STILL: Exploit Code Detection via Static Taint and Initialization Analyses

[1]Xinran Wang [1]Yoon-Chan Jhi [1,2]Sencun Zhu [2]Peng Liu
[1]Department of Computer Science and Engineering
[2]College of Information Sciences and Technology
Pennsylvania State University, University Park, PA 16802
{xinrwang, szhu, jhi}@cse.psu.edu; pliu@ist.psu.edu

Abstract

We propose STILL*, a generic defense based on* S*tatic* T*aint and* I*nitia*L*ization ana*L*yses, to detect exploit code embedded in data streams/requests targeting at various Internet services such as Web services. STILL first blindly disassembles each request, generates a (probably partial) control flow graph, and then uses novel static taint and initialization analysis algorithms to determine if strong evidence of self-modifying (including polymorphism) and/or indirect jump code obfuscation behavior can be collected. If such evidence exists, STILL will raise an alarm and block the request; otherwise, STILL will perform another form of static taint analysis to check whether unobfuscated or other types of obfuscated exploit code (e.g., metamorphism, etc) is embedded in the request. To the best of our knowledge, compared with existing static analysis approaches developed for the same purpose, STILL is (a) the first one that can detect self-modifying code and indirect jump, and (b) a more comprehensive static analysis solution in defending against anti-signature, anti-static-analysis and anti-emulation code obfuscation (for all the code obfuscation techniques we are aware of, STILL is robust to all but one).*

1 Introduction

Besides triggering human beings to do such unwise things as opening malicious email attachments, disclosing their passwords and identities, and visiting fake web sites, human-intervention-free remote exploit code is a primary vehicle for attackers to harvest bots and launch various attacks. The most serious vulnerability exploited by binary exploit code is buffer overflow, which exists pervasively in Internet services (e.g., port 80 Web service), OS services (e.g., Windows DCE-RPC or CIFS/SMB), database services, applications (e.g., browser plug-ins), and so on. Buffer overflow vulnerabilities allow attackers to use a network request to inject a piece of exploit code into the "body" of a service or application program. Once such exploit code is executed, the attacker may gain full control of the victim machine. In different attacks, exploit code may be in different forms: a piece of shellcode to break into a certain type of hosts, an infection vector for Internet worms such as CodeRed and Slammer, and so on. From both CERT [1] and Microsoft Security Bulletin [5] we can clearly see that the majority of vulnerabilities/attacks in the Microsoft Windows family are buffer-overflow based automatic remote code execution.

Network intrusion detection systems (NIDSes) such as Snort [6] use manually prepared string-matching signatures to detect code-injection attacks. To contain zero-day attacks, automatic signature generation techniques have recently been proposed to automatically extract string-matching signatures from malicious payloads. Some representative examples of these techniques are EarlyBird [24], Polygraph [21], Hamsa [17], and Packet Vaccine [29]. However, a common limitation of string-matching signature based defenses, whether their signatures are manually prepared or automatically generated, is that they (e.g., [24, 27]) or their flow classifiers (e.g., [17]) are not very resilient to code-obfuscation.

To address the limitations of the signature-based defense mentioned above, researchers have recently proposed several static analysis approaches to identify and analyze the code contained in buffer overflow attack packets [8, 15, 28]. These approaches are based on the observation that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services such as Web services and SQL services. These static analysis approaches can be divided into two stages. The first stage is to distill instruction sequences from network packets by disassembly. These instruction sequences may be real code or random instructions. The second stage is to analyze the disassembly results by exploiting control flow [15] or data flow analysis [8, 28] to discern code/malicious code from data. Compared with string-matching signature based approaches, static analysis approaches have two notable advantages. First, they can

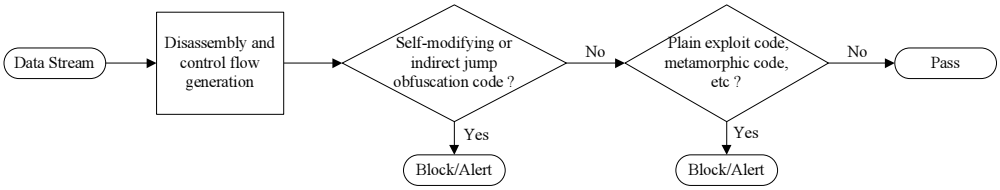

Figure 1. The architecture of STILL

detect new (or zero-day) exploit code for both known and unknown vulnerabilities. Second, they are more resilient to polymorphism and metamorphism.

However, by applying some anti-static-analysis obfuscation techniques such as self-modifying and indirect jump [18, 23], an attacker can evade these static analysis approaches such as SigFree [28] and [8, 15]. There are two main techniques to thwart static analysis [23]. One aims at thwarting disassembly via indirect jump. An indirect jump instruction transfers control to the address contained in a register operand and its destination is hard to be statically determined. The other technique is to thwart control flow and data flow analyses by self-modifying code. Control transfer instructions (CTIs) can be dynamically changed to non-CTIs at runtime by self-modifying code, thus obfuscating control flows. Data transfer/operation instruction can be changed to other instructions dynamically at runtime by self-modifying code, thus obfuscating data flows.

Our Approach We propose *STILL, a novel static taint and initialization analysis based approach, to detect not only unobfuscated exploit code (without any obfuscation), traditional polymorphic and metamorphic exploit code, but also self-modifying and indirect jump obfuscation code*. STILL is based on the same observation as in [8, 15, 28] that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services.

STILL detects attacks as follows. It works as a proxy-based blocker in the application layer of clients and/or servers. When it captures a data stream, it disassembles the data stream and generates a control flow graph. It analyzes the disassembled result in two stages. First, STILL detects self-modifying and indirect jump obfuscation code. Although the real exploit code may be hidden by self-modifying and indirect jump, the obfuscation code itself provides some strong evidence of self-modifying and/or indirect jump behavior. STILL detects this behavior by static taint analysis and initialization analysis. Since polymorphism is a kind of self-modifying, STILL can also detect polymorphic code in this stage. Of course, an attacker might use neither self-modifying nor indirect jump obfuscation. Hence, in the second stage, STILL tries to detect the plain exploit code, which may even have been obfuscated by metamorphism. STILL also exploits static analysis and initialization analysis in this stage to combat other obfuscation techniques. Figure 1 depicts the architecture of STILL.

Contributions The main merits of STILL are as follows. First, STILL (including *static taint analysis* and *initialization analysis*) is a static analysis technology that can detect both self-modifying code and indirect jump, whereas previous static analysis approaches [8,15,28] could be easily thwarted by these obfuscation techniques. Second, STILL is robust to almost all anti-signature, anti-static-analysis and anti-emulation obfuscation. Third, STILL is a more comprehensive solution for detecting zero-day exploit code, regardless of whether the exploit code is obfuscated or not. Note that although the proposed techniques, like many intrusion detection systems, somehow look heuristic, we believe they are very fundamental and greatly raise the bar for future attacks.

A prototype implementation of STILL has been developed. Experimental results show that STILL successfully detected all 12,000 polymorphic shellcode payloads generated by 10 well-known polymorphic engines, all 34 plain Linux and Windows shellcodes generated by the Metasploit framework, and 5 worms (CodeRed I, CodeRed II, Sasser, Blaster, and Slammer) that are available to us. Moreover, our large-scale testing shows that STILL has an extremely low false positive rate.

Organization The rest of the paper is organized as follows. In Section 2, we give an overview of exploit code obfuscation. In Section 3, we show how we disassemble a data stream and then generate a control flow graph (3.1), the approach to detecting self-modifying and indirect jump obfuscation code (3.2), and the approach to detecting plain exploit code (3.3). We show the strength of STILL and discuss the limitations in Section 4. In Section 5, we show our experimental results. Finally, we summarize the work related to ours in Section 6 and conclude the paper in Section 7.

2 Obfuscation of Exploit Code

In this section, we give a brief overview of the known techniques for obfuscating exploit code. We classify these obfuscation techniques into three categories according to their purposes: anti-signature, anti-static-analysis and anti-emulation. We show how anti-static-analysis thwarts pre-

vious static analysis techniques such as SigFree [28] and others [8, 15]. We will discuss the effectiveness of STILL in handling these obfuscation techniques in Section 4.

2.1 Anti-signature

Anti-signature obfuscation is the type of obfuscation techniques for thwarting traditional string-matching signature based detection. It includes metamorphism and polymorphism. Metamorphism is the type of obfuscation techniques which evade signature-based detection by instruction reordering, register renaming, garbage insertion, instruction replacement and equivalent functionality. More details of these techniques can be found in [10, 20].

Polymorphism is a type of obfuscation techniques which normally rely on encryption to mutate code. Original code is encrypted and hidden in the payload, and it is decrypted to its original form during execution. However, a small portion of it, the decryption routine is left unencrypted. Some polymorphic engines further rewrite the unencrypted decryption routine by metamorphism each time the exploit code is propagated [12, 19].

2.2 Anti-static-analysis

Anti-static-analysis is the type of obfuscation techniques for thwarting static analysis. It can be divided into three subcategories: anti-disassembly, self-modifying and memory access obfuscation. Note that some anti-static-analysis techniques such as self-modifying and indirect jump can easily thwart previous static analysis techniques [8, 15, 28]. Next, we describe these techniques in slightly more details.

Self-Modifying Self-modifying code is code that modifies itself when being executed. Self-modifying is a very powerful technique to thwart static analysis since it can completely hide the semantics of original instructions. More specifically, it may be used to thwart static analysis in a number of ways. First, it can be used to obfuscate the control flow graph (CFG) of exploit code, while CFG is the obligatory input to a static analysis approach. As an example, in Figure 2(a) the original loop instruction at address 16 is replaced in Figure 2(b) by a nop instruction at $1d$ and an invalid instruction at $1e$, and these two instructions will be modified back to the loop instruction at runtime by instruction $0c$ in Figure 2(b). Second, it can be used to obfuscate data flow of exploit code to thwart those data flow analysis based approaches such as SigFree [28] and [8]. For example, a data transfer instruction can be changed to a nop instruction dynamically at runtime by self-modifying code, thus obfuscating data flows. Note that polymorphic code is also a kind of self-modifying code, which hides original exploit code. For example, in Figure 2(a) the encrypted payload (encrypted original exploit code) will be decrypted at runtime and then be executed.

```
00:  2BC9              sub ecx,ecx
02:  83E9 B0           sub ecx,-0x50
05:  E8 FFFFFFFF       call 0x09
09:  FFC0              inc eax
0b:  5E                pop esi
0c:  8176 0E DC40D776  xor [esi+0xE],0x76D740DC
13:  83EE FC           sub esi,-0x4
16:  E2 F4             loopd 0x0C
18:  (Encrypted payload)
     ...
```

(a)

```
00:  2BC9              sub ecx,ecx
02:  83E9 B0           sub ecx,-0x50
05:  E8 FFFFFFFF       call 0x09
09:  FFC0              inc eax
0b:  5E                pop esi
0c:  668176 0A 7264    xor [esi+0x13],0x0A72
12:  8176 0E DC40D776  xor [esi+0x15],0x76D740DC
19:  83EE FC           sub esi,-0x4
1d:  90                nop
1e:  FE                (invalid instruction)
1f:  (Encrypted payload)
     ...
```

(b)

Figure 2. (a) A decryption routine generated by Engine Pex [4]. (b) The decryption routine obfuscated by self-modifying which confuses control flows.

Anti-disassembly is the type of obfuscation techniques that try to confuse traditional disassembly algorithms (e.g., linear sweep [18]). It includes junk byte insertion, opaque predicate, code overlap, indirect jump and branch function.

Indirect jump transfers control to the instruction whose address is in a register operand. Because the value of the register may not be statically determined, disassembly algorithm such as recursive traversal cannot provide an accurate disassembly. Therefore, attackers may use indirect jump to replace relative jump in the payload. For example, relative jump instruction "jmp 0x05" can be replaced by indirect jump instruction "jmp eax", where eax contains absolute address of the target. The value of eax is normally hard to determine until at run-time, thus thwarting static analysis.

Branch function is a function $f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$ [18]. By replacing unconditional branches in a program with calls to a branch function, attackers can obscure the flow of control in the program.

Memory Access Obfuscation is the type of obfuscation techniques that use indirect addressing for memory access to thwart static analysis. For example, instruction "mov ebx,[ss:esp]", which moves the top item of stack into ebx, may be obfuscated by instructions " mov eax,esp; mov ebx,[ss:eax]".

2.3 Anti-emulation

There are many anti-emulation techniques, such as using interrupts in polymorphic decryption routines, inserting de-

lay loops, executing random code. These techniques have existed in virus writer community for many years and more details can be found in [26].

3 A Generic Code Detection Technique

In this section, we present STILL in three steps, as shown in Fig. 1.

3.1 Disassembly and Control Flow Graph Generation

The first step of STILL is to disassemble the input data stream and generate a control flow graph. A major challenge here is that we do not know whether the data stream contains code or not, and what the entry point of the code is when code is present. As such, it is not directly clear which parts of a stream should be disassembled. The problem is exacerbated by the fact that different types of instructions have varying lengths and most bit combinations map to valid instructions in the Intel IA32 instruction set. In fact, even a stream of random bytes (or a part of a stream) could be disassembled into a valid instruction sequence [15, 28].

In previous work [8, 15, 28] several disassembly algorithms have been proposed to address the aforementioned challenges. In STILL, we exploit the O(N) disassembly algorithm used in [28], where N is the length of the data stream. This algorithm will result in a set of instruction sequences. 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 (random) instruction sequences from any binary strings (e.g., a GIF file). Two example instruction sequences are shown in Figure 2: sequence 1 includes instructions 00 to 16 in Figure 2(a); sequence 2 includes instructions 00 to $1d$ in Figure 2(b).

STILL is more robust to anti-disassembly techniques than previous work [8, 15, 28]. For example, previous work exploits some heuristics to prune basic blocks. In [15], a basic block is considered invalid in three cases: (1) if it contains one or more invalid instructions; (2) if it is on a path to an invalid block; (3) if it ends in a control transfer instruction that jumps into the middle of another instruction. In [28], some similar heuristics are applied. STILL does not use the first two heuristics because invalid instructions could be the result of self-modifying obfuscation. For example, the basic block (instructions from 00 to $1e$ in Figure 2), which ends with an invalid instruction, should not be pruned. STILL does not use the third heuristic either because a control transfer instruction may jump into the middle of another instruction by using code overlap obfuscation [23].

We note that in the presence of indirect jump and self-modifying obfuscation, it is impossible to completely and statically disassemble the entire body of the exploit code embedded in a data stream using the recursive traversal algorithm. Fortunately, the partially disassembled result may already provide some strong evidence of self-modifying and/or indirect jump behavior. In Figure 2(b), neither the decryption routine (e.g., the loop instruction can no longer be seen in the disassembly result) nor the original exploit shellcode can be successfully disassembled. However, the instructions from 00 to $1d$ indicate the self-modifying behavior. Our approach to detecting self-modifying and indirect jump code in the next section will only use this partially disassembled result as the input.

3.2 Detection of Self-modifying and Indirect Jump Obfuscation Code

Once the (partial) instruction sequences have been extracted in the first step (i.e., the disassembly process), the next step is to determine whether they are real exploit code. As we showed in Section 2, there are many techniques for obfuscating shellcode. In this section, however, our discussion will concentrate on detecting self-modifying and indirect jump exploit code, because these two types of exploit code can evade the detection of previous static analysis schemes [8, 15, 28] and are very challenging to detect. Clearly, if STILL can detect these two types of exploit code, it will also be capable of detecting branch function and polymorphic code because branch function uses an indirect jump to transfer control to the original target and polymorphic code is a kind of self-modifying code. In Section 4 we will show that STILL is rather effective in handling the other types of obfuscation.

The new techniques in STILL to detecting self-modifying and indirect jump exploit code are called *static taint analysis* and *initialization analysis*. We observe that self-modifying and indirect jump exploit codes first need acquire the absolute address of payload. Then, the absolute address will be used in a certain way (referred to as *abstract semantics*) reflecting self-modifying and indirect jump behavior, whereas this behavior is very rare in random instruction sequences. Accordingly, self-modifying and indirect jump exploit codes are detected as follows. First, the variable which holds the absolute address of the payload is found in the instruction sequences and used as a *taint seed*. Then, static taint analysis is used to track the tainted values and detect whether tainted data are used in the abstract semantics that could indicate the presence of self-modifying and indirect jump exploit code. Finally, we use initialization analysis to reduce false positives.

Taint Seed The absolute address of payload is used as a taint seed in STILL. Both indirect jump and self-modifying obfuscation code need the absolute address of payload, because the target for indirect jump and memory read/write for self-modifying must use absolute addresses in IA-32 architecture [2]. They have to know their own absolute address in order to jump within the payload or modify the payload at runtime. There are two reasons that attackers want to get the absolute address of payload at runtime rather than predicting it or hardcoding it. First, the address of payload cannot be predicted in most cases because exploit code is placed in a dynamically changing stack or heap [23]. Second, even if attackers can sometimes get the address, it is not good practice to hardcode it, especially in the case of a worm [25]. The absolute address of payload could be different for different versions and different patches of the same operating system (e.g. Windows 2000 with Service Patch 1, Patch 2 and Windows XP Service Patch 1); hence, hardcoding the absolute address could greatly limit the broad spread of a worm.

The only way to get the absolute address of payload is to read the PC (Program Counter) register, which stores the absolute address of the next instruction to be executed. Since the IA-32 architecture does not provide any instruction to directly read PC, attackers have to acquire it at runtime. To the best of our knowledge, currently only three ways (called getPC) are used in the attacker community. First, attackers may use a relative call. Whenever a call is executed, the return address is pushed into the stack just before the control is transferred to the call target. Therefore, the top item of the stack is used as the taint seed at the relative call target. Second, the attackers can also get the address by using the *fstenv* instruction, which saves the current FPU (Floating Point Unit) operating environment at the memory location specified by its operand [22]. The FPU operating environment includes the instruction pointer of the last executed float point instruction. Hence, we can also find the taint seed by checking a float point instruction and a succeeding fstenv instruction. Finally, attackers may also get the address by using the structured exception handling (SEH) mechanism of Windows [13]. However, as mentioned in [23], this technique is feasible only with older versions of Windows. In this paper, we do not consider this case. Note that whenever a new way for getting PC (if exist) is found, we can easily add a corresponding method in STILL to find the taint seed while the rest parts of STILL will not be affected by this update.

Static Taint Analysis After the taint seed is found, a new *static taint analysis* approach is used to statically determine which variables are tainted in an instruction sequence. A taint seed itself is a tainted variable.

A tainted variable is propagated to a new tainted variable by data transfer instructions that move data (e.g., push, pop, move) and data operation instructions that perform arithmetic or bit-logic operations on data (e.g., add, sub, xor) in the IA-32 instruction set. Other instructions such as control transfer instructions do not affect the taint process. For data transfer instructions, the destination operand will be tainted if and only if the source operand is tainted. For data operation instructions, the destination operand will be tainted if and only if either the source or the destination operand is tainted. Note that for data movement and arithmetic instructions, constants are considered untainted.

Detection by Abstract Semantics Certain abstract semantics can be observed from the tainted data of a real instruction sequence, whereas these abstract semantics are very rare in random instruction sequences. Hence, if these abstract semantics are detected through static taint analysis, an alert will be raised. The following are the abstract semantics for self-modifying and indirect jump, respectively.

Self-modifying Self-modifying obfuscation works in three steps. First, it reads the payload; second, it modifies the read result; finally, it writes the modified result back to the payload. Attackers may implement these three steps in two ways. One way is to use a single updating instruction in the payload to implement these three steps. Instruction $0c$ in Figure 2(b) is such an example. The other way uses several instructions, one instruction for reading payload, several instructions for modifying the read results and one instruction for writing the modified results back to payload. The CLET [12] shellcode generation engine uses this approach. Accordingly, there are two cases where tainted data indicate self-modifying obfuscation. First, the tainted data are used as the address of the updating instructions. Second, the tainted data are used as the address of a memory read instruction or the address of a memory write instruction. We note that the read result will be used to generate the write result; therefore, we start a new taint analysis process to taint the read result. If the newly tainted data are used as the source operand of a memory write instruction, it clearly indicates self-modifying obfuscation. Figure 3 shows these two ways of identifying self-modifying code through static taint analysis.

Indirect jump To detect indirect jump obfuscation, we check whether tainted data are used as target addresses of control transfer instructions such as branch, return, and function call instructions. Normally, it is rare that tainted data are used as jump targets in random instruction sequences.

Reducing False Positives Although for random instruction sequences it is not common that the tainted data are used in the same way as the way they are used by self-modifying and indirect jump, we still find some false positives in our experiments. To reduce false positives, we further use *initialization analysis*. We observed that the operands of self-modifying and indirect jump code must be initialized.

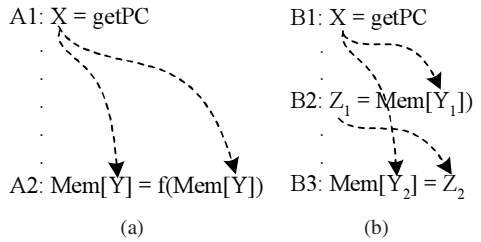

Figure 3. (a) The tainted data are used as the addresses of the updating instructions. Variable X is tainted at A_1 at the beginning. Variable Y is tainted by X and used as the address of the updating instruction A_2. (b) Attackers use a memory read instruction to read payload, modify the read result, and write the modified result back to the payload by using a memory write instruction. Variable X is tainted at B_1 at the beginning. Variable Y_1 is tainted by X and used as the address of memory read instruction B_2. Variable Y_2 is tainted by X and used as the address of memory write B_3. Variable Z_2 is tainted by Z_1 and used as the source operand of B_3.

Specifically, target addresses of indirect jump should be initialized; the operands of memory updating or writing instructions in self-modifying code should be initialized. If these operands are uninitialized, we will not consider them as attacks.

We say a variable is initialized if it is defined by a constant or other initialized variables. It is hard for attackers to know the run-time values of registers before malicious code is executed. That is, their values are unpredictable to attackers. Therefore, normally it is reasonable to assume that the initial states of all registers are uninitialized at the beginning (this assumption was also made by others [8, 23, 28], implicitly or explicitly).

There are two special cases of variable initialization. One case is the PC value, which we always consider to be initialized. The other case is for the instructions whose output is independent of their input. For example, instructions such as "xor eax, eax" or "sub eax, eax" always set eax to zero regardless of the value stored in eax. In this work, we recognize these special instructions and consider the result variables to be initialized. Note that attackers may evade our initialization analysis by crafting certain sequence of instructions to initialize a variable to desired constant without being detected. We will discuss this limitation as well as the solution in Section 4.

3.3 Detection of Plain Exploit Code

So far, we have described how we prevent attackers from exploiting either self-modifying or indirect jump obfuscation techniques. In this section, we detect the plain exploit code *with or without other obfuscation*, based on system calls and function calls detection. In order to access system resources, the plain exploit code has to use system calls or function calls. In modern operating systems such as Windows and Linux, the system call is the only way to transit from user-mode to kernel-mode to execute privileged instructions. In order to talk with kernel, the plain exploit code has to use either system calls or user-mode APIs (which eventually use system calls). Hence, if we find code patterns of system calls or function calls to user-mode API in an instruction sequence, we consider the instruction sequence as plain exploit code. Previous work [8, 28] also uses a similar idea to detect exploit code. However, their schemes are vulnerable to metamorphic obfuscation [28]. Our approach is robust to most obfuscation techniques such as metamorphism by using static taint and initialization analyses.

Windows and Linux expose system call interfaces through interrupt "int 0x2e" and"int 0x80h" , respectively. Newer versions of Windows and Linux are capable of using optimized "sysenter" instruction. Because the length of these three instructions is only two bytes, even normal network streams may contain plenty of these byte values. Therefore, using only these instructions as a detection criterion will cause a high false positive rate. We observed that before these system call instructions normally several instructions are used to transfer parameters. System call number is an obligatory parameter for a system call, which is stored in the eax register. In addition, most system calls need at least one additional input parameter stored in the ebx register. In fact, only 15 of all 190 system calls do not require the parameter in Linux. Since most of these 15 system calls such as sys_getpid and sys_getuid are merely used to read system or process information in Linux, we believe exploit code must use other system calls to achieve its purposes. Accordingly, we detect system call exploit code in the following way. We first check if an instruction sequence contains system call instructions "int 0x2e","int 0x80h" or sysenter. If the instruction sequence contains one of them, we will analyze the instruction sequence by initialization analysis. If the registers eax and ebx are initialized before system call instructions, we conclude that the instruction sequence is exploit code.

Note that our detection heuristic works not only for plain exploit code but also for metamorphic code. Instruction "int 0x2e","int 0x80h" and sysenter are the only three instructions to invoke system calls in Windows and Linux, thus the only way to obfuscate system calls is to obfuscate the instructions for transferring parameters. However, because the registers eax and ebx need be initialized despite the metamorphism being used, metamorphic system call code cannot evade our detection.

Attackers may use an alternative way to talk with operat-

ing system kernel. The only other way is through an existing user-mode API, which eventually invokes system calls, on the target machine. STILL can also detect function calls to these existing user-mode APIs by initialization analysis in exploit code. Due to page limit, we will not discuss the details. The experimental results of system call and function call detection will be showed in Section 5.

4 Security Analysis

In this section, we analyze the strength and limitations of STILL.

4.1 Strength

Besides detecting non-obfuscated exploit code, STILL is also robust to most of these obfuscation techniques. Previous sections showed how STILL detects exploit code that uses polymorphic, self-modifying, indirect jump and branch function obfuscation. Next we discuss how STILL handles the other types of obfuscation techniques.

Metamorphism Metamorphic obfuscation techniques obfuscate the behaviors of a program; however, an obfuscated program still bears the same concrete semantic or abstract semantic of the original one. Since our approach is based on the detection of these invariable abstract semantics, it is robust to metamorphic obfuscation.

Anti-disassembly is a type of obfuscation techniques that try to confuse traditional disassembly algorithms such as linear sweep [18]. It includes junk byte insertion, opaque predicate, code overlap, indirect jump and branch function obfuscation techniques.

Junk byte insertion in which junk bytes are inserted at locations that are not reachable at run-time to hinder disassembly. This insertion may mislead linear sweep algorithms, but it cannot mislead recursive traversal algorithms [16], which our algorithm is based on.

Opaque predicate is used to transform unconditional jumps into conditional branches. It allows an obfuscator to insert junk bytes either at jump targets or in the place of the fall-through instruction. We note that opaque predicate may make our approach mistakenly interpret junk byte as executable code. However, this mistake will not cause our approach to miss any real malicious instructions. Therefore, our approach is also immune to obfuscation based on opaque predicates.

Code overlap is one obfuscation technique in which several instructions share one or more bytes. Code overlap can also confuse linear sweep algorithms, but it cannot confuse recursive traversal algorithms [23].

Anti-emulation Since our approach is a static analysis approach, it is immune to anti-emulation obfuscation techniques.

4.2 Limitations

The present version of our tool has a few limitations. First, the current implementation does not handle memory address obfuscation. This limitation can be handled by treating memory access conservatively [9].

Second, attackers may use special ways to initialize a variable to a constant as a counterattack to initialization analysis. For example, the sequence of instructions ($mov\ eax, ebx;\ add\ eax, 0x1;\ sub\ ebx, -0x1;\ xor\ eax, ebx$) shows a special way to initialize eax to 0. This limitation can be handled by combining our initialization analysis with symbolic execution [14]. This combination, however, may cause some performance overhead. Fortunately, this overhead is not always incurred, since initialization analysis is only used to reduce false positives.

Finally, like previous work [8, 15, 28], STILL does not detect return-to-libc attacks which do not contain any code. One way to alleviate return-to-libc attacks is to map (through mmap()) the addresses of shared libraries so that the addresses contain null bytes [1] [11].

5 Experimental Results

We implemented a stand-alone and a proxy-based prototype of STILL to evaluate our technique. The stand-alone prototype consists of the following three parts: (1) *loader* to load the input files from disks; (2) *disassembler* to distill instruction sequences from the input; (3) *analyzer* to analyze the instruction sequences. The prototype will raise one of the following four types of detailed warnings for each input file that is identified with exploit code: (Type1) A function call exploit code is detected; (Type2) A system call exploit code is detected; (Type3) An indirect jump obfuscation code is detected; and (Type4) A self-modifying obfuscation code is detected. Note that polymorphic exploit code is included in Type4. The proxy-based prototype is similar except that it operates as a proxy taking its input from the network.

5.1 Detection Effectiveness

Polymorphic Shellcode To evaluate the detection effectiveness of STILL, we collected 12,000 polymorphic attack messages from 10 publicly available polymorphic engines. Among these ten, seven engines are from the Metasploit framework [4], including Countdown, Alpha2, JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai. The other three engines are CLET [12], ADMmutate [19], and JempiScodes [3]. Countdown uses a decrementing byte as the key for decryption; Pex and

[1] Null bytes, which are C string terminators, cause attacks terminate before they overflow the entire buffer.

JumpCallAdditive are xor decoders and use call to get PC; PexFnstenvMov and PexFnstenvSub are xor decoders and use fnstenv to get PC; Alpha2 generates alphanumeric payload; ShikataGaNai, CLET, ADMmutate, and JempiScodes are advanced polymorphic engines, which also obfuscate the decryption routine by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to defeat data mining methods.

For JempiScodes, we generated 3,000 different attack messages, 1,000 per each of its three obfuscation algorithms. We also generate 9,000 different attack messages, 1,000 per each of the other nine engines. We tested the stand-alone prototype of STILL using these 12,000 attack messages. All of these messages are successfully detected and reported as warning Type4. We also use these 12,000 attack messages to test SigFree. SigFree can only detect 4,103 of them, because the other attack messages have only 6 to 10 useful instructions which is much below than the detection threshold 15 used in SigFree. Our result shows that STILL has much better detection coverage than SigFree.

Plain exploit code from Metasploit framework There are totally 23 different Windows plain shellcode and 11 different Linux plain shellcode in Metasploit framework v2.5. We generated 34 attack messages for each of them. Note that for multi-stage shellcode such as "Linux IA32 Staged Bind Shell", we use the loading (first) stage code to generate attack messages to show that we can detect multi-stage shellcode at its first stage. We tested all of them by the stand-alone prototype. All Linux plain shellcodes are detected as Type2. "Windows Execute Command" and "Windows Execute net user" are detected as Type3, and all other Windows plain shellcodes are detected as Type1.

Worms We also tested on worms CodeRed I, CodeRed II, Sasser, Slammer and Blaster. CodeRed I, CodeRed II, Sasser and Slammer are successfully detected as Type1. Blaster is successfully detected as Type4, since it exploits polymorphism obfuscation.

5.2 False Positives

Legitimate HTTP traffic We tested the prototype over real HTTP traffic shown in Table 1(a). Due to privacy concerns, we were unable to deploy a prototype in a large network to examine real-time traffic. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded all the http requests and replies of a user during his/her daily Internet surfing. During a three-week period, 7 people installed the proxy and helped with collecting totally 378,158 HTTP requests/replies stored in 7 separate datasets. The replies and requests include various types of multimedia data such as video/audio and image files, manually typed urls, clicks through various web sites, search results from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPS requests. The overall size of the traces is about 1.77 GB.

Table 1.

(a) HTTP real-traffic datasets

Name	Requests	Replies	Size(MB)
Dataset1	5,026	3,584	317.24
Dataset2	32,666	27,253	288.53
Dataset3	87,260	52,772	355.99
Dataset4	17,871	16,941	320.91
Dataset5	26,759	28,028	129.41
Dataset6	17,075	10,042	237.52
Dataset7	46,107	6,774	158.28
Total	232,764	145,394	1,807.88

(b) False positives classified by mime-types

Mime-type	Warning Type			
	1	2	3	4
application/octetstream	10	1	0	0
application/x-mms-framed	0	1	0	0
application/x-shockwave-flash	0	2	0	0
audio/mpeg	0	1	0	0
image/gif	0	6	0	0
image/jpeg	0	2	0	0
text/plain	0	1	0	0
flv-application/octet-stream	0	0	0	2
video/flv	0	0	0	1
video/x-flv	0	1	0	2
Total	10	15	0	5

Our detection results are as follows. First, STILL raised no alarms on the HTTP requests to servers. Second, STILL raised warnings on 30 HTTP binary replies coming into client-side web browsers. Because none of them contain malicious exploit code, they all are false positives and the false positive rate is 0.0079%. Table 1(b) shows the number of false positives classified by mime-types in HTTP headers of input. We note that although there are 30 false positives, most of them can be easily precluded by further manual or automatic analysis. First, by checking these files, we found that nine of them were indeed Win32 executable in PE (Portable Executable) file format. Because the purpose of STILL is to detect if a request contains code/mailicous code rather than to discern malicious code from legitimate code, we could let end users to decide whether or not to block these executables. Second, some Type2 warnings are obviously false positives because register eax is set to a number much larger than 256, which is not a valid system call number in Linux and Windows. Finally, we can reduce false positives by checking the context. For example, if we believe there are no vulnerabilities in a Shockwave Player (because of other reliable tools), we may ignore the two false positives where the mime type is "application/x-shockwave-flash".

5.3 Performance Evaluation

The stand-alone prototype was written in C++ programming language in Win32 environment, and compiled with Borland C++ version 5.5.1 at optimization level -O2. The experiments were performed in a Windows XP Professional

SP2 environment running on a Pentium CPU operating at 3.0GHz, equipped with 1GB 533MHz DDR2-SDRAM ECC main memory.

We measured the throughput of the stand-alone prototype over the real traffic datasets shown in Table 1(a). The results are summarized in Table 2. To remove the overhead specific to the test environment, we loaded the input files into main memory first and measured the time for our detection algorithm to scan the files. Also, the time spent by the loader was deliberately excluded when we measured the elapsed time. The average processing speed was 1.95Mbits/sec; however, 83% of elapsed time was spent on disassembly. Since the *instruction decoder* part of our disassembler, borrowed from Ollydbg debugger [30], was originally used for offline decoding, we believe optimizing the instruction decoder could dramatically increase the overall throughput. We will investigate it in our future work. For the 1.77 GB of input processed, our analyzer processed disassembled sequences at 11.62Mbits/sec.

Table 2. Experiment results

Input data	Elapsed time (sec)		Analysis throughput (Mbps)
	Disassembly	Analysis	
Dataset1	855.785467	267.055825	9.503267
Dataset2	1075.931188	179.813182	12.836709
Dataset3	1373.455207	167.464756	17.00598
Dataset4	1047.691866	233.835534	10.978939
Dataset5	483.113477	82.381132	12.567039
Dataset6	825.120756	228.234238	8.325504
Dataset7	493.956068	85.463046	14.816603

6 Related Work

This paper is mainly relevant to the following work.

Exploit Code Detection by Static Analysis Kruegel et al. [15] exploited control flow structures to detect polymorphic shellcode and worms. Chinchani and Berg [8] used pattern-matching and data flow analysis techniques to detect exploit code inside network flows. SigFree [28] blocks attacks by detecting the presence of code in data requests. It uses a new technique called code abstraction to differentiate code and data. One benefit of these static analysis approaches is that they can detect both foreseen exploit code exploiting known vulnerabilities and zero-day exploit code exploiting unknown vulnerabilities. In addition, they are in general more resilient to polymorphism and metamorphism (than string-matching signatures). However, Polychronakis et al. [23] demonstrated that some anti-static-analysis techniques such as self-modifying and indirect jump can easily thwart these existing static analysis techniques.

Exploit Code Detection by Emulation Polychronakis et al. [23] firstly proposed a NIDS-embedded CPU emulator to detect polymorphic shellcode. Zhang et al. [31] proposed another emulator to detect polymorphic shellcode. In addition, they use a static analysis method to identify the start point of polymorphic shellcode which is potentially faster than [23]. The emulators, being a dynamic analyzer, are immune to most anti-static-analysis techniques. However, dynamic analysis is vulnerable to several anti-emulation techniques, which have existed in virus writer community for many years [26]. For example, it may be thwarted by random worms that only execute on a specific condition such as a randomly generated date or time. Another example is that it must use some heuristics to determine when to stop analyzing a program, because execution may not terminate. An experienced attacker may bypass the stopping heuristics by introducing a delay loop that simply wastes cycles. We note that the emulators in [23, 31] detect only polymorphic shellcode, thus plain (with or without metamorphism) exploit code such as worm Code Red will evade its detection. Motivated by [23], we proposed STILL, which is robust to both anti-static-analysis and anti-emulation techniques. In addition, STILL is a comprehensive solution, which can detect both polymorphic and plain (with or without metamorphism) exploit code.

Disassembly The first step of static analysis for executables is disassembly, which translates machine code to assembly code and generates CFG. Linn and Debray [18] introduced some obfuscations that can thwart the disassembly process. Kruegel et al. [16] investigated disassembly of obfuscated binaries. Balakrishnan and Reps analyzed memory accesses obfuscation in x86 executables [7].

7 Conclusion

We developed STILL, a novel static taint and initialization analysis approach, to address code obfuscation. Our static taint analysis technique enables STILL to collect strong evidence of self-modifying and/or indirect jump code obfuscation behaviors. As a result, while self-modifying code or indirect jumps may avoid the detection of other static analysis approaches, STILL can detect them with a high accuracy. Our experiment results show that STILL detected all the 12,000 exploit codes generated by 10 well-known shellcode generation engines and STILL achieves 0.0079% false positive rate in analyzing 378,158 HTTP requests/replies.

Acknowledgment

We would like to thank the members of Penn State Cyber Security Lab for collecting real traces. The work of Xinran Wang and Sencun Zhu was supported in part by the National Science Foundation (CAREER NSF-0643906); the work of Yoon-Chan Jhi and Peng Liu was supported in part by NSF CNS-0716479, AFOSR MURI: Automatic Recovery of Enterprise-wide Systems after Attack or Failure with Forward Correction, and AFRL award FA8750-08-c-0137.

References

[1] Computer emergency response team (cert). http://www.cert.org.

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

[3] Jempiscodes - a polymorphic shellcode generator. http://www.shellcode.com.ar/en/proyectos.html.

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

[5] Microsoft security bulletin. http://www.microsoft.com/technet/security/current.aspx.

[6] the de facto standard for intrusion deetection/preventions. http://www.snort.org.

[7] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction*, 2004.

[8] R. Chinchani and E. Van Den Berg. A fast static analysis approach to detect exploit code inside network flows. In *RAID*, 2005.

[9] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables.

[10] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July 1997.

[11] Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/0063.html.

[12] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Superbus Von Underduk. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[13] C. Ionescu. Getpc code. http://www.securityfocus.com/archive/82/327348/2006-01-03/1.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[16] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004*, August 2004.

[17] Z. Li, M. Sanghi, Y. Chen, M. Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, May 2006.

[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM CCS*, 2003.

[19] S. Macaulay. Admmutate: Polymorphic shellcode engine. http://www.ktwo.ca/security.html.

[20] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd ACSAC*, 2007.

[21] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Security and Privacy Symposium*, May 2005.

[22] Noir. Getpc code. http://www.securityfocus.com/archive/82/327100/2006-01-03/1.

[23] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *DIMVA*, 2006.

[24] S. Singh, C. Estan, G. Varghese, and S. Savage. Earlybird system for real-time detection of unknown worms. Technical report, Univ. of California at San Diego, 2003.

[25] sk. History and advances in windows shellcode. Phrack, vol. 11, no. 62, July 2004.

[26] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[27] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM CCS*, 2005.

[28] X. Wang, C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *15th Usenix Security Symposium*, July 2006.

[29] X. F. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *ACM Conference on CCS*, 2006.

[30] O. Yuschuk. Ollydbg disassembler, 2006.

[31] Q. Zhang, D. S. Reeves, P. Ning, and S. Purushothaman Iyer. Analyzing network traffic to detect self-decrypting exploit code. In *AsiaCCS*, 2007.