International Journal of Security and Its Applications
Vol. 10, No. 6 (2016) pp.107-122
http://dx.doi.org/10.14257/ijsia.2016.10.6.12

# Shellix: An Efficient Approach for Shellcode Detection

Mo Chen, Donghai Tian, Yuan Liu, Changzhen Hu, Xin Wang, and Ning Li

*China Aerospace Engineering Consultation Center*
*Beijing Institute of Technology*
*Institute of Computer Application, CAEP*
*Beijing Institute of Technology*
*China Aerospace Engineering Consultation Center*
*China Aerospace Engineering Consultation Center*

***Abstract***

*Code injection attacks are widely used by attackers to compromise computer systems. Attackers could obtain the control of a victim's computer system by injecting shellcode to the vulnerable program. The existing solutions to detect shellcode can be grouped into two categories: static analysis and dynamic analysis. Static analysis can detect shellcode efficiently, but cannot handle the shellcode that employs obfuscation techniques. Dynamic analysis is able to detect the obfuscated shellcode, however it is still limited to detect the recent virtualization-aware shellcode. In this paper, we present a novel shellcode detection approach without using any virtualization technology. We implement our approach based on the commodity OS kernel which is compatible to the existing system. Our approach is able to detect the shellcode that could be aware of the virtualization environment and reduces the probability of exposing detection environment. Our experimental evaluations show that our system can effectively detect a large set of shellcode instances with good performance.*

***Keywords****: Shellcode Detection, Payload Execution, Hardware-based, OS Kernel*

## 1. Introduction

Code injection attacks are proposed for several years. Generally, the attackers send malicious data that contain shellcode to the vulnerable program running on the target system. Based on the vulnerability exploited, the attacker may change the program's execution and even control the entire system by the injecting the shellcode. Despite many solutions are proposed to solve these attacks in the last decades, such vulnerabilities (*e.g.*, stack/heap/ integer overflow, format string) are still the main source of vulnerabilities in applications. This makes the code injection attacks become still popular approaches for attackers to compromise computer systems. Therefore, how to detect the code injection attacks is still a very important research topic.

The existing solutions to this problem can be divided into two categories: static analysis and dynamic analysis. Static analysis just disassembles the shellcode statically, thus it can detect the shellcode efficiently. However, this approach is limited to detect the advanced shellcode that employ obfuscation techniques. Dynamic analysis based on emulation technique could detect the obfuscated/polymorphic shellcode. Basically, these methods employ CPU emulators to directly execute network streams as binary code [1]. Their methodology is based on the fact that shellcode should contain a series of valid instructions, whereas the benign network data may contain some illegal or invalid instructions that will cause exceptions when they are executed.

Since dynamic analysis could detect a particular set of polymorphic shellcode with the ability of self-decrypting, it has been well developed these years and proved to be effective. However, most shellcode detection systems based on dynamic analysis could

ISSN: 1738-9976 IJSIA

not achieve high performance, because the instructions translation in the CPU emulators introduces a high performance.

Snow *et. al.*, [2] introduce a shellcode detection system, called ShellOS, which is quite different from previous dynamic systems [1, 3, 4]. They leverage the KVM [33] (Kernel-based virtual machine) hypervisor to make most of instructions execute on real CPU directly. So ShellOS can achieve a high performance than other shellcode detection systems. Nevertheless, there is still small part of instructions needed to be emulated in KVM. As a result, it is possible that the shellcode may first detect the virtualized environment before the effective part of the shellcode get executed [5, 6, 7]. Recently, Abbasi [8] *et. al.*, propose a number of techniques and develop several virtualization-aware shell-codes to evade the emulated-based shellcode detection systems. In this paper, we design a novel approach to detect shellcode. Our approach allows executing network streams on hardware directly without using virtualization technology. That means that our approach can detect the virtualization-aware shellcode. We have implemented a prototype system, called Shellix, which is based on Windows for detecting Windows Shellcode. To make Shellix compatible with widely deployed Linux-based gateway, we also have implemented the Linux version of Shellix.

Comparing with previous shellcode detection systems, Shellix has these special features as follows:

Hardware-based: Shellix enables network streams to be executed on physical CPU directly, without relying on the virtualization technology. It is able to detect the virtualization-aware shellcode. Therefore, it can reduce the probability of exposing detection environment.

Easy-to-deploy: Most of previous shellcode detection systems [1-4] are specially designed, they may not have a good compatibility with existing OSs which make it difficult wide deployment. In contrast, Shellix kernel is developed based on commodity operating system kernels including Windows and Linux kernel. So it is compatible with existing operating systems and easy to widely deploy.

Good performance: Our evaluations show that the detection performance of Shellix is as good as that of most shellcode detection systems.

## 2. Threat Model

This paper is focused on detecting the self-contained shellcode which is aimed at compromising the user application. The other kinds of shellcode, such as non-self-contained shellcode and kernel-level shellcode, are not in the scope of this paper. We assume the goal of an attacker A is to inject shellcode to a vulnerable user application B, and then hijacks B's execution. Once the attacker A gets the execution control of B, A could do anything B is authorized to do (*e.g.*, memory read/write in B's memory space, altering the control flow of B). Since our work focus on the user-level shellcode, we assume the OS kernel is trusted, and it can not be directly accessed by the shellcode.

## 3. Design Overview

Shellix relies on a dynamic analysis approach, by which network streams are executed as binary code. The success of this methodology relies on the fact that the shellcode should be a series of valid instructions, whereas the benign data may contain illegal or invalid instructions that would cause exceptions when it get executed. Since the entry point of shellcode can not determined exactly beforehand, Shellix has to try each byte of the network stream as the first instruction. Similar to the previous dynamic methods [1, 2], we refer to each instruction sequence starting at each byte offset of the network stream as an execution chain.

Unlike previous dynamic analysis approaches, Shellix relies on a user detection program to execute network stream, without relying on any virtualization technology. The

 

network stream would be executed as real machine instructions. Shellix provides an appropriate execution environment to the real target vulnerable applications. Therefore, it is unlikely to be discovered by the virtualization-aware shellcode. As we know, most of execution chains in the network stream are benign and meaningless, and they may contain some illegal or invalid instructions.

Consequently, executing these instructions would make the detection program crash. In order to execute each execution chain correctly and continuously, we have to address the following technical challenges:

**C1. Registers and Memory Damage.** After an execution chain is executed, the correctness of registers and user memory of the detection program cannot be guaranteed (*e.g.*, the *esp* register and stack memory data may be corrupted).

**C2. Causing Exception.** An exception would be raised by executing invalid or illegal instructions. Thus, the detection program will be killed by the OS.

**C3. Falling into Infinite Loops.** To detect these loops, the emulator-based methods need to count the number of executed instructions. However, the commodity hardware does not provide this primitive. For these challenges, we present the corresponding solutions in the following sections.

### 3.1. Shellcode Detection

Only executing network streams cannot finally determine if these network streams contain shellcode. So we employ the runtime heuristics [3] for Windows shellcode detection. In our prototype, we apply the PEB heuristic.

PEB heuristic is based on the characteristic that many shellcodes have to locate the base address of *kernel32.dll* for Windows API resolution through PEB (Process Environment Block). PEB is a user-level structure that holds extensive process-specific information of Windows. The core idea of this heuristic is to check whether the PEB-related memory addresses are accessed by injected shellcode.

Our approach to trap the specific memory access is based on the idea of stealth breakpoint [9], which allows setting unlimited number of memory traps. In specific, we utilize the hardware paging mechanism to set breakpoints on the PEB-related memory addresses. In this way, if these addresses are accessed, the OS could trap these access operations.

It is worth noting that the other detection heuristics (*e.g.*, SEH heuristics) which are compatible with dynamic analysis approach can also be applied [3]. Our prototype does not cover all kinds of shellcode, because our main focus is to provide an efficient non-virtualization network streams execution platform for VM-aware shellcode detection, but not to explore the shellcode detection heuristics.

### 3.2. Register and Memory Protection

To address the challenge C1, we save the registers and memory contents of the detection program into the kernel space before the detection program begins to execute the network stream. The saved registers and memory contents are copied back when an execution chain is terminated. In this way, the detection program could rollback to a clean state. Our method ensures that each execution chain has the same execution environment.

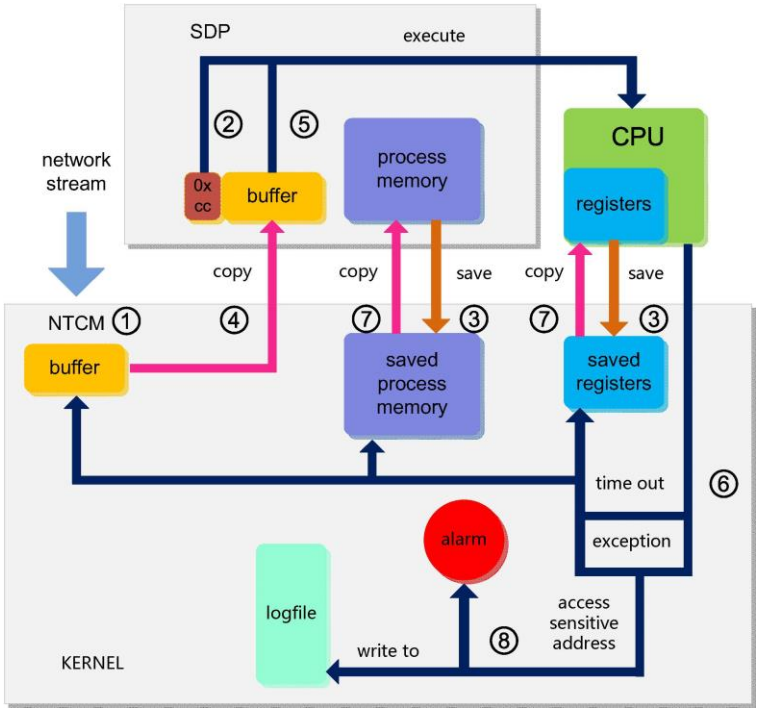


**Figure 1. Shellix Architecture**

### 3.3. Exception Handling

To deal with the challenge C2, we hook the OS default exception handling routine to avoid the detection program being killed after an exception is raised. Specifically, we restore the saved registers and memory contents, modify the program counter to the beginning of the next execution chain, and then make the exception handling routine return before the detection program is killed by the OS. In this way, the detection program will work continuously.

### 3.4. Infinite Loop Detection

To address the challenge C3, we detect the infinite loop by utilizing the timer interrupt. As we know, the timer interrupt will be generated at a certain frequency by hardware. So this gives us a chance to know how long time the execution chain has been executed when the timer interrupt occurs.

By constructing a great number of experiments, we observe that, in a normal case, the length of bytes that CPU can execute is far beyond the length of an execution chain in one timer interrupt period. Thus, we could judge that the execution chain falls into infinite loop, if it does not generate any exception or the OS does not trigger the shellcode detection runtime heuristics after one timer interrupt period.

Accordingly, we hook the timer interrupt handler to restore registers and memory contents of the detection program and change the program counter for the next execution chain, if the infinite loop happens.

### 3.5. Architecture

Figure 1 shows the architecture of Shellix. It composes of three parts: Network Traffic Capture Module (NTCM), Shellcode Detection Program (SDP) and Kernel. The main components are SDP and Kernel. The working procedure of SDP is shown in Figure 2. It mainly includes two parts. One is a global array that stores the network data for execution. The other one is a function pointer which points to the start address of the global array. In

 

this way, SDP could execute the data in the global array when the *(*ptr)* function pointer is invoked. Moreover, the PEB heuristic is also established in SDP. The Kernel component is a kernel module to perform some kernel operations. It is used to 1) handle exceptions, 2)copy network stream to SDP, 3) save and restore registers and memory contents for SDP, and 4) deal with infinite loops.

```
1 void (*ptr)(); //function pointer
2 unsigned char buf[BUF_SIZE]; // data for execution
3 void main() {
4 buf[0] = 0xcc; // firstly execute an assembly instruction
                    int 3 to cause a breakpoint exception
5 ptr = (unsigned long)buf; // make ptr point to buf
6 (*ptr)(); }// call ptr to execute data
```

**Figure 2. The Working Procedure of SDP**

The workflow of Shellix is as follows. Firstly, NTCM captures and stores the network traffic from NIC (Network Interface Card) into the Kernel (Step 1 in Figure 1). Then, SDP execute a byte *0xcc* (equal to assembly instruction *int 3*) (Step 2) to raise a breakpoint exception. Next, in the breakpoint exception handling routine, the Kernel component saves the registers and memory contents of SDP in the OS Kernel (Step 3). After that, the network stream to be analyzed is copied to the global array of SDP (Step 4). Then, each execution chain in the network stream is executed one by one (Step 5). Whenever these execution chains raise exceptions or fall into infinite loops (Step 6), the Kernel restores the registers and memory contents with the saved ones (Step 7), also the network stream is copied back to the SDP buffer (Step 4). If an exception is caused by accessing the sensitive memory addresses, the Kernel will raise alarms (Step 8). When the last execution chain is terminated, the Kernel copies the new network stream to the SDP.

### 3.6. System Robustness

Since our paper focuses on the user-space program's shellcode which is less likely to directly access the OS kernel, the Kernel and NTCM components of Shellix that reside in the kernel-space are secure and can not be damaged by the shellcode. However, the SDP component is a common user-space program, so it is vulnerable for the shellcode. An attacker who knows the existence of Shellix may compromise the whole system by sending a piece of shellcode to the SDP without using any exploits, since the shellcode will be directly executed in SDP.

To address this problem, we prohibit the SDP to invoke system calls after it is loaded in the memory. Particularly, we hook all the system calls of the OS kernel and disable them when the SDP is ready to execute the network stream. In this way, the system calls (*e.g.*, file access, command execution, network communication) issued by the shellcode can not be executed. Without these functions, the shellcode can not compromise Shellix. Whereas, doing so will only have little impact to our detection system, because Shellix does not need to invoke system calls at run-time (Figure 2).

### 3.7. System Deployment

Shellix can be deployed in the IDS of a LAN. In this way, the network streams which contain shellcode can be discovered before they reach the hosts in the LAN. To improve the throughput of whole system, Shellix can be deployed in concert with the network identifiers for large files transfer streams [10, 11]. Since these streams are unlikely to contain shellcode, the identifiers can firstly filter these streams. Thus, only a part of network streams are necessary to be executed in Shellix.

# 4. Implementation

We have implemented Shellix on Windows XP SP3 x86, and Linux x86 with kernel version 2.6.32 [12] respectively. To solve the technical challenges mentioned above, we need to change the original interrupt/exception handling routine of the OS kernel for the SDP. For this purpose, we make use of inline-hook technique. The advantage of this technique is that it does not need to recompile the source code of the OS kernel and makes little change to the original binary code. The core idea of inline-hook is to replace some binary instructions of the kernel function to a *call* instruction which points to the custom function by hard-coding. After processing the custom procedure, the custom function executes the overriden instructions of the hooked kernel function, and then gives the execution back to the hooked kernel function at the end.

Additionally, since the basic principle of implementation for modern OSs is similar, our approach can be applied to other operating systems, such as Mac OS. In the following subsections, we firstly introduce the implementation of Shellix in Windows environment. The implementation of Shellix in Linux environment is described in the last subsection.

## 4.1. Shellcode Detection

**Heuristic Detection** PEB heuristic is based on the characteristic that many shellcode need to locate the base address of *kernel32.dll* for Windows API resolution. To determine the base address of *kernel32.dll*, most shellcode make use of the *PEB* structure. As Figure 3 shows, the shellcode first reads the address *FS: [0x30]* to access the *PEB*. Then it accesses the pointer (at *0x0c* byte in *PEB*) to *PEB_LDR_DATA* structure which holds the first entry of InInitializationOrder module list. After that, the shellcode accesses the second list entry of InInitializationOrder module list, and finally gets the base address of *kernel32.dll*.

According to PEB heuristic, we set breakpoints at the PEB-related memory addresses that would be accessed by the shellcode (*e.g.*, the entry of *PEB* and *PEB_LDR_DATA*). If all these breakpoints are accessed, Shellix will identify the potential existence of shellcode.

```
1 xor eax, eax                ; eax = 0
2 mov eax, fs:[eax+0x30] ; get a pointer to PEB
3 mov eax, [eax+0x0C]    ; get a pointer to PEB_LDR_DATA
4 mov esi, [eax+0x1C]    ; get the first entry of
                             InInitializationOrder module list
5 mov eax, [eax]             ; get the second list entry
6 mov edi, [eax+0x08]    ; get the kernel32 base address
```

**Figure 3. The Representative Examples of Code that Gets the Base Address of kernel32.dll through the PEB**

**Breakpoint Setting** We make use of the stealth breakpoint technique proposed by Vasudevan *et. al.*, [9] to trap the PEB-related memory access.

Specifically, we set the access attribute of the page table entries (that are associated with the memory addresses) to be *not present*. By doing so, a page fault exception would be raised if shellcode accesses the breakpoint address. We hook the page fault exception handling routine and check if the fault address stored in CR2 register is in the list of breakpoints. If it is, the Windows kernel reports this event and logs it in a file. To ensure that the shellcode continues its execution, we set back the access attribute of this page to *present* to prevent recursive page fault in this exception handling routine.

On the other hand, to trap the memory access at the same memory address once again, we activate the debug mechanism of the OS kernel to reset the associated page table

 

entries. The debug exception is triggered by executing each instruction when the *TF* flag in *EFLAG* register is set. We activate the *TF* flag in the page fault handling routine, once the memory breakpoint is accessed. By doing so, when the program re-executes the instruction that causes the page fault exception, it will trap into OS kernel due to the debug exception. Then, we set the attribute of the page table entry to not *present* in the debug exception handler. Finally, we clear the *TF* flag and disable the debug mechanism.

### 4.2. Register and Memory Save and Restoration

As shown in Line 4 of Figure 2, SDP firstly executes an instruction *int 3* to cause a breakpoint exception before executing the network stream. By utilizing this exception, the Windows Kernel firstly saves the user mode registers which are represented by a data structure *KTRAP_FRAME*, and then transfers them to the exception handling routine as a parameter. In the breakpoint exception handling routine, we save the *KTRAP_FRAME* value in the kernel-space. Moreover, we also save the memory contents of SDP in the kernel-space. Whenever an exception is raised by executing an execution chain, we use the saved *KTRAP_FRAME* value to recover the current *KTRAP_FRAME* value and copy the saved memory contents back to SDP. In this way, we can ensure that each execution chain has the same execution environment.

Since the execution chains cannot modify the readable memory regions, we only need to save the memory blocks whose access attribute are writable (exclude the block used to store data for detection). We traverse the SDP's data structure *MMVAD* which stores the information of memory blocks, and check the read/write control field *CONTROL_AREA* to decide whether this memory block is writable. If it is, we save it into the Kernel.

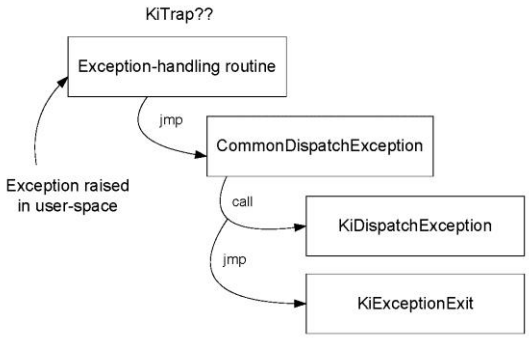


**Figure 4. Windows Exception-Handling Mechanism**

### 4.3. Exception Handling

As shown in Figure 4, when a user-space program raises an exception, the Windows kernel will enter to the different exception handling routines according to the exception type (*e.g.*, exception handling routine *_KiTrap0E* is used to handle page fault exception). Most of the exception handling routines will jump into the *CommonDispatchException* routine, and then the *CommonDispatchException* call the function *KiDispatchException* for the final exception handling. If there are no user-registered exception handlers or debuggers for handling this exception, the user-space program will be killed by the OS.

To change the default exception handling procedures, we hook the *KiDispatchException* function in the kernel. Specifically, if an exception is raised by SDP, we first restore the registers and memory contents of SDP and then modify the program counter to the next execution chain in this function. Then we make the function return immediately. In this way, SDP will not be killed by the OS.

A small part of exception handling routines does not call the function *KiDispatchException* (*e.g., Invalid TSS exception* handling routine), they directly call the

function *KiExcepitonExit* to make the SDP exit. For these exception handing routines, we hook the function *KiExceptionExit*. The handling procedure in *KiMyExceptionExit* is similar with what we do in *KiDispatchException*.

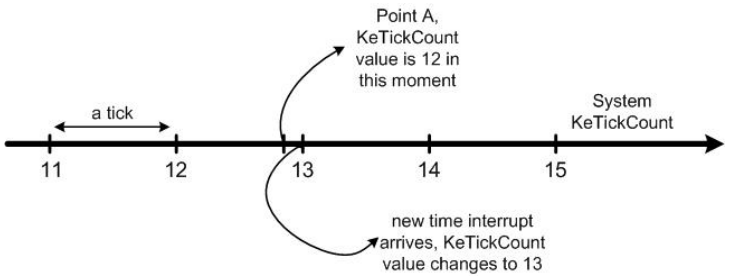


**Figure 5. Time Interrupt Interval Choosing**

### 4.4. Dealing with Infinite Loop

Windows kernel uses a global variable *KeTickCount* to record the number of timer interrupt since the OS starts (the time between two timer interrupt is called a **tick**). It increases by one when a timer interrupt occurs.

To figure out how long time an execution chain is executed, we check the difference between the *KeTickCount* value at the start of the execution chain and the current *KeTickCount* value when the timer interrupt occurs. Specifically, we first get the *KeTickCount* value at the beginning of each execution chain and save it as GlobleKeTickCount. Then we hook the timer interrupt handler of Local APIC, *HalpClockInterrupt()*. Whenever the timer interrupt happens, we use the current *KeTickCount* value to compare with the *GlobleKeTickCount*. If the difference between the *GlobleKeTickCount* and the current *KeTickCount* is equal to two ticks, we can judge that the execution chain has fallen into infinite loop. Then we restore the registers and memory contents of SDP and change the program counter for the next execution chain.

We use two ticks as the threshold instead of using one tick, because the exception may occur at any time between two timer interrupt. As shown in Figure 5, at point A, the current execution chain ends and the new execution chain starts. The *KeTickCount* value is 12 at this moment. Continuing executing, the timer interrupt will be encountered soon after few instructions are executed in the new execution chain. If we abort this new execution chain for the reason that the *KeTickCount* value is already 13 (actually the infinite loop does not occur), we may fail to detect the shellcode that does not locate in the beginning of execution chain.

Since we cannot set a certain instruction execution threshold like software-based CPU emulation approaches for detecting infinite loop, we could only set an approximate value by adjusting the timer interrupt frequency (TIF) of OS. The appropriate TIF is hard to determine. Low TIF means that a longer time is needed to detect the infinite loop. On the other hand, the infinite loop can be detected quickly if a high TIF is used. However, the timer interrupt handling also occupies CPU processing time. Too frequent timer interrupt makes CPU use a large amount of time to process timer interrupt. Both low and high TIF could affect the shellcode detection accuracy or performance. To find the proper timer interrupt frequency, we performed an experiment. The details are given later in Section 5.2.

 

### 4.5. Execution Loop Control

As stated above, we make SDP rollback to a clean state when an execution chain is abort. As a result, SDP will only execute the first execution chain if we do not make any changes.

To make SDP execute execution chains one by one, we modify the value of *KTRAP_FRAME.Eip* when an execution chain is trapped into an exception handling routine. Since Windows kernel uses this value to restore the *eip* register of SDP, SDP will execute the network stream from the *Eip* address once the exception handling routine is returned. As stated in Section 3.5, we use an instruction *int 3* to create a chance to save the registers and memory contents before executing the network stream. At that moment, the value of *KTRAP_FRAME.Eip* is the start address of first execution chain. We save that value in the global variable *GlobleEip*. And we use a global counter to count the number of executed execution chains. When an execution chain is trapped into an exception handling routine, we use the value of *GlobleEip* and the counter to figure out *KTRAP_ FRAME.Eip* value. In this way, Shellix can execute the next execution. Note that, since the SDP has rolled back to the clean state before, changing the program counter to the next execution chain will not occur any risks.

Due to the design of SDP (Line 7 in Figure 2), SDP will stop to execute the network stream when it encounters a return instruction in function *(*ptr)()*. To address this problem, we modify the function's return address, which is stored on the stack, to a kernel space address. By doing so, a page fault exception will be raised if the return instruction is encountered. Then SDP could execute the next execution chain after the exception is handled. When we need to exit the SDP's execution, the original return address is copied back to the stack of SDP.

### 4.6. Implementation on Linux

In addition to the Windows implementation, we have also developed a prototype based on Linux. The main reason for us doing so is that our system can be easily optimized and integrated into the IDS which relies on Linux.

To apply the PEB heuristic in Linux, we need to mimic a part of Windows environment related to the PEB heuristic. Otherwise, the Windows shellcode would be considered as invalid instruction sequences in Linux. Particularly, we allocate a chunk of heap memory of SDP for building the simulated Windows environment. Then we establish the PEB-related data structure (*e.g., PEB, PEB_LDR_DATA*) and initialize them in this allocated memory. Finally, we set up the *FS* register to point to the base address of TEB (Thread Environment Block) which contains the address of *PEB*. The implementation of breakpoint setting is the same with the Windows version.

For register and memory save and restoration, dealing with infinite loop, and execution loop control, the basic implementation ways are very similar between the Linux version and the Windows version. The only difference is that Linux uses different data structures, variables and functions to implement the same functionalities compared to Windows. Particularly, 1) Linux kernel uses data structure *pt_regs* instead of *KTRAP_FRAME* to save the user mode registers. 2) The memory block information is stored in the data structure *vm_area_struct*, and the *vm_flags* is the read/write control field. 3) Linux uses the global variable, called *jiffies*, to count ticks, and the timer interrupt handler for Local APIC is named *smp_apic_timer_interrupt()*. 4) The value used to restore the *eip* register of SDP is the *pt_regs.ip*.

For exception handing, we hook all the exception handlers of the Linux kernel respectively. Once the exception handler is invoked, we restore the registers and memory contents of SDP, modify the program counter to the next execution chain in this function, and make the handler return immediately. Note that the page fault handler is a little different from other exception handlers due to its complexity. We could not hook this

handler at the beginning. Particularly, we should consider the situation that the page fault is raised by accessing the memory page that has been allocated but paged out. (We define this type of page fault as *fake-page-fault*, and define the page fault raised by accessing the memory page that has not been allocated as *real-page-fault*.) This type of exception can not be hooked and should be handled by the OS kernel. In Windows kernel, we do not need to consider the *fake-page-fault*, since it is handled in the function *MmAccessFault*, and we only need to hook the function *KiDispatchException* that handles *real-page-fault*.

## 5. Evaluation

To conduct the experiment, we build a testbed with two machines running LanTraffic V2 Enhanced [13] in Windows to send and receive traffic, and two machine running Shellix (Windows version and Linux version respectively) to detect shellcode. The environment of these machines is 2-core Intel Core2 Duo E8400 3.00GHz with 4GB RAM. All these machines work in the same LAN environment.

**Table 1. Shellcode Detection**

| | non-encoder | alpha_upper | call4_dword_xor | countdown | fnstenv_mov | alpha_mixed |
|---|---|---|---|---|---|---|
| Win XP | √ * | √ * | √ * | √ * | √ * | √ * |
| Linux | √ * | √ * | √ * | √ * | √ * | √ * |
| | unicode_upper | shikata_ga_nai | unicode_mixed | jmp_call_additive | TAPiON | VE-aware |
| Win XP | √ * | √ * | √ * | √ * | √ * | √ * |
| Linux | √ * | √ * | √ * | √ * | √ * | √ * |

Note : √* means #1-4,#6,#7,#10 payload are detected.

### 5.1. Effectiveness

To evaluate effectiveness of Shellix to detect the obfuscated shellcode, we use Metasploit [14] to generate attack payloads, and the popular encoders listed in Table 1 are used to obfuscate them. Additionally, to test if Shellix can detect the virtualization-aware shellcode, we develop a custom shellcode encoder (VM-aware) that uses the cache timing analysis approach [7] to test the virtualization environment, and attach it before the attack payloads.

Figure 3 shows the representative example of code that gets the base address of *kernel32.dll* through the *PEB*. Except for the first instruction, each instruction corresponds to a specific memory access. All the memory accesses are necessary for the shellcode to find the base address of *kernel32.dll*. We set the breakpoints at the virtual addresses that will be accessed by these instructions. (In Linux version, we allocate a block of memory whose virtual address is from 0x00900000 to 0x00907fff from heap of SDP, and then build all the PEB-related data structures in this allocated memory.)

Particularly, the FS register is set up to point to the virtual address 0x7ffdf000 (0x09000000 in Linux version) which is the base address of TEB data structure. According to the assembly instruction *mov eax, fs:[eax+0x30]*, the virtual address 0x7ffdf030 (0x00900030 in Linux version) will be accessed for reading the pointer of the *PEB* (This pointer is initialized to point to the base address 0x7ffd6000 (0x00901000 in Linux version) of *PEB* data structure). So we set a breakpoint at 0x7ffdf030 (0x00900030 in Linux version) for this instruction. The following breakpoints are set similarly (here we do not list all the virtual addresses). If all the virtual addresses which are set as breakpoints are accessed, we can confirm that the target network stream contains shellcode.

**False Negatives Evaluation** We use Metasploit to generate 10 kinds of shellcode payloads (give them sequence number #1-10 respectively). For each generated attack

 

instance, we mix it into 64KB randomly generated data, and use LanTraffic to send the corresponding TCP traffic. The TCP traffic is captured by NTCM of Shellix and finally sent to SDP for analysis.

Table 1 shows the results. By both Windows and Linux version of Shellix, the #1-4,#6,#7,#10 payloads can be successfully detected under all kind of encoders, the rest three payloads are missing.

By manually analyzing the missed three payloads, we find #5 and #9 payload use *SEH* to locate *kernel32.dll*, and the another one uses the hardcoded address of kernel32.dll. Excluding the detection fails due to the non-PEB-based shellcode, Shellix can successfully detect all the shellcode using different popular encoders listed in Table 1. Additionally, it is not hard to implement the heuristics for detecting the shellcode based on *SEH* and hard-coded.

We repeat this experiment for 10 times using different attack payloads. Shellix successfully detects all the shellcodes that use *PEB* to locate *kernel32.dll*. The experiments show that Shellix has a good detection effectiveness for popular obfuscated shellcode. In contrast, due to the inaccurate emulation of particular instructions, some CPU-emulator-based shellcode detection systems may fail to detect some obfuscated shellcode, e.g. Gene is failed to detect the shellcode using *ulpha_upper* encoder [2]. This situation is almost unlikely happened in Shellix, since Shellix is directly based on the hardware. Moreover, Shellix successfully detects the VM-aware shellcode. That means the virtualization-aware shellcode can hardly discover the existence of our system. To our best knowledge, Shellix is the first shellcode detection system that has this feature.

**False Positives Evaluation** To test the false positives of Shellix, we use a program to generate the random binary data that mimic benign network streams. We totally generate 50GB random data that consist of about 0.5 million streams. The length of each stream is also randomized, ranging from 4KB to 1024KB. These streams are scanned by Shellix, and no one stream triggers all the breakpoints set according PEB heuristic sequentially. Specifically, 3 streams trigger the No.1 breakpoint (corresponding to *mov eax, fs:[eax+0x30]*), no one stream triggers the No.1 and No.2 breakpoint(corresponding to *mov eax, [eax+0x0C]*) sequentially. We repeat this experiment for 10 times using different random binary data. The false positive rate is still zero (the results of both Windows version and Linux version are the same using the same data).

### 5.2. Performance

**Timer Interrupt Frequency (TIF) Selection** As stated in Section 4.4, both too low and too high TIF are not beneficial to achieve high performance. So, in order to find the proper timer interrupt frequency, we let Shellix execute the same generated data stream in different TIF.

Shellix is configured to execute the same 50MB data and record the time overhead in different TIFs. The experiment repeats 10 times using different data. The time overhead in each TIF is the average value of the 10 times experiments.

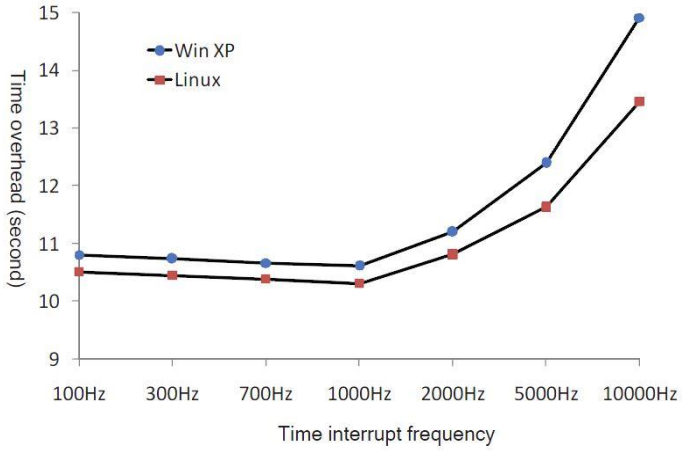


**Figure 6. Time Overhead in Each TIF**

Figure 6 shows the result. The time overhead is lowest in 1000Hz and gradually increases on both sides. Comparing the time overhead in 100Hz, 1000Hz and 10000Hz, the overhead growth between 1000Hz and 10000Hz is much higher than the one between 1000Hz and 100Hz. It indicates that the performance of Shellix is not sensitive to the variation of execution threshold ranging from 100Hz to 1000Hz (As the TIF increases, the number of instructions executed per execution chain decreases.). According to this feature, we can set the TIF slightly lower in practical use to execute more instructions in each execution chain. This is more effective to defeat the shellcode encoder like TAPiON [15] which is used to bypass the shellcode detection systems by placing many loops that make the detection system exceed the execution threshold before executing the real shellcode.

**Runtime performance** For the best runtime performance, we let Shellix execute the generated random binary data in 1000Hz TIF. Also, Shellix is configured to execute every execution chain without setting upper limit number of execution chains. In addition, we make Shellix execute the data that only contain *0xcc* (It means that every execution chain will only execute one instruction *int 3*) to test the highest throughput of Shellix in theory.

Figure 7 shows the runtime throughput rate of Shellix. For random binary data, the top throughput rate (in 1000Hz TIF) could reach the value of 39.8Mbps in Windows version and 40.8Mbps in Linux version. The theoretical highest throughput rates are 59.7Mbps and 61.2Mbps respectively. This performance is as good as most CPU-emulator-based shellcode detection systems. In most cases, it should be higher than them.

Gene [3] is an influential CPU-emulator-based shellcode detection system in recent years. It performances well in detecting shellcode and could achieve a good throughput rate. We intend to choose Gene for comparison. Regrettably, we are not able to obtain this system from its authors for direct comparison.

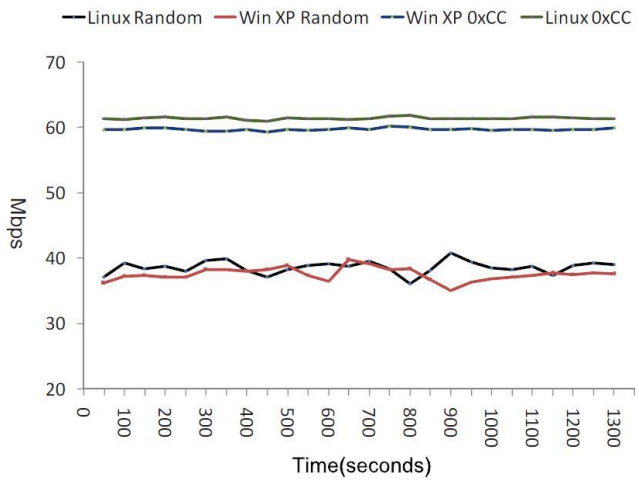


**Figure 7. Runtime Performance**

However, according to the evaluation of Gene [3], we could know that, in the all traffic mode, the throughput rate of Gene is about 22Mbps under 4K instruction execution threshold [3], it is lower than Shellix (see Figure 7). Moreover, with the growth of instruction execution threshold, Gene's throughput rate decreases obviously (about 8Mbps under 128K instruction execution threshold) [3]. In fact, most CPU-emulator-based shellcode detection systems have this negative characteristic [1, 3]. For high-level throughput rate, some detection systems set the execution threshold about 2k instructions for practical use [1]. This might make some obfuscated shellcode, whose complete decryption requires executing much more instructions (even more than 10000 instructions) [1], bypass the detection.

However, the throughput rate of Shellix changes very little and has a higher throughput rate than most of CPU-emulator-based shellcode detection systems in high-level instruction execution threshold (low TIFs, ranging from 100Hz to 1000Hz, see Figure 6)(according to paper [2], in 1000TIF, the mainstream CPU could execute at least about 60k instructions per tick). Please note that we do not employ any optimization methods, such as setting upper limit number of execution chains needed to be executed in one network stream, for safety. If some optimization methods are utilized, the throughput rate of Shellix could be higher.

## 6. Discussion

As evaluation results show, Shellix could work as well as the similar shellcode detection systems based on virtualization technology. Unlike the previous approaches which need to install a CPU emulator or virtual machine whose installation is very complicated, Shellix just needs to load a kernel module or driver, and then runs the detection program. So it is much easier for wide deployment. Furthermore, by utilizing reverse engineering, we can develop other Windows versions of Shellix (*e.g.*, Windows 7 x86) to make it more compatible with the recent commodity OS.

Like other shellcode detection systems [2], Shellix also has its inherited limitations. One is that the accurate instruction execution threshold may not be well determined. We can only judge the infinite loop from recording the number of timer interrupt that the execution chain has elapsed. In our future work, we plan to explore more efficient method for exiting from loops. Additionally, we will implement more shellcode detection heuristics for Shellix so that it could detect more kinds of shellcode.

## 7. Related Work

The existing approaches to detect shellcode could be grouped into two categories: static analysis [16-25] and dynamic analysis [1, 3, 26-28, 2, 4, 29-32].

Static analysis approaches make use of the compiler techniques to disassemble the network stream to get the instructions, and then compare them with the signatures generated from known shellcode. These methods can detect the shellcode efficiently, but are limited to detect the obfuscated (*e.g.*, polymorphic and metamorphic) shellcode.

To overcome the weakness of static analysis, Polychronakis *et. al.*, [1] firstly propose the dynamic analysis approach to uncover the self-modifying shellcode by using a CPU emulator to execute the network stream as binary code. This approach is very effective to detect polymorphic shellcode. Based on this idea, many improved approaches have been proposed. Zhang *et. al.*, [26] propose an approach that integrates static and dynamic analysis technique to achieve high detection performance. Fratantonio *et. al.*, [4] present a dynamic shellcode analyzer, called Shellzer. This approach utilizes the CPU debug mode which allows the shellcode being executed step by step. In this way, it can generate a complete list of the Windows invocation API functions called by the shellcode. In addition, many researchers apply dynamic analysis approach to detect "heap-spray" attacks which are very popular to compromise web browsers [29-32].

Snow *et. al.*, [2] propose a shellcode detection system, called ShellOS, which is quite different from previous dynamic analysis approaches. They observe that most of previous dynamic analysis approaches employ software-based CPU emulators. These emulators could not emulate the real CPU instructions very well, so they are easy to be detected by shellcode. Moreover, executing the instructions on the CPU emulator is very slow. To address this problem, ShellOS relies on the hardware-assisted virtualization technology, which allows most of instructions directly executing on the real CPU. However, ShellOS is still vulnerable to detect the virtualization-aware shellcode, because some special instructions still need to be emulated by software. Recently, Abbasi *et. al.*, [8] propose a number of techniques that evade emulated-based shellcode detection system (EBSDS). They point out that any emulation gaps should lead to the exposure of emulator to the attackers, and summarize the limitations of emulator in two aspects: a) Unsupported Instruction: Most EBSDSs do not have the capabilities of emulating full instruction set. For example, some instructions, such as FPU, MMX, SSE instructions, are out of the instructions set they can emulate. b) Emulator Detection: When executing instructions, emulators have some behaviors which are quite different from real machine, such as initializing all the eight general purpose registers in a same value. For these defects, Abbasi et al. propose several virtualization-aware technology and create a corresponding shellcode that uses these technology which firstly detect the virtualization environment before the effective part of shellcode get executed.

## 8. Conclusion

In this paper, we present a new shellcode detection system, called Shellix, to achieve efficient and reliable shell-code detection. We take advantage of the existing OS kernel and do not employ any virtualization technology. This is very useful to detect virtualization-aware shellcode. Moreover, our system is good for wide deployment. Finally, the evaluation result shows that the performance of our approach is as good as the approaches based on the virtualization technology.

## Acknowledgments

This work was supported in part by the National High Technology Research and Development Program of China (863 Program) under Grant No. 2009AA01Z433; the Project of National Ministry under Grant No. A21201-10006; the Open Foundation of

 

State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences) under Grant No. 2013-4-1; and the Cyber and Information Security Laboratory, CAEP under Grant No. J-2014-KF-01.

## References

[1] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Network-level Polymorphic Shellcode Detection using Emulation", Detection of Intrusions and Malware and Vulnerability Assessment, **(2006)**, pp. 54–73.
[2] K. Z. Snow, S. Krishnan, F. Monrose and N. Provos, "ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attack", USENIX Security Symposium, **(2011)**.
[3] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Comprehensive Shellcode Detection Using Runtime Heuristics. Annual Computer Security Applications Conference", **(2010)**, pp. 287–296.
[4] Y. Fratantonio, C. Kruegel and G. Vigna, "Shellzer: A Tool for the Dynamic Analysis of Malicious Shellcode", Proceedings of the 14th International Symposium on Recent Advanced in Intrusion Detection (RAID), **(2011)**.
[5] L. Martignoni, R. Paleari, G. F. Roglia and D. Bruschi, "Testing CPU Emulators. International Symposium on Software Testing and Analysis", **(2009)**, pp. 261–272.
[6] R. Paleari, L. Martignoni, G. F. Roglia and D. Bruschi, "A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators", USENIX Workshop on Offensive Technologies, **(2009)**.
[7] T. Raffetseder, C. Kruegel and E. Kirda, "Detecting System Emulators. Information Security", vol. 4779, **(2007)** pp. 1–18.
[8] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon and S. Etalle, "On Emulation-Based Network Intrusion Detection Systems", Proceedings of the 17th International Symposium on Recent Advanced in Intrusion Detection (RAID), **(2014)**.
[9] A. Vasudevan and R. Yerraballi, "Stealth Breakpoints. 21st Annual Computer Security Applications Conference", **(2005)**, pp. 381–392.
[10] T. Karagiannis, A. Broido, M. Faloutsos and K. Claffy, "Transport Layer Identification of P2P Traffic", ACM/SIGCOMM IMC, **(2004)**.
[11] Q. Sun, S. Li and X. Guo, "Quick Finding of Network Video Stream. International Conference on Computer Science and Information Technology", **(2008)**.
[12] Linux Kernel 2.6.32, https://www.kernel.org/pub /linux/kernel/v2.6/.
[13] LanTraffic V2 Enhanced, http://www.zti-teleco m.com/.
[14] The Metasploit Project, http://www.metasploit.com/.
[15] P. Bania, TAPiON, **(2005)**. http://pb.specialised.inf o/all/tapion/.
[16] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. Li, R. Kuo and K.-P. Fan, "Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities", Proceedings of the Network Operations and Management Symposium (NOMS), **(2004)**, pp. 235–248.
[17] R. Chinchani and E. van der Berg, "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows", Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), **(2005)**.
[18] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution", Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), **(2002)**.
[19] J. Newsome, B. Karp and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms", Proceedings of the IEEE Security and Privacy Symposium, **(2005)**, pp. 226–241.
[20] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns", Usenix Security Symposium, **(2003)**.
[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song and R. E. Bryant, "Semantics-aware Malware Detection", IEEE Symposium on Security and Privacy, **(2005)**.
[22] X. Wang, Y.-C. Jhi, S. Zhu and P. Liu, "STILL: Exploit Code Detection via Static Taint and Initialization Analyses", Annual Computer Security Applications Conference, **(2008)**, pp. 11–20.
[23] C. Kruegel, W. Robertson and G. Vigna, "Detecting Kernel-Level Rootkits Through Binary Analysis", Annual Computer Security Application Conference (ACSAC), **(2004)**.
[24] K. Wang and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection", Proceedings of the 7th International Symposium on Recent Advanced in Intrusion Detection (RAID), **(2004)**, pp. 201–222.
[25] M. Egele, M. Szydlowski, E. Kirda and C. Kruegel, "Using Static Program Analysis to Aid Intrusion Detection. Detection of Intrusions and Malware & Vulnerability Assessment", **(2006)**.
[26] Q. Zhang, D. S. Reeves, P. Ning and S. P. Iyer, "Analyzing Network Traffic to Detect Self-Decrypting Exploit Code", ACM Symposium on Information, Computer and Communications Security, **(2007)**.
[27] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "An Empirical Study of Real-world Polymorphic Code Injection Attacks", USENIX Workshop on Large-Scale Exploits and Emergent Threats, **(2009)**.

 

[28] B. Gu, X. Bai and Z. Yang, A. C. Champion and D. Xuan, "Malicious Shellcode Detection with Virtual Memory Snapshots", International Conference on Computer Communications (INFOCOM), **(2010)**, pp. 974–982.
[29] P. Ratanaworabhan, B. Livshits and B. Zorn, "A Defense Against Heapspraying Code Injection Attacks", USENIX Security Symposium, **(2009)**, pp. 169–186.
[30] C. Curtsigner, B. Livshits, B. Zorn and C. Seifert, "Zozzle: Fast and Precise In-Browser Javascript Malware Detection. USENIX Security Symposium", **(2011)**.
[31] M. Cova, C. Kruegel and G. Vigna, "Detection and Analysis of Drive-by-download Attacks and Malicious Javascript Code. International conference on World Wide Web", **(2010)**, pp. 281–290.
[32] M. Egele, P. Wurzinger, C. Kruegel and E. Kirda, "Defending Browsers against Drive-by downloads: Mitigating Heap-spraying Code Injection Attacks", Detection of Intrusions and Malware & Vulnerability Assessment, **(2009)**.
[33] KVM - Kernel-based Virtual Machine, http://www.redhat.com/resourcelibrary/whitepa pers /doc- kvm.

## Authors

**Mo Chen**, is currently an engineer in China Aerospace Consultation Center, Beijing, China. He received the B.S. degree in the School of Computer, Beijing Institute of Technology. His research interest lies in operating system, virtualization technology.

 