Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 2, Issue. 3, March 2013, pg.95 – 105

RESEARCH ARTICLE

Signature Free Virus Blocking Method to Detect Software Code Security

N.Magarani [1], ***K.Devipriya*** [2], ***S.Madhan Kumar*** [3], ***T.Viknesh Kumar*** [4], ***K.Kumaresan*** [5]

[1]Computer Science and Engineering, K.S.R College of Engineering, Tiruchengode
kavimaha92@gmail.com

[2]Computer Science and Engineering, K.S.R College of Engineering, Tiruchengode
priyaksrcse@gmail.com

[3]Computer Science and Engineering, K.S.R College of Engineering, Tiruchengode
madhanks89@gmail.com

[4]Computer Science and Engineering, K.S.R College of Engineering, Tiruchengode
vickeystormchaser@gmail.com

[5]Assistant Professor, Computer Science and Engineering, K.S.R College of Engineering, Tiruchengode
mkkumz@gmail.com

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

Key Terms: - Distilling Instruction Sequence Distiller, Instruction Sequencer Analyser, Proxy Based Sigfree, Code Abstraction and SSL Proxy.

I. INTRODUCTION

The history of cyber security, buffer over- flow 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. 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), 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.

System Model

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. 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

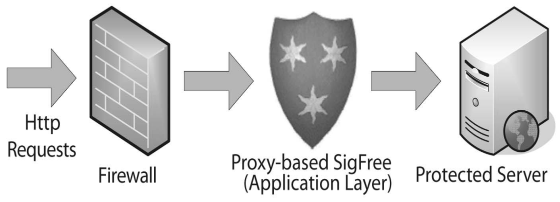


Fig. 1. SigFree is an application layer blocker between the protected server and the corresponding firewall.

To overcome the above limitations, in this paper, we propose SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code". In particular, as summarized in on Windows platforms, most web servers (port 80) accept data only. Accordingly, SigFree (Fig. 1) works as follows: SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new 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. We are using 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 contains code.

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

SigFree is signature free, thus it can block new and unknown buffer overflow attacks. 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 is an economical deployment with very low maintenance cost, which can be well justified by the aforementioned features.

II. RELATED WORK

SigFree is mainly related to three bodies of work:

- Prevention/detection techniques of buffer overflows.
- Worm detection and signature generation.

2.1 Prevention/Detection of Buffer Overflows Existing prevention/detection techniques of buffer over- flows 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. 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.

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 StackGuard, ProPolice and Return Address Defender (RAD).

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows

Class 1D: Hardware modifications. A main idea of hard-ware modification is to store all return addresses on the processor.

Class 1E: 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 can capture some—but not all—of the running symptoms of buffer overflows. For example, accessing nonexecutable 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:

Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes:

- *Macro symptoms:* To raise early warnings of Internet-wide worm infection.
- *Local traffic symptoms:* To detect content invariance, content prevalence, and address dispersion to generate worm signatures and/or block worms.
- *Worm code running symptoms:* To detect worms

Disadvantages:

SigFree is useful to implement in web servers but it is not useful to implement in real time applications because of Microsoft's Jet Database Engine.

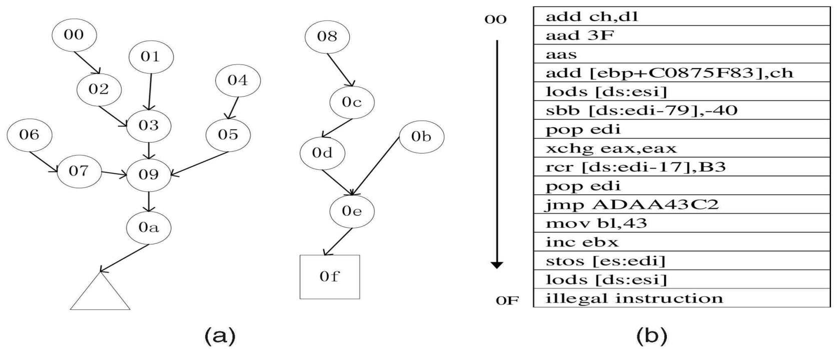
SigFree does not contain signature for new and unknown attacks.

III. Instruction Sequence Distiller

This section first describes an effective algorithm to distill instruction sequences from requests, followed by several pruning techniques to reduce the processing overhead of instruction sequence analyzer.

Distilling Instruction Sequences:

To distill an instruction sequence, we first assign an address (starting from zero) to every byte of a request, where address is an identifier for each location in the request. Then, we disassemble the request from a certain address until the end of the request is reached or an illegal instruction opcode is encountered. There are two traditional disassembly algorithms: *linear sweep and recursive traversal*



```
add ch,dl
aad 3F
aas
add [ebp+C0875F83],ch
lods [ds:esi]
sbb [ds:edi-79],-40
pop edi
xchg eax,eax
rcr [ds:edi-17],B3
pop edi
jmp ADAA43C2
mov bl,43
inc ebx
stos [es:edi]
lods [ds:esi]
illegal instruction
```

Algorithm 1 Distill all instruction sequences from a request initialize EISG G and instruction array A to empty

for each address i of the request do add instruction node i to G

i the start address of the request

while i <¼ the end address of the request do inst decode an instruction at i

```
if inst is illegal then

A½i&   illegal instruction inst

set type of node i “illegal node” in G else

A½i&   instruction inst

if inst is a control transfer instruction then for each possible target t of inst do

if target t is an external address then add external address node t to G

add edge eðnode i; node tÞ to G
else

add edge eðnode i; node i þ inst:lengthÞ to G

i (1)
```

In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from an N-byte request, w

simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is O(N^2).

One drawback of the above algorithm is that the same instructions are decoded many times. For example, instruction “pop edi” in Fig. 2 is decoded many times by this algorithm.

To reduce the running time, we design a memorization algorithm [47] by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request.

An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruc-tion array. Fig. 3 shows the data structure for the request shown in Fig. 2. The details of the algorithm for creating the data structure are described in Algorithm 1. Clearly, the running time of this algorithm is OðNÞ, which is optimal as each address is traversed only once.

- The *linear sweep algorithm* begins disassembly at a certain address and proceeds by decoding each encoun-tered instruction.
- The *recursive traversal algorithm* also begins disassembly at a certain address, but it follows the control flow of instructions. In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from an N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is O(N^2). One drawback of the above algorithm is that the same instructions are decoded many times.

3.1 Excluding Instruction Sequences:

The previous step may output many instruction *sequences* at different entry points. Next, we exclude some of them based on several heuristics. Here, excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any).

The fundamental rule in excluding instruction sequences is not to affect the decision whether a request contains code or not. if a request contains a fragment of a program, the fragment must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from a remaining sequence only by few instructions.

In the fig A,

Step 1. If instruction sequence s_a is a subsequence of instruction sequence s_b, we exclude s_a. The rationale for excluding s_a is that if s_a satisfies some characteristics of programs, s_b also satisfies these characteristics with a high probability.

Step 2. If instruction sequence s_a merges to instruction sequence s_b after a few instructions and s_a is no longer than s_b, we exclude s_a. It is reasonable to expect that s_b will preserve s_a 's characteristics.

IV. DETERMINISTIC ROUTING

A distilled instruction sequence may be a sequence of random instructions or a fragment of a program in machine language. In this section, we propose three schemes to differentiate these two cases:

- ***Scheme 1***

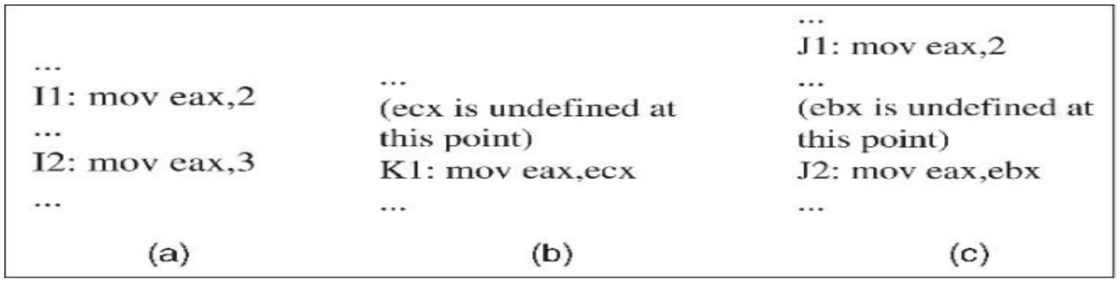
exploits the operating system characteristics of a program; Scheme 2 and Scheme 3 exploit the data flow characteristics of a program. Scheme 1 is slightly faster than Scheme 2 and Scheme 3, whereas Scheme 2 and Scheme 3 are much more robust to obfuscation. Scheme 1 is fast since it does not need to fully disassemble,

For most instructions, we only need to know their types. This saves a lot of time in decoding operands of instructions.

Note that although Scheme 1 is good at detecting most of the known buffer overflow attacks, it is vulnerable to obfuscation.

- ***Scheme 2***

Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may easily obfuscate his program by introducing enough data flow anomalies. In this paper, we use the detection of data flow anomaly in a different way called code abstraction. We observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path has a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.



```
...
I1: mov eax,2
...
I2: mov eax,3
...
```

```
...
(ecx is undefined at
this point)
K1: mov eax,ecx
...
```

```
...
J1: mov eax,2
...
(ebx is undefined at
this point)
J2: mov eax,ebx
...
```

Fig.2. Data flow anomaly in execution paths. (a) Define-define anomaly. Register eax is defined at I1 and then defined again at I2. (b) Undefine-reference anomaly. Register ecx is undefined before K1 and referenced at K1. (c) Define-undefine anomaly. Register eax is defined at J1 and then undefined at J2.

Pruning useless instructions. Next, we leverage the detected data flow anomalies to remove useless instruc-tions. A useless instruction of an execution path is an instruction that does not affect the results of the execution path; otherwise, it is called useful instructions. We may find a useless instruction from a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction that causes the “reference” is a useless instruc-tion. For instance, the instruction K1 in Fig. 2, which causes undefine-reference anomaly, is a useless instruction. When there is a define-define or define-undefine anomaly, the instruction that caused the former “define” is also con-sidered as a useless instruction. For instance, the instruc-tions I1 and J1 in Fig. 2 are useless instructions because they caused the former “define” in either the define-define or the define-undefine anomaly.

After pruning the useless instructions from an execution path, we will get a set of useful instructions. If the number of useful instructions in an execution path exceeds a threshold, we will conclude the instruction sequence is a segment of a program. Algorithm 2 shows our algorithm to check if the number of useful instructions in an execution path exceeds a threshold. The algorithm involves a search over an EISG in which the nodes are visited in a specific order derived from a depth first search. The algorithm assumes that an EISG G and the entry instruction of the instruction sequence are given, and a push down stack is available for storage. During the search process, the visited node (instruction) is abstractly executed to update the states of variables, find data flow anomaly, and prune useless instructions in an execution path.

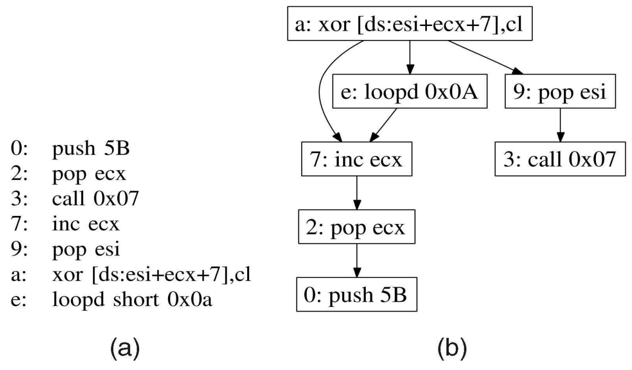
Algorithm 2 check if the number of useful instructions in an execution path exceeds a threshold

```
Input: entry instruction of an instruction sequence, EISG G
total   0; useless   0; stack   empty
initialize the states of all variables to “undefined” push the entry instruction, states, total, and useless to stack
while stack is not empty do
 pop the top item of stack to i, states, total, and useless if total _ useless greater than a threshold then
return true
if i is visited then
continue (passes control to the next iteration of the WHILE loop)
mark i visited total total þ 1
Abstractly execute instruction i (change the states of variables according to instruction i)
if there is a define-define or define-undefine anomaly then
useless             useless þ 1
   if there is an undefine-reference anomaly then useless useless þ 1
   for each instruction j directly following i in the G do push j, states, total, and useless to stack
return false
```

Handling special cases. Next, we discuss several special cases in the implementation of Scheme 2.

General purpose instruction. The instructions in the IA-32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instruc-tions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors [50]. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we consider other groups of instructions as useless instructions.

Initial state of registers. For registers, we set their initial states to “undefined” at the beginning of an execution path.



```
0:   push 5B
2:   pop ecx
3:   call 0x07
7:   inc ecx
9:   pop esi
a:   xor [ds:esi+ecx+7],cl
e:   loopd short 0x0a
```

Fig. 3. (a) A decryption routine that only has seven useful instructions.(b) A def-use graph of the decryption routine. Instruction a can reach all other instructions through paths a ! e ! 7 ! 2 ! 0 and a ! 9 ! 3, and therefore, the dependence degree of instruction a is 6.

The register “esp,” however, is an exception since it is used to hold the stack pointer. Thus, we set register esp “defined” at the beginning of an execution path.

Indirect address. An indirect address is an address that serves as a reference point instead of an address to the direct memory location. For example, in the instruction “move eax,[ebx+01e8],” register “ebx” may contain the actual address of the operand. However, it is difficult to know the runtime value of register “ebx.” Thus, we always treat a memory location to which an indirect address points as state “defined” and hence no data flow anomaly will be generated. Indeed, this treatment successfully prevents an attacker from obfuscating his code using indirect addresses.

Useless control transfer instructions (CTIs). A CTI is useless if the corresponding control condition is undefined or the control transfer target is undefined. The status flags of the EFLAGS register are used as control condition in the IA-32 architecture. These status flags indicate the results of arithmetic and shift instructions, such as the ADD, SUB, and SHL instructions. Condition instructions Jcc (jump on condition code cc) and LOOPcc use one or more of the status flags as condition codes and test them for branch or end-loop conditions. During a program execution at runtime, an instruction may affect a status flag on three different ways: set, unset, or undefine [50]. We consider both set and unset are defined in code abstraction. As a result, a status flag could be in one of the two possible states—defined or undefined in code abstraction. To detect and prune useless CTIs, we also assume that the initial states of all status flags are undefined. The states of status flags are updated during the process of code abstraction.

The objective in selecting the threshold is to achieve both low false positives and low false negatives. Our experi-mental results in Section 6 show that we can achieve this objective over our test data sets. However, it is possible that attackers evade the detection by using specially crafted code, if they know our threshold. For example, Fig. 7a shows that a decryption routine has only seven useful instructions, which is less than our threshold (15 to 17 in our experiments) of Scheme 2.

➢ ***Scheme 3***

Unlike Scheme 2, which compares the number of useful instructions with a threshold, Scheme 3 first calculates the dependence degree of every instruction in the instruction sequence. If the dependence degree of any useful instructions in an instruction sequence exceeds a threshold, we conclude that the instruction sequence is a segment of a program. Dependency is a binary relation over instructions in an instruction sequence. We say instruction j depends on instruction i if instruction i produces a result directly or indirectly used by instruction j. Dependency relation is transitive, that is, if i depends on j and j depends on k, then i depends on k. For example, instruction 2 directly depends on instruction 0 in Fig. 7a and instruction 7 directly depends on instruction 2, and by transitive property instruction 7 depends on instruction 0. We call the number of instructions, which an instruction depends on, the dependence degree of the instruction. To calculate the dependence degree of an instruction, we construct a def-use graph. A def-use graph is a directed graph G ¼ ðV ; EÞ where each node v 2 V corresponds to an instruction and each edge e ¼ ðv_i; v_jÞ 2 E indicates that instruction v_j produces a result directly used by instruction v_i. Obviously, the number of instructions that an instruction can reach through any path in the def-use graph is the dependence degree of the instruction.

Parameter Tuning:

All three schemes use a threshold value to determine if a request contains code or not. 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 these schemes against 50 unencrypted attack requests generated by Metasploit framework,

This allows us to obtain a better threshold, which can be used to protect not only web servers but also other Internet services. Note that although worm Slammer attacks Microsoft SQL servers instead of web servers, it also exploits buffer overflow vulnerabilities.

➢ Threshold of push-calls for Scheme 1. Fig. 3a shows that all instruction sequences distilled from a normal request contain at most one push-call code pattern. Fig. 3b shows that for all the 53 buffer overflow attacks we tested, every attack request contains more than two push-calls in one of its instruction sequences. Therefore, by setting the threshold number of push-calls to 2, Scheme 1 can detect all the attacks used in our experiment.

➢ Threshold of useful instructions for Scheme 2. Fig. 3c shows that no normal requests contain an instruction sequence that has more than 14 useful instructions. Fig. 3d shows that an attack request contains over 18 useful instructions in one of its instruction sequences. Therefore, by setting the threshold to a number between 15 and 17, Scheme 2 can detect all the attacks used in our test. The three attacks, which have the largest numbers of instructions (92, 407, and 517), are worm Slammer, CodeRed.a, and CodeRed.c, respectively. This motivates us to investigate in our future work whether an exceptional large number of useful instructions indicate the occurrence of a worm.

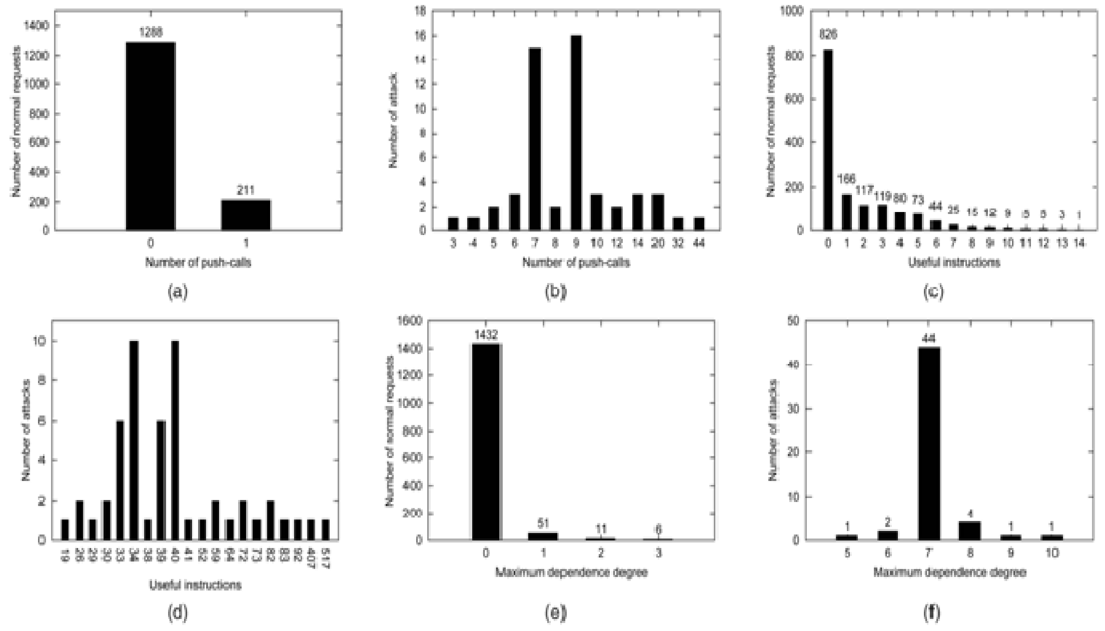


Fig. 3. (a) The number of push-calls in normal requests. (b) The number of push-calls in attack requests. (c) The number of useful instructions in normal requests. (d) The number of useful instructions in attack requests. (e) Maximum dependence degree in normal requests. (f) Maximum dependence degree in attack requests in a request.

- Threshold of dependence degree for Scheme 3. Fig. 3e shows that no normal requests contain an instruction sequence whose dependence degree is more than 3. Fig. 3f shows that for each attack there exists an instruction sequence whose dependence degree is more than 4. There-fore, by setting the threshold to 4 or 5

V. LIMITATIONS

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. With respect to SigFree, due to the obscurity of the flow of control, branch function may cause SigFree to break the executable codes into multiple instruction sequences. Nevertheless, it is still possible for SigFree to find this type of buffer overflow attacks as long as SigFree can still find enough number of useful instructions or dependence degree in one of the distilled instruction sequences.

- Second, SigFree cannot fully handle self-modifying code. Self-modifying code is a piece of code that dynamically modifies itself at runtime and could make SigFree mistakenly exclude all its instruction sequences. It is crafted in a way such that static analysis will reach illegal instructions in all its instruction sequences, but these instructions become legal during execution. To address this attack, we may remove Step 3 in excluding instruction sequences; that is, we do not use inevitably reachable illegal instructions as a criterion for pruning instruction sequences. This however will increase the computational overhead as more instruction sequences will be analyzed. Self-modifying code can also reduce the number of useful instructions, because some instructions are considered useless. We note that there are also no general static solutions for handling self-modifying code at the present state of the art. Nevertheless, it is still possible for SigFree to detect self-modifying code, because self-modifying code itself is a piece of code that may have enough number of useful instructions or dependence degree. Our future work will explore this area.

- Third, the executable shellcodes could be written in alphanumeric form [53]. Such shellcodes 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 shellcodes; however, it will increase computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

- Fourth, SigFree 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 (through mmap()) the addresses of shared libraries so that the addresses contain null bytes.

- Finally, it is still possible that attackers evade the detection of Scheme 3 by using specially crafted code, once they know the threshold of Scheme 3. However, we believe it is fairly hard, as the bar has been raised. For example, it is difficult, if not impossible, to reduce the dependence degree to 3 or fewer in all the instructions of Countdown decryption routine

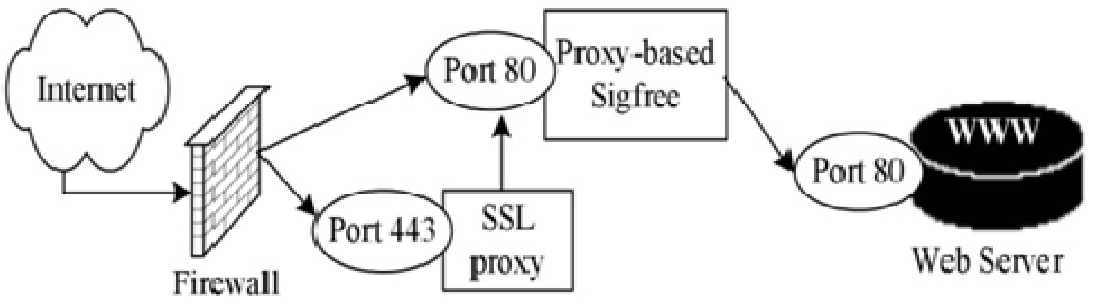


Fig. 4. SigFree with an SSL proxy

VI. Application-Specific Encryption Handling

The proxy-based SigFree 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 (Fig. 4) may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install SigFree 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 (e.g., modssl in Apache). In this case, SigFree will need to be implemented as a server module (though not shown in Fig. 4), 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.

VII. Applicability

So far, we only discussed using SigFree to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to buffer overflow attacks. For example, the proxy-based SigFree can be used to protect all internet services that do not permit executable binaries to be carried in requests. SigFree should not directly be used to protect some Internet services that do accept binary code such as

FTP servers; otherwise, SigFree will generate many false positives. To apply SigFree for protecting these Internet services, other mechanisms such as whitelisting need to be used.

In addition to protecting servers, SigFree 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 JPEG Processing (GDI+), and WinAmp. This means that attackers may embed their malicious code in PDF, JPEG, or MP3-list files to launch buffer overflow attacks. In fact, a virus called Hesive was disguised as a Microsoft Access file to exploit buffer overflow vulnerability of Microsoft's Jet Database Engine.

Once opened in Access, infected .mdb files take advantage of the buffer overflow vulnerability to seize control of vulnerable machines. If mass-mailing worms exploit these kinds of vulnerabilities, they will become more fraudulent than before, because they may appear as pure data-file attachments.

SigFree can be used to alleviate these problems by checking those files and email attachments that should not include any code. If the buffer being overflowed is inside a JPEG or GIF system, ASN.1 or base64 encoder, SigFree cannot be directly applied. Although SigFree can decode the protected file according to the protocols or applications it protects, more details need to be studied in the future.

Although SigFree is implemented in the Intel IA-Therefore, we believe that detection capabilities and resilience to obfuscation will be preserved after porting. Of course, some implementation details such as handling special cases in Scheme 2 need to be changed. We will study this portability issue in our future work. Finally, as a generic technique, SigFree can also block other types of attacks as long as the attacks perform binary code injection. For example, it can block code-injection format string attacks.

VIII. CONCLUSION

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

REFERENCES

[1] Intel IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture Intel, http://developer.intel.com/design/pentium4/manuals/253665.htm, 2007.

[2] Citeseer: Scientific Literature Digital Library, http://citeseer.ist.psu.edu, 2007.

[3] T. Berners-Lee, L. Masinter, and Sigfree: A Signature-Free Buffer Overflow AttackBlocker.XinranWang,Chi-chun pan,peng,Liu,SencunZhu(2012)

[4] Dynamic Taint Analysis for Automatic Detection,Analysis and signature software J. Huang,

[5] "Detection of Data Flow Anomaly through Program Instrumentation," IEEE Trans. Software Eng., . M. McCahill, Uniform Resource Locators (URL), RFC 1738 (Proposed Standard),updated by RFCs 1808, 2368,2396, 3986, http://www.ietf.org/rfc/rfc1738.txt,2007 G.S. Kc and A.D. Keromytis, "E-NEXSH: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," Proc.21st

[6] 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. J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," IEEE Security and Privacy, vol. 2, no. 4, 2004.

[7] 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.

[8] 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.

[9] 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.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," Proc. 20th ACM Symp. Operating Systems Principles (SOSP), 2005.

[11] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[12] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[13] S. Singh, C. Estan, G. Varghese, and S. Savage, "The Earlybird System for Real-Time Detection of Unknown

Worms," technical report, Univ. of California, San Diego, 2003.
[14] H.-A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," Proc. 13th USENIX Security Symp. (Security), 2004.
[15] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatic Signature Generation for Polymorphic Worms," Proc. IEEE Symp. Security and Privacy (S&P), 2005.
[16] R. Chinchani and E.V.D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2005.
[17] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," Proc. Fifth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2002.
[18] 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.
[19] The Metasploit Project, http://www.metasploit.com, 2007.
[20] Jempiscodes—A Polymorphic Shellcode Generator, http:// www.shellcode.com.ar/en/proyectos.html, 2007.
[21] S. Macaulay, Admmutate: Polymorphic Shellcode Engine, http:// www.ktwo.ca/security.html, 2007.
[22] T. Detristan, T. Ulenspiegel, Y. Malcom, and M.S.V. Underduk, Polymorphic Shellcode Engine Using Spectrum Analysis, http:// www.phrack.org/show.php?p=61&a=9, 2007.
[23] 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.
[24] Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," IEEE Software, vol. 19, no. 1, 2002.