Volume 4, Issue 3, March 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

SigFree: A Signature-Free Buffer Overrun and Overflow Attack Blocker

Syed Zubair, Abu Taha Zamani
Lecturer,Deanship of Information Technology,
Northern Border University,Kingdom of Saudi Arabia

Abstract: ***This article propose SigFree, a real time, signature-free, out-of the-box, overwrites adjacent memory, application layer blocker for preventing buffer overrun and overflow attacks, one of the most serious threats to Software vulnerabilities, cyber security, cloud storage and memory safety. Moreover, buffer overflow and overrun vulnerabilities dominate the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. Sigfree can filter out code injection buffer overrun and overflow attack messages targeting at various Internet services such as e-mail, file transfer, and command execution on remote system and web service. Motivated by the observation that buffer overrun and overflow attacks typically contain executables whereas legitimate client requests never contain executables in almost all Internet services, SigFree blocks attacks by detecting the presence of code and executables. SigFree is signature free, thus it can block new and unknown buffer overflow and overrun attacks. . In this paper, we survey the various types of buffer overflow and overrun vulnerabilities and attacks, and survey the various defensive measures that mitigate buffer overflow and overrun vulnerabilities, including our own Stack Guard method. We then consider which combinations of techniques can eliminate the problem of buffer overflow and overrun vulnerabilities, while preserving the functionality and performance of existing systems. 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 overrun and overflow and overrun attacks; SigFree is also immunized from most attack-side by source or machine code. Since SigFree is transparent to the servers being protected, it is good for economical Internet wide deployment with very low maintenance. We implemented and tested SigFree; our experimental study showed that SigFree could block all types of code injection attack packets (above 750) tested in our experiments. We are able to explicitly obfuscate the true intent of the code. All the resulting attacks Defeat the widely used in Intrusion Detection System.***

Keywords— Intrusion Detection System, Buffer overflow and overrun and overrun attacks, code injecting 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. Buffer overflow attacks are the most popular choice in these attacks, as they provide substantial control over a victim host. "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." (Note that the buffer could be in stack or heap.) Although taking a broader viewpoint, buffer overflow attacks do not always carry code in their attacking requests (or packets) 1, code-injection buffer overflow attacks such as stack smashing count for probably 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. (a) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardwares, 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. (b) Class 1F defenses can be very secure, but they either suffer from significant run time 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. 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 relies on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation. To overcome the above limitations, in this paper we propose SigFree, a real time buffer overflow attack blocker, to protect Internet services. The idea of SigFree 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 , (a) 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) accept data only; workstation services (ports 139 and 445) accept data only. (b) 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 executable code, this observation indicates that if we can precisely distinguish (service requesting) messages that contain code from those that do not contain any code, we can protect most Internet services (which accept data only) from code-injection buffer overflow attacks by blocking the messages that contain code. The merits of SigFree are summarized below. They show that SigFree has taken a main step forward in meeting the four requirements afore mentioned.

- 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, 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 has negligible through output degradation.

Buffer overflows were understood and partially publicly documented as early as 1972, when the Computer Security Technology Planning Study laid out the technique: "The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine." Today, the monitor would be referred to as the kernel. The earliest documented hostile exploitation of a buffer overflow was in 1988. It was one of several exploits used by the Morris worm to propagate itself over the Internet. The program exploited was a service on Unix called finger. Later, in 1995, Thomas Lopatic independently rediscovered the buffer overflow and published his findings on the Bugtraq security mailing list. A year later, in 1996, Elias Levy (aka Aleph One) published in Phrack magazine the paper "Smashing the Stack for Fun and Profit", a step-by-step introduction to exploiting stack-based buffer overflow vulnerabilities. Since then, at least two major internet worms have exploited buffer overflows to compromise a large number of systems. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services (IIS) 5.0 and in 2003 the SQL Slammer worm compromised machines running Microsoft SQL Server 2000. In 2003, buffer overflows present in licensed Xbox games have been exploited to allow unlicensed software, including homebrew games, to run on the console without the need for hardware modifications, known as modchips. The PS2 Independence Exploit also used a buffer overflow to achieve the same for the PlayStation 2. The Twilight hack accomplished the same with the Wii, using a buffer overflow in The Legend of Zelda: Twilight Princess. In 1988 when Morris large scaled attack based on buffer overflows, one of the most spread virus attacks of cyber world awakes. 23 years later buffer overflow attacks are daily schedule. Till now attacks are not stopped or nor controlled. The code red worm from 2001 caused 2.5 billion USD damage, and this is only one large scale buffer overflow misusing. Since C language does not enforce bounds checking or certain standard functions it is possible to write more data to buffer than it can store. This leads to overrun of buffer which can be used by the attacker to modify the state. The Consequences of buffer overflow are that attacker can use buffer overflow to inject code and execute code afterwards. Each week, security vulnerabilities are discovered in widely deployed software. Many of these security threats stem from buffer-overflow exploitation by which a malicious user attempts to gain control of a computer system by overwhelming it with skillfully crafted input data. Most of these vulnerabilities are detectable at compile time; however, few compilers provide such capabilities. Buffer overflows have been detected in many types of software ranging from Web browsers to Web servers. Software such as Internet Explorer, Hypertext Preprocessor (PHP), and Apache all have been victim to such vulnerabilities. Because of the widespread availability and use of vulnerable software, buffer-overflow exploits can be a serious threat to system and data integrity. To make matters worse, malicious users often write programs to help others easily exploit these software flaws. To fully understand how buffer overflows work in helping a malicious user take control of a system, we must both examine some fundamental Computer Science (CS) concepts and also view sample code to probe more deeply into the details of these exploits. The C code examples provided in this article are designed to work with the i686 architecture. Buffer overflows generally occur on the heap or the stack, as explained below. However, the data on the heap does not often control instruction flow and therefore is usually not of interest. This article focuses solely on

buffer overflow that occurs on the stack. The act of writing data on the stack to disrupt normal program execution is called stack-smashing.

2. Existence System

Detection of Data Flow Anomalies There is 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.

It takes considerable effort to prevent buffer overflows. On the one hand static methods produce false / negative results, which cause manual corrections in the source code by the developer. On the other hand instrumentation methods have lot of overhead and they are not transparent. Stack based methods does not prevent from all attacks. Hardware methods provide less overhead but need to have deeper architectural changes. The dynamic methods are too expensive to protect the systems against buffer overflow attacks.

Methods used in existing system are

2.1 Stack Based

Stack based: Adding redundant information / routines to protect the stack or parts of stack.

2.1.1 Stack guard

A simple approach to protect programs against stack smashing and with little modification against EBP overflows. This is achieved by a compiler extension that adds so called canary values before the EIP saved at the function prologue (see Figure 2.1). Before the return of a protected function is executed, the canary values are checked. If the canary values were not modified Stack guard implies that the programs integrity is assured (assuming that the canary values were altered, the program execution is aborted and an exception is issued). This approach protects only against stack smashing. However, there are still ways to overcome Stack guard. An attacker could guess the canary values. This is quite hard, if the values are chosen randomly for each guarded function, but it is also possible to choose a canary value made of terminator characters, which makes every string/file copy function to stop at the canary value. So even restoring the canary value would not lead to a successful program flow detour. Another way to thwart Stack guard is to find a pointer to overflow that it points to the address of the saved EIP and use that pointer as target for a copy function. This way the EIP is overwritten without modifying the canary values. Overhead produced is moderate with up to 125% and that this method is not transparent, meaning that the source code is needed for recompilation. This fact makes Stack guard useless for many legacy software products on the market, because they are not open source.

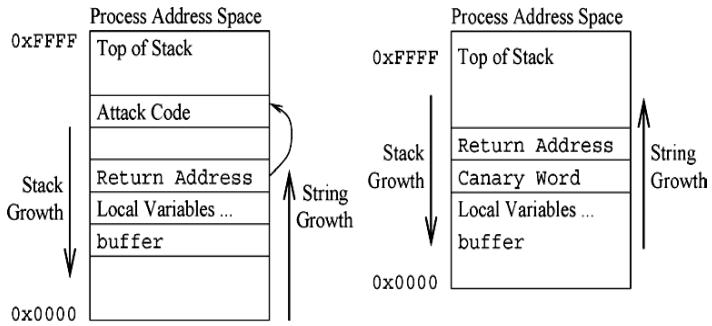

Figure 2.1 stack layout using stack guard

2.1.2 Libsafe and libverify

Two methods that should protect against buffer overflow attacks. The first method is libsafe, a transparent approach set up in a DLL that replaces standard (vulnerable) functions by standard bounds checked functions (e.g. strcpy could be replaced by strncpy). The upper limit of the bounds is calculated based on the EBP, so the maximum amount written to a buffer is the size of the stack frame. This method only works if the EBP can be determined, since there exist compiler options that make this impossible; further compatibility issues could arise with legacy software.

2.2 Instrumentation

Instrumentation: Replacing of standard functions / objects like pointers to equip them with tools.

2.2.1 Safe pointer

Safe pointer structure is to detect all pointer and array access errors. Meaning that both, temporal and spatial errors are detected.

The structure consists of five entries:

a. Value (the value of the safe pointer, it may contain any expressible address)
b. Base (the base address of the referent)
c. Size (the size of the referent in bytes)
d. Storage class (either Heap, Local or Global)
e. Capability (unique capability. Predefined capabilities are forever and never, else it could be an enumerated number as long as its value is unique)

Base and size are spatial attributes capability and storage class are temporal attributes. The capability is also stored in a capability store when it is issued and deleted if the storage is freed or when the procedure invocation returns.

This ensures that storage that is not available (like freed heap allocated memory) is not accessed anymore. The transformation of a program from unsafe to safe pointers involves pointer conversion (to extend all pointer definitions), check insertion (to instrument the program to detect memory access errors) and operator conversion (to generate and maintain object attributes).

Safe pointers work with spatial and temporal access errors. This method protects the stack and the heap memory against overflowing of arrays. Attacks that are not prevented are off-by-one, since it is a typical programmer's error and signed/unsigned overflows. Another disadvantage is the overhead, since the size of the safe pointer itself is 15 bytes, the overhead is 275%. The text segment overhead ranges from 35% to 300% and the total size of the Data, BSS and heap segment has an overhead ranging from 5% up to 330%. With run-time optimizations10 the overall execution overhead ranges from 130% to 540%. Last but not least this method is not transparent, because recompilation is needed to implement safe pointers.

2.2.2 C Range Error Detector (CRED)

CRED is the idea to replace every out-of-bounds (OOB) pointer value with the address of a special OOB object created for that value. To realize this, a data structure called object table collects the base address, and size information of static, heap and stack objects. To determine, if an address is in-bounds, the checker first locates the referent object by comparing the in-bounds pointer with the base and size information stored in the object table. Then, it checks if the new address falls within the extent of the referent object. If an object is out-of-bounds, an OOB object is created in the heap that contains the OOB address value and the referent object. If the OOB value is used as an address it is replaced by the actual OOB address. The OOB objects are entered into an out-of-bounds object hash table, so it is easy to check if a pointer points to an OOB object by consulting the hash table. The hash table is only consulted if the checker is not able to find the referent object in the object table or cannot identify the object as unchecked. Arithmetic and comparison operations to OOB objects are legal, since the referent object and its value is retrieved from the OOB object. But if a pointer is dereferencing, it is checked if the object is in the object table or if it is unchecked else the operation is illegal. Buffer overflows are not prevented but the goal is thwarted because copy functions need to dereference the OOB value, the program is halted before more damage happens. This fact could be still used as DoS attack, since the program (service) is halted and needs to be restarted or even worst if it has to be re-administrated. This method works, by comparing the non-instrumented code, the CRED instrumented code and a code where the instrumentation is which is the base for CRED. Since recompilation is needed, this method is not transparent. But the instrumented code is fully compatible to non-instrumented code. The overhead of this approach ranges from 1% to 130%, but it do not show how certain kind of buffer overflow attacks, like signed/unsigned and off-by-one overflows are handled. The same arguments as on the safe pointers in the previous section can be applied here.

2.3 Hardware based

Hardware based: Architecture check for illegal operations and modifications.

The approach deals with an architectural change implementing a Secure Return Address Stack (SRAS), which is a cyclic, finite LIFO structure that stores return addresses. At a return call the last SRAS entry is compared with the return address from the stack and if the comparison yields that the return address was altered the processor can terminate the process and inform the operation system or continues the execution based on the SRAS return address. Since the SRAS is finite and cyclic, n/2 of the SRAS content has to be swapped on an under- or overflow.

Two methods:

a. OS-managed SRAS swapping. The operation system executes code that transfers contents to or from memory which is mapped to physical ages that can only be accessed by the kernel
b. Processor-managed SRAS swapping. The processor maintains two pointers to two physical pages that contain spilled SRAS addresses and a counter that indicates the space left in the pages. If the Pages over or underflow, the OS is invoked to de-allocate pages, else the processor can directly transfer contents to and from the pages without invoking the OS.

The performance impact of this approach is due to the swapping, since the hardware operation saving an address is small. The examples for upper and lower bounds that could be used as stack size and with a "good" choice the performance impact of CPU swapping is about 1% or less and OS swapping has an impact of 1% up to 67,9%. Another shortcoming is that only the return address is protected, which prevents an attacker from redirecting the control flow. But heap overflows can be used to gain control of the program flow to, not to mention that other stack based attacks are still working. It also mentions that their approach should be applied in conjunction with other countermeasures. The problem is that the SRAS is not compatible with non LIFO routines, such as C++ exception handling. This makes it necessary to change the non LIFO routines to LIFO routines or it must be possible to turn off the SRAS protection.

2.4 Static

Static: Checking the source code for known vulnerable functions, do flow analyses check the correct boundaries, use of heuristics?

This deals with the idea to comment the source code that LCLint can interpret them and generate a log file which can be used to identify possible vulnerabilities. If a source code is analyzed, LCLint evaluates conditions to fulfill safe execution of the finally compiled program. These conditions are written to the log file so the programmer and check if these conditions are true for every case that could happen while execution. Then the programmer can write control comments

into the source to let LCLint what conditions are fulfilled or if LCLint should ignore parts of the source code. These way errors can be found before compilation, but this method has certain shortcoming. Since it is not possible to efficiently determine invariants, to take advantage of idioms used typically by C programmers. Since this method is not exact, the rate of false positives grows. Further LCLint is a lightweight checker, meaning that the program flow is also checked using heuristics, since determining all possible program states might need exponential time. The false positives can be commented out, but this means more work to the developers of the software and since heuristics are used, false negatives are produced either. All these facts and the fact that this method is not transparent make it only suitable for new or small projects. The last aspect we want to point out is that this is the only method so far that produces no overhead, since the compilers skip comments.

2.5 Operation System Based

Operation system based: Declare the stack as non-executable to prevent code execution.

2.5.1 Data Execution Prevention (DEP)

With the release of the Service Pack 2 for Windows XP and Service Pack 1 for Windows 2003 a new protection was introduced to machines using these operation systems, the DEP. Microsoft explains a bit how the DEP works. In cooperation with Intel (Execute Disable bit feature) and AMD (no-execute page-protection processor feature) a new CPU flag was implemented called the NX-Flag. It marks all memory locations in a process as non-executable unless the location explicitly contains executable code. If the machine running with DEP support has no NX-Flag, the DEP can be enforced by the operation system (software enforcement). This protection prevents execution of injected code, if the code was injected in a non-executable area. The DEP can be bypassed. Further this method requires that even valid, working processes are (sometimes) recompiled. Another shortcoming is, that this method does not prevent the buffer overflow itself, so attacks like variable attack or BSS/heap overflows are not prevented.

2.6 Solar Designer

The Solar Designer patch does nearly the same as the DEP, but it makes the stack non executable. Since Linux needs the executable stack for signal handling, this restricts the normal behavior of Linux. If the attack is able to determine code that would act like a shell code and execute this code instead of injected code the patch can be bypassed. To conclude, buffer overflows are not prevented, only the code execution. Attacks like the variable attack or BSS/heap overflows are still possible, and heap overflows can also be used to execute arbitrary code.

3. Proposed System

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 their project: Case 1 not only assumes the occurrence of the call/jmp instructions, but also expects the push instruction appears before the branch; Case 2 relies on the interrupt instruction; Case 3 relies on instruction ret; Case 4 exploits hidden branch instructions. Besides, they used a special rule to detect polymorphic exploit code which contains a loop. Although they mentioned that the above rules are initial sets and may require updating with time, it is always possible for attackers to bypass those pre-known rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, SigFree exploits a different data flow analysis technique, which is much harder for exploit code to evade.

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

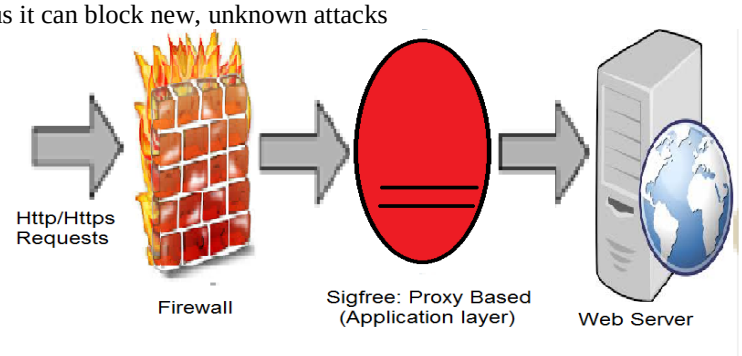

Figure 3.1 Signature Free prototype

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 Meta spoilt 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.

3.1Buffer Overflow Variants

Today buffer overflow attacks are known and well understood. In general every buffer that can be accessed by an attacker might be compromised if vulnerable functions are used. Such variables are located on the stack and heap. The attacks are partitioned as follows

a. Stack smashing used to execute inject code
b. Variable attack used to modify program state
c. Heap overflow used to execute arbitrary code or to modify the variables
d. Off-by-one a classic programmers error , only one byte is overwritten

3.2 Input Design

The input design is the link between the information system and the user. It comprises the developing specification and procedures for data preparation and those steps are necessary to put transaction data in to a usable form for processing can be achieved by inspecting the computer to read data from a written or printed document or it can occur by having people keying the data directly into the system. The design of input focuses on controlling the amount of input required, controlling the errors, avoiding delay, avoiding extra steps and keeping the process simple. The input is designed in such a way so that it provides security and ease of use with retaining the privacy. Input Design considered the following things:

a. What data should be given as input?
b. How the data should be arranged or coded?
c. The dialog to guide the operating personnel in providing input.
d. Methods for preparing input validations and steps to follow when error occur.

3.3 Objectives

Input Design is the process of converting a user-oriented description of the input into a computer-based system. This design is important to avoid errors in the data input process and show the correct direction to the management for getting correct information from the computerized system.It is achieved by creating user-friendly screens for the data entry to handle large volume of data. The goal of designing input is to make data entry easier and to be free from errors. The data entry screen is designed in such a way that all the data manipulates can be performed. It also provides record viewing facilities. When the data is entered it will check for its validity. Data can be entered with the help of screens. Appropriate messages are provided as when needed so that the user will not be in maize of instant. Thus the objective of input design is to create an input layout that is easy to follow.

3.4 Output Design

A quality output is one, which meets the requirements of the end user and presents the information clearly. In any system results of processing are communicated to the users and to other system through outputs. In output design it is determined how the information is to be displaced for immediate need and also the hard copy output. It is the most important and direct source information to the user. Efficient and intelligent output design improves the system's relationship to help user decision-making. Designing computer output should proceed in an organized, well thought out manner; the right output must be developed while ensuring that each output element is designed so that people will find the system can use easily and effectively. When analysis design computer output, they should Identify the specific output that is needed to meet the requirements. Select methods for presenting information. Create document, report, or other formats that contain information produced by the system. The output form of an information system should accomplish one or more of the following objectives.

a. Convey information about past activities, current status or projections of the Future.
b. Signal important events, opportunities, problems, or warnings.
c. Trigger an action.
d. Confirm an action.

3.5 Architecture

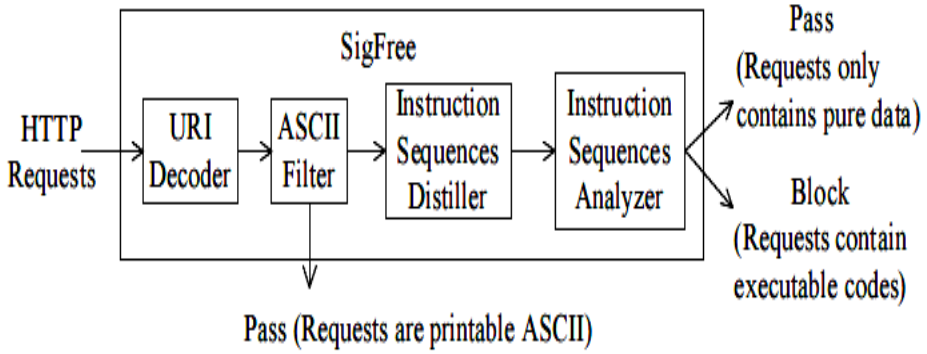

Figure 3.5 Architecture of SigFree

4. System Implementation

Implementation is the stage of the project when the theoretical design is turned out into a working system. Thus it can be considered to be the most critical stage in achieving a successful new system and in giving the user, confidence that the new system will work and be effective. The implementation stage involves careful planning, investigation of the existing system and it's constraints on implementation, designing of methods achieve changeover and evaluation of changeover methods.

4.1 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 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, SigFree 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." Three such compilers are Stack-Guard, Pro Police 2, and Return Address Defender (RAD). In addition, Smirnov and Chiueh proposed Compiler DIRA 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, SigFree 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 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. 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. Address space randomization, in its general form, can detect exploitation of all memory errors. Instruction set randomization can detect all code injection attacks. 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 area 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.

4.2 Worm Detection and Signature Generation

Because buffer overflows are a key target of worms when they propagate from one host to another, SigFree 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. [Class2B] techniques use such local traffic symptoms as content invariance, content prevalence and address dispersion to generate worm signatures and/or block worms.

4.3 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 packets. Along purpose P1, Chritodorescu and Jha proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. Investigated disassembly of obfuscated binaries. SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, instead of determining if a piece of code has malicious intent or not. (Note that SigFree does not check if the code contained in a message has malicious intent.) Due to this reason, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree.

4.3.1 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. 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 Code Red 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 has not been observed yet. In this work, we assume an attacker can use any request method and embed the malicious code in any field.

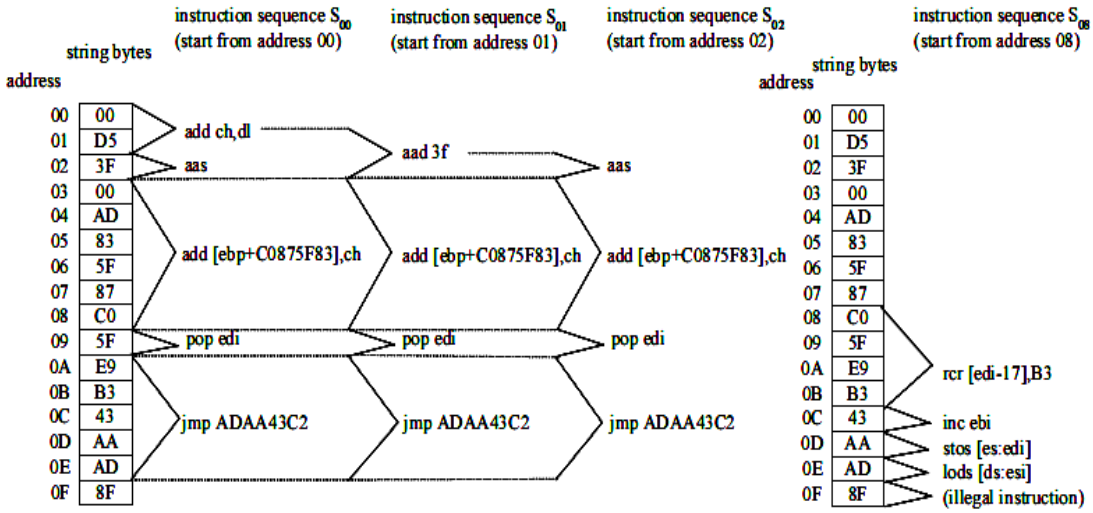

Figure 4.1 Instruction Sequence Distilled

4.3.2 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.

4.3.3 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 shell codes that actually use printable ASCII).

4.3.4 Instruction sequences distiller (ISD).

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.

Algorithm Distill all instruction sequences from a request
initialize EISG G and instruction array A to empty
for each address k of the request do
add instruction node k to G
k ← the start address of the request
while k <= the end address of the request do
i decode an instruction at k
if i is illegal then
A[k] ← illegal instruction i
set type of node k "illegal node" in G
else
A[k]← instruction i
if i is a control transfer instruction then
for each possible target t of i do
if target t is an external address then
add external address node t to G
add edge e(node k, node t) to G
else
add edge e(node k, node k + i.length) to G
k ← k + 1

Figure 4.2 Algorithm to distill instruction sequence

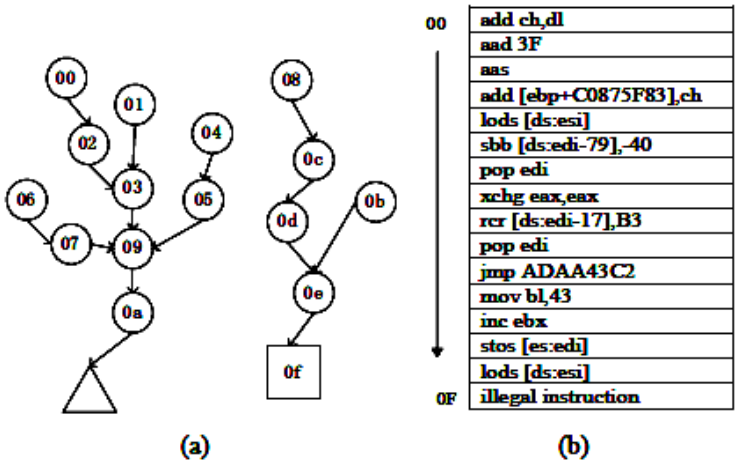

Figure 4.3 Data structure for instruction sequences distilled

5. Conclusion

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. A combination of developer education for defensive programming techniques as well as software reviews is the best initial approach to improving the security of custom software. Secure programming and scripting languages are the only true solution in the fight against software hackers and attackers.

References

[1] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu,"SigFree: A Signature-Free Buffer Overflow Attack Blocker", IEEE Transactions On DependableAnd Secure Computing, Vol.7, No.1, January-March 2010.
[2] Eric Haugh and Matt Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities".
[3] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole "Buffer Overflows: Attacks and Defenses forthe Vulnerability of the Decade",http://www.cse.ogi.edu/DISC/projects/immunix.
[4] Hassen Sallay,Khalid A. AlShalfan, Ouissem Ben Fred j,"A scalable distributed IDS Architecture for High speed Networks", IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.8, August 2009.
[5] Buffer overflows vulnerability diagnosis for commodity software by jiang.pp 6-18
[6] Buffer overflow attacks by James C Foster pp 47-49
[7] Chinta.Srihari, Lalu Banothu, "Sigfree-A Signature-Free Buffer Overflow Attack Blocker" International Journal of Engineering and Science ISSN: 2278-4721, Vol. 1, Issue 6 (October 2012), PP 14-31
[8] Pankaj B. Pawar, Malti Nagle, Pankaj K. Kawadkar, "Prevention of Buffer overflow Attack Blocker Using IDS" International Journal of Computer Science and Network (IJCSN) Volume 1, Issue 5, October 2012 www.ijcsn.org ISSN 2277-5420.
[9] L.RaghavendarRaju, Prof.D. Jamuna, M.JanardhanReddy. "Blocker: A Blockerof Signature Free Buffer Overflow Attack" International Journal of Engineering Research & Technology (IJERT) Vol. 1 Issue 7,September - 2012
[10] Kotha Jothsna, Dr.R.V.Krishniah "A Signature-Free Buffer Overflow Attack Detection Using DST" International Journal of Advanced Research in Computer Science and Software Engineering www.ijarcsse.com Volume 3, Issue 6, June 2013
[11] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," IEEE Security and Privacy, vol. 2, no. 4, 2004.
[12] Intel ia-32 architect software developer's manual volume 1: Basic architecture.pp 9
[13] Metasploit project. http://www.metasploit.com. pp 47-68
[14] Security advisory: Acrobat and adobe reader plug-in buffer overflow. http://www.adobe.com/support/techdocs/321644.html. pp 13-15
[15] Stunnel – universalssl wrapper. http://www.stunnel.org.
[16] Symantec securityresponse: back door. hesive. pp 6-18 http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html
[17] Winamp3 buffer overflow. http://www.securityspace.com/smysecure/catid.html?id=11530. pp 6-18
[18] Pax documentation. http://pax.grsecurity.net/docs/pax.txt, November 2003 against stack smashing attacks.

