distills all possible instruction sequences from the query parameters of Request-URI and Request-Body (if the request has one). *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 fragment of) a program.

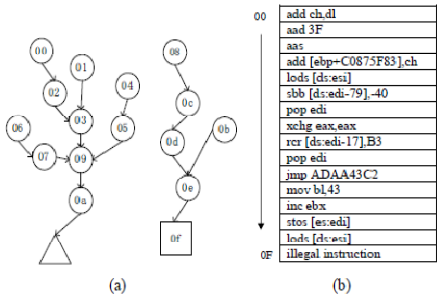

Figure 3 Data structure for the instruction sequences distilled

3. TESTING THE SYSTEM

Clearly, it is critical to set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. To find out the appropriate thresholds, we tested both schemes of SigFree against 50 unencrypted attack requests generated byMetasploit framework, worm Slammer, CodeRed (CodeRed.a) and a CodeRed variation (CodeRed.c), and 1500 binary HTTP replies (52 encrypted data, 23 audio, 195 jpeg, 32 png, 1153 gif and 45 flash) intercepted on the network of College of Information Science and Technology. Note that we tested on HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies include more diverse binaries (test over real traces of web requests is reported). Also note that although worm Slammer attacks Microsoft SQL servers rather than web servers, it also exploits buffer overflow vulnerabilities.

We tested SigFree over real traces. Due to privacy concerns, we were unable to deploy SigFree in a public web server to examine realtime web requests. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded a normal user's http requests during his/her daily Internet surfing. During a one-week period, more than ten of our lab members installed the proxy and helped collect totally 18,569 HTTP requests. The requests include manually typed urls, clicks through various web sites, searchings from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPs requests. In this way, we believe our data set is diverse enough, not worse than that we might have got if we install SigFree in a single web server that provides only limited Internet services.

Our test based on the above real traces did not yield an alarm. This output is of no surprise because our normal web requests do not contain code.

To evaluate the performance of SigFree, we implemented a proxy-based SigFree prototype using the C programming language inWin32 environment. SigFree was compiled with Borland C++ version 5.5.1 at optimization level O2. The prototype implementation was hosted in a Windows 2003 server with Intel Pentium 4, 3.2GHz CPU and 1G MB memory.

The proxy-based SigFree prototype accepts and analyzes all incoming requests from clients. The client testing traffics were generated by Jef Poskanzer's http load program 3 from a Linux desktop PC with Intel Pentium 4 2.5GHz CPU connected to theWindows server via a 100 Mbps LAN switch. We modified the original http load program so that clients can send code-injected data requests.

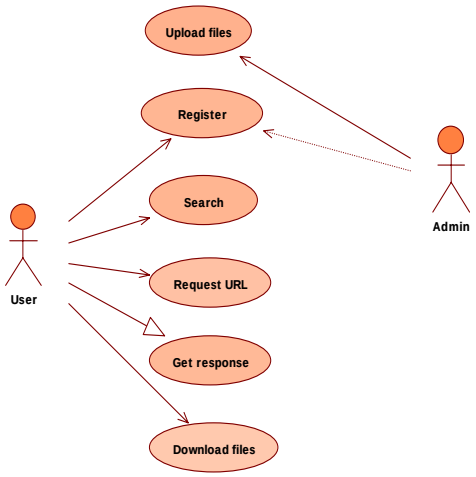

Figure 4 Use case Diagram

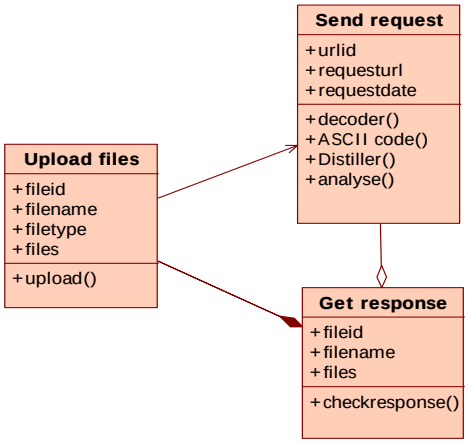

Figure 5 Class Diagram

4. RESULTS AND CONCLUSIONS

Most malware detection schemes include 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 [9, 3] and attackers may use them to evade detection. Table 1 shows that SigFree is robust to most of these obfuscation techniques.

Junk byte insertion is one of the simplest obfuscation against disassembly. Here junk bytes are inserted at locations that are not reachable at run-time. This insertion however can mislead a linear sweep algorithm, but can not mislead a recursive traversal algorithm [3], which our algorithm bases on.

Opaque predicates are used to transform unconditional jumps into conditional branches. Opaque predicates are predicates that are always evaluated to either true or false regardless of the inputs. This allows an obfuscator to insert junk bytes either at the jump target or in the place of the fall-through instruction.

We note that opaque predicates may make SigFree mistakenly interpret junk byte as executable codes. However, this mistake will not cause SigFree to miss any real malicious instructions. Therefore, SigFree is also immune to obfuscation based on opaque predicates.

SigFree also has several limitations. First, SigFree 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, the executable shell codes could be written in alphanumeric form [5]. Such shell codes will be treated as printable ASCII data and thus bypass our analyzer.

By turning off the ASCII filter, Scheme 2 can successfully detect alphanumeric shellcodes; however, it will increase unnecessary computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

We proposed SigFree, a real-time, signature free, out of- the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats, to various Internet services. SigFree does not require any signatures, thus it can block new, unknown attacks. SigFree is immunized from most attack-side code obfuscation methods, good for economical Internet wide deployment with little maintenance cost and negligible throughput degradation, and can also handle encrypted SSL messages.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their comments which were very helpful in improving the quality and presentation of this paper.

REFERENCES:

[1] Buffer overrun in jpeg processing (gdi+) could allow code execution 833987 http://www.microsoft.com/technet/security/bulletin/ MS04-028.mspx

[2] Fnord snort preprocessor. http://www.cansecwest.com/spp fnord.c.

[3] Intel ia-32 architecture software developer's manual volume 1: Basic architecture.

[4] Metasploit project. http://www.metasploit.com.

[5] Security advisory: Acrobat and adobe reader plug-in buffer overflow.http://www.adobe.com/support/techdocs/321644.htm

[6] Stunnel – universal ssl wrapper. http://www.stunnel.org.

[7] Symantec security response: backdoor.hesive. http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html

[8] Winamp3 buffer overflow. http://www.securityspace.com/ smysecure/ catid.html?id=11530.

[9] Pax documentation. http://pax.grsecurity.net/docs/pax.txt, November 2003.

[10]BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In Proc. 2000 USENIX Technical Conference (June 2000).