INTRODUCTION

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 bonnets. 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 [1]. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets), I code-injection buffer overflow attacks such as stack smashing probably count for most of the buffer overflow attacks that have happened in the real world. Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly desired requirements: (R1) simplicity in maintenance; (R2) transparency to existing (legacy) server OS, application software, and hardware; (R3) resiliency to obfuscation; (R4) economical Internet-wide deployment. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to occur 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 [3], [4], (1F) Capturing code running symptoms of buffer overflow attacks [5], [6], [7], [8]. (Note that the above list does not include binary-code-analysis-based defenses, which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows: 1) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardware, thus they are not transparent. Moreover, Class 1E defenses generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination headroom as economical deployment. 2) Class 1F defenses can be very secure, but they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. As a result, Class 1F defenses have limited transparency and potential for economical deployment. 3) Class 1A defenses need source code, but source code is unavailable to many legacy applications. Besides buffer overflow defenses, worm signatures can be generated and used to block buffer overflow attack packets [9], [10], [11]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

MATERIALS AND METHODS

Experiments

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” [12]. In particular, as summarized in [12], 1) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434), which are used to monitor Microsoft SQL Databases, accept data only. 2) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306, and 5432 accept data only. Since remote exploits are typically binary executable code, this observation indicates that if we can precisely distinguish (service requesting) messages containing binary code from those containing no binary code, we can protect most Internet services (which accept data only) from code injection buffer overflow attacks by blocking the messages that contain binary code. Accordingly, 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 (if any). However, in this phase, some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree uses a novel technique called code abstraction. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions (Scheme 2) or dependence degree (Scheme 3) to a threshold to determine if this instruction sequence (distilled in phase 1) contains code. Unlike the existing code detection algorithms [12], [13], [14] that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade (Section 2 gives a more detailed comparison).

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

- **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.

**Sigfree Overview**

**Basic Definitions and Notations**

This section provides the definitions that will be used in the rest of this paper. Definition 1 (instruction sequence). An instruction sequence is a sequence of CPU instructions, which has one and only one entry instruction and there exists at least one execution path from the entry instruction to any other instruction in this sequence. A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Fig. 2 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as si in Fig. 2, where is the entry location of the instruction sequence in the string. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below, we call them random instruction sequences, whereas use the term binary executable code to refer to a fragment of a real program in machine language. Definition 2 (instruction flow graph). An instruction flow graph (IFG) is a directed graph G ¼ δV; EP where each node v 2 V corresponds to an instruction and each edge e 2 δV; vjP 2 E corresponds to a possible transfer of control from instruction vi to instruction vj. Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model the control flow of an instruction sequence, we further extend the above definition.

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

**Fig. 2** shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as si in Fig. 2, where is the entry location of the instruction sequence in the string. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below, we call them random instruction sequences, whereas use the term binary executable code to refer to a fragment of a real program in machine language. Definition 2 (instruction flow graph). An instruction flow graph (IFG) is a directed graph G ¼ δV; EP where each node v 2 V corresponds to an instruction and each edge e 2 δV; vjP 2 E corresponds to a possible transfer of control from instruction vi to instruction vj.

**Fig. 3** Data structure for the instruction sequences distilled from the request in Fig. 2. (a) EIFG. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

Definition 3 (extended IFG). An extended IFG (EIFG) is a directed graph G ¼ δV; EP, which satisfies the following properties: each node v 2 V corresponds to an instruction, an illegal instruction (an “instruction” that cannot be recognized by CPU), or an external address (a location that is beyond the address scope of all instructions in this graph); each edge e ¼
accordingly, we name the types of nodes in an EIGF instruction node, illegal instruction node, and external address node. The reason that we define IIFG and EIFG is to model two special cases, which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s08 in Fig. 2, the transfer of control is from instruction “lods [ds:esi]” to an illegal instruction at address 0F. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s00 in Fig. 2 has an instruction “jmp ADAAC3C2,” which jumps to external address ADAAC3C2.

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 <4 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\[node i; node t\] to G end

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