IEICE TRANS. INF. & SYST., VOL.E96–D, NO.10 OCTOBER 2013

LETTER

Efficient Shellcode Detection on Commodity Hardware**

Donghai TIAN[†,††a)], **Mo CHEN**[†,††*], **Changzhen HU**[†,††], *Nonmembers*, *and* **Xuanya LI**[†,††*], *Student Member*

SUMMARY As more and more software vulnerabilities are exposed, shellcode has become very popular in recent years. It is widely used by attackers to exploit vulnerabilities and then hijack program's execution. Previous solutions suffer from limitations in that: 1) Some methods based on static analysis may fail to detect the shellcode using obfuscation techniques. 2) Other methods based on dynamic analysis could impose considerable performance overhead. In this paper, we propose Lemo, an efficient shellcode detection system. Our system is compatible with commodity hardware and operating systems, which enables deployment. To improve the performance of our system, we make use of the multi-core technology. The experiments show that our system can detect shellcode efficiently.

key words: *efficient shellcode detection, multi-core technology*

1. Introduction

In order to exploit vulnerable programs, code-injection attacks are widely used. By utilizing the vulnerabilities, attackers could inject shellcode into the target process to change or even hijack the program's execution. To prevent the vulnerable programs from being exploited, it is important to filter out the network data that contains shellcode.

In the past few years, many detection approaches [1], [3], [4], [6], [9] have been proposed. Basically, these methods can be divided into two categories: static analysis [3], [4] and dynamic analysis [1], [6], [9]. The core idea of static analysis is to disassemble the network stream and then analyze the code-level patterns that could be signatures obtained from existing shellcode. This approach is efficient to identify known exploit code. However, it is limited in detecting shellcode that employs binary obfuscation techniques. To address this problem, Polychronakis et al. propose the dynamic methods based on the network-level emulation [1]. This method first disassembles network data into several execution chains and then emulates these chains. Compared with static analysis methods, dynamic analysis methods can achieve better detection completeness and accuracy. Nevertheless, this approach imposes considerable performance overhead. Moreover, it is easy for the shellcode to detect the emulated environment so that this dynamic method could be evaded [2].

Recently, Snow et al. proposed a new framework, ShellOS [5], enabling fast detection and forensic analysis of shellcode. Although ShellOS has made great progress on dynamic methods, it still has some limitations as follows: 1) Since ShellOS relies on the hardware assisted virtualization technology, it is still possible for the shellcode to detect the presence of the virtualized environment and then hide its malicious behavior. 2) ShellOS is built upon the KVM hypervisor. As a result, the performance will be affected due to the interactions between the VM and hypervisor. 3) ShellOS is a special OS kernel developed from scratch, so it is not easy to widely deploy. 4) ShellOS does not make full use of multi-core technology.

In this paper, we present a prototype system, Lemo, which leverages commodity hardware and operating systems to achieve efficient shellcode detection. Lemo is implemented on the Linux kernel, and it does not need the support of hardware assisted virtualization. Our approach makes the following contributions:

- We propose an efficient shellcode detection approach based on commodity hardware. Moreover, our approach only requires minimal changes to the commodity OS kernel.
- We leverage the multi-core technology to achieve high performance. To deal with the synchronization problem, we utilize Lamport's ring buffer algorithm [8].
- We design and implement a prototype of Lemo based on Linux. The evaluations show that our system can detect shellcode with good performance.

2. Overview of Our Approach

The goal of Lemo is to build a system that achieves efficient shellcode detection based on the existing hardware and operating system. Our high level idea is consistent with previous dynamic analysis methods: just to execute the network stream on the CPU and then check the presence of the shellcode in the consequent instruction sequences. Since the first instruction of the shellcode is not known in advance, Lemo is required to execute from each offset of the network

Manuscript received January 30, 2013.
Manuscript revised May 13, 2013.

[†]The authors are with the School of Software, Beijing Institute of Technology, China.
[††]The authors are also with State Key Laboratory of Information Security (Institute of Information Engineering), Chinese Academy of Sciences, China.
[*]Presently, with the School of Computer, Beijing Institute of Technology, China.

[**]This work is supported partially by the National High-Tech Research Development Program of China under Grant No. 2009AA01Z433 and the Open Fundation of State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093).

a) E-mail: donghaitad@gmail.com

DOI: 10.1587/transinf.E96.D.2272

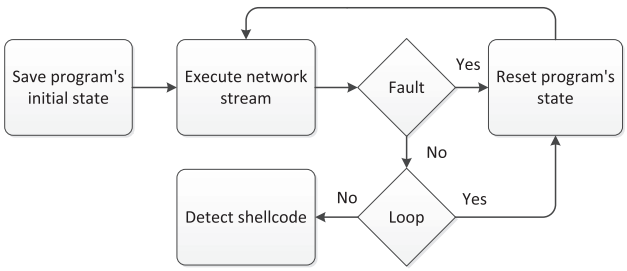

Fig. 1 System workflow.

stream. In other words, our system tries each byte of the network stream as a potential entry point of shellcode. Similar to the dynamic approaches [1], [6], [9], we refer to a complete execution from each offset of the stream as an execution chain. Compared with the dynamic approaches that require intercepting and tracking each instruction, our approach allows instruction sequences being executed directly on the CPU, and only trap a small number of instructions selectively by utilizing hardware paging mechanisms. Therefore, the impact on system throughput will be minimized using our approach.

The general workflow of our system is illustrated in Fig. 1. First, Lemo saves the detection program's initial memory state for later recovery. Then, the detection process executes the network stream directly on the CPU. Since the network stream may contain random instructions (e.g., privileged instructions), executing them on the user mode could cause system faults. In contrast, the shellcode will not contain such instructions. Therefore, the system fault could indicate the start location of the shellcode in the network stream is not correct. To run the next execution chain, we reset the detection program's state by restoring the previously saved memory content. Furthermore, our detection program may execute the network stream for a long time due to the accidental formation of loop instructions. To detect and terminate the loop execution quickly, we utilize the timer interrupt. Next, we reset the program's state for the next execution chain. After the execution chains containing faulty and loop instructions are filtered, we make use of the hardware paging mechanism to detect shellcode.

3. System Design and Implementation

We have developed Lemo, a prototype based on Linux kernel (with version 2.6.32) to demonstrate our approach. We provide the method to handle system faults (or exceptions) in Sect. 3.1. The shellcode detection approach is described in Sect. 3.2. We present the parallel architecture in Sect. 3.3.

3.1 Fault Handling

As mentioned previously, executing the network streams directly on the CPU may result in system faults (or exceptions) due to the random instructions within the streams. Moreover, running these random instructions may corrupt the normal data of our detection process. Therefore, we should

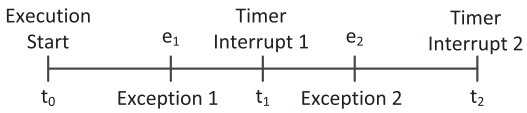

Fig. 2 Execution timeline.

adopt a mechanism to recover the corrupted data. Before recovering the data, the process' initial clean state must be first saved in the kernel space so that the user code cannot touch the data. To minimize the size of our saved memory, we only need to preserve the writable pages of our detection process. Relying on the `vm_area_struct` structure of the task descriptor (i.e., `task_struct`), we can locate the pages to be saved.

To get a chance for storing the process' initial state, we should somehow make the process' execution trap into the OS kernel before executing the network data. For this purpose, we insert the INT 3 instruction into the front of the network stream. After the INT 3 instruction is executed, the debug exception will be raised. Accordingly, we hook the exception handler `do_debug` to save the memory content of the detection process. For preserving the current CPU registers, we leverage the fact that the OS kernel saves registers of the user program on the kernel stack before entering fault handlers. In Linux, we can locate these registers via a data structure `pt_regs`. Then, the execution control is transferred from the OS kernel to our detection program.

To execute the network stream in our detection process, we set a function pointer to the address of the program's buffer that contains the network data. Next, we invoke this function to change the program's execution to the network stream. When this execution encounters an instruction that causes a system fault, we recover the program's normal data by restoring the previously saved memory content and CPU registers. Moreover, we change the program counter stored on the stack so that our detection process will start the next execution chain once the execution control is returned to the user mode.

In some execution chains, our detection process may encounter a return instruction so that the function executing the network stream will return and exit. To address this problem, we modify the program counter located on the stack to the address of kernel space. In this way, an exception will happen after the function returns. Then, we hook the corresponding fault handler for the next execution chain.

In addition, we have to deal with the execution chain that contains infinite loops. To abort the loop execution as soon as possible, we make use of the timer interrupt. In the multi-core environment, the APIC timer is used for maintaining the time activities related to a specific CPU (e.g., how long the current process is running). For the implementation, we hook the timer interrupt handler `apic_timer_interrupt()` to check if the running time of the execution chain exceeds the predefined limit and then perform the associated operations.

Specifically, we need to handle situations shown in

Fig. 2. Before starting a new execution chain, we record the tick counter (i.e., `jiffies`) at time t_0. Then, an exception may occur at time e_1. Accordingly, we recover the program's previous state and then change the program counter to the start point of the next execution chain. Also, the tick counter is incremented. Next, a timer interrupt is raised at time t_1. To determine whether the execution is in the loop, the current tick is compared with our tick counter. If the current tick is larger than the tick counter, the program's previous state is restored. Moreover, we increase the tick counter and start a new execution chain. On the other hand, if the current tick is equal to our tick counter at time t_1, we will do the similar work at time e_2 or t_2.

It is worth noting that if we want to exit infinite loops quickly, the timer frequency should be set higher. However, doing so would also increase the performance overhead due to the world switch between the user process and OS kernel.

3.2 Shellcode Detection

To set up an execution environment for shellcode, our system needs to mimic a Windows environment. In general, Win32 code expects the Process Environment Block (PEB) to be present in its execution environment. This Windows internal structure contains a variety of process-specific information, such as current structured exception handling frame, Win32 thread information and so on. In order to invoke user-level Windows API, shellcode has to first resolve the API addresses that could be acquired via the PEB. In the Windows environment, a pointer to the PEB is stored in the FS register. Setting up this segment register requires modifying the process local descriptor table (LDT). In our implementation, we first allocate a chunk of memory for the PEB. Then, we utilize the system call (i.e., `modify_ldt()`) to set up the FS register.

In order to detect shellcode in our execution environment, we employ the PEB heuristic [6]. This detection heuristic relies on the conventional behaviour of shellcode: After executing the network string, the shellcode will first read the memory address of FS:[0x30] to locate the PEB, and then read the `LoaderData` field (0xc bytes offset) of the PEB to locate the `PEB_LDR_DATA` structure. Based on this structure, the shellcode will walk through the loaded modules list to resolve the base address of `kernel32.dll`.

To trap the memory operations on the PEB during the shellcode's execution, we make use of the stealth breakpoint technique [7]. Specifically, we clear the present bit of the page table entry (PTE) mapping for the addresses that are associated with the PEB. As a result, when the shellcode try to access the PEB, it will cause a page fault. Accordingly, we hook the page fault handler (i.e., `do_page_fault()`) in Linux kernel to check whether the operation hits our predefined breakpoints. If it is, the handler sets the present bit in the PTE such that the corresponding memory page can be accessed. To ensure trapping future access to this page, the handler has to enable the single step debug mechanism for reseting the present bit in the PTE and then disable the debug mechanism.

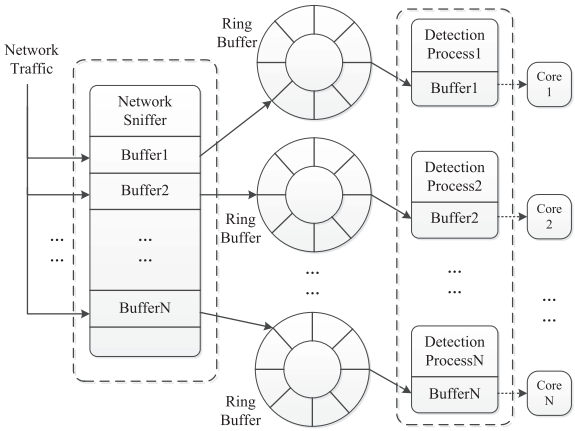

Fig. 3 Parallel architecture.

3.3 Parallel Architecture

To improve the performance of our system, we leverage the increasingly popular multi-core technology. As shown in Fig. 3, our system can be divided into two parts: Network Sniffer and Detection Processes. Relying on different CPU cores, these two components can work in parallel. Specifically, the Network Sniffer is responsible for capturing network packets. Before transferring these network stream to the Detection Processes, the Network Sniffer has to assemble the TCP stream in that attackers may disperse their shellcode across different IP packets. After that, the Detection Processes perform the TCP stream analysis to inspect the presence of shellcode. By applying the multi-core technology, the Network Sniffer and Detection Processes are mapped to different CPU cores.

To dispatch network packets to different buffers of Detection Processes, we employ a segmentation scheme similar to a previous study [12]. Specifically, we divide the network string into continuous overlapping segments, and the length of the overlapped part between any two neighboring segments equals to the maximum length of shellcode, which we assume is 2048 bytes. In this way, any shellcode that is less than 2048 bytes will be included in at least one segment.

In order to deal with the synchronization problem between the Network Sniffer and Detection Processes, we utilize the Lamport's ring buffer algorithm [8], which is shown in Fig. 4. The ring buffer is located in the OS kernel space so that its TCP data cannot be corrupted when executing the network stream.

The specific definition of the ring buffer structure is illustrated in Line 1∼7. The reason of containing the 52 bytes padding field (pad[52]) in this structure is to make the data aligned in the first-level hardware cache. By doing so, our system performance could be improved. To ensure concurrent operations on the ring buffer without using locks, the producer (i.e., Network Sniffer) needs to invoke the `Enqueue` function to insert the network data into the buffer

```
1  struct ring_buffer{
2      unsigned long write;
3      unsigned long read;
4      unsigned long size;
5      unsigned char pad[52];
6      struct element data[1];
7  };
8  static inline unsigned long Next(unsigned long
       index, unsigned long size){
9      return (index+1) % size;
10 }
11 void Enqueue(struct ring_buffer *sring, struct
       element *tcp_data){
12     struct element *data;
13 loop1:
14     if( next(sring->write, sring->size) ==
           sring->read )
15         goto loop1;
16     data = sring->buffer + sring->write;
17     memcpy(data, tcp_data);
18     sring->write = Next(sring->write, sring->
           size);
19 }
20 void Dequeue(struct ring_buffer *sring){
21     struct element *data;
22 loop2:
23     if( sring->write == sring->read )
24         goto loop2;
25     data = sring->data + sring->read;
26     Analyze data;
27     sring->read = Next(sring->read, sring->
           size);
28 }
```

Fig. 4 Lamport's ring buffer algorithm.

(Line 11~19). On the other hand, the consumer (i.e., Detection Process) has to invoke the `Dequeue` function to consume the data from the buffer (Line 20~28). In particular, the Detection Process analyzes the data copied from the ring buffer in its user address space (Line 26).

Once the detection process finishes the TCP data analysis, its execution is transferred to the OS kernel. Then, we update the TCP data in the Detection Process' buffer for the next analysis. To reduce the packet loss of the Network Sniffer, different ring buffers are allocated for different Detection Processes. Moreover, when one of these ring buffers is fully filled with the TCP data, the Network Sniffer will choose the other one for transferring the data so that the Sniffer could continue capturing network packets without being blocked.

4. Evaluation

To test the capabilities of Lemo, we conduct a series of experiments and throughput measurements. All the experiments are carried out on a Dell PowerEdge T410 work station with a 2.13 G Intel Xeon E5606 CPU and 4 GB memory.

4.1 Effectiveness

We evaluate the effectiveness of Lemo for shellcode detection with Metasploit Framework [10]. Particularly, we select 8 classical exploits that make use of the PEB for Windows API resolution as our experimental samples. For each sample, we apply the advanced polymorphic engines within Metasploit to generate 700 different shellcode instances. We feed these samples to our detection process for network data analysis. The experiments show that all these 5600 shellcode instances are successfully detected by our system.

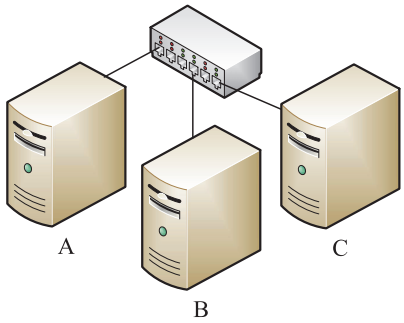

Fig. 5 Experimental testbed.

4.2 Throughput

To evaluate the network throughput of our system, we set up a test bed consisting of 3 machines, which is shown in Fig. 5. All these machines are connected by a 100 MB hub. To generate the TCP network traffic, we make use of LanTraffic [11]. Specifically, the LanTraffic sender runs on the machine A while the LanTraffic receiver is located on the machine C. The length of TCP data is configured as 64 KB. Our detection system runs on the machine B. We set the timer frequency to 1000 HZ. All the execution chains in the TCP stream are fully analyzed. The evaluations show that our system is able to achieve 38.6 Mbps network throughput with one CPU core. When we utilize 2 CPU cores for two different detection processes, our system can achieve 73.8 Mbps network throughput without packet loss. Compared with the emulation-based methods [1], [6], [9], the network throughput is greatly improved.

5. Conclusion

In this paper, we present Lemo, an efficient shellcode detection system based on commodity hardware. We exploit the fault (exception) handling mechanisms of commodity operating systems to analyze network streams by executing them directly on the CPU. Moreover, we leverage the multi-core technology to improve the system performance. Our evaluations show that Lemo can detect shellcode effectively with good performance.

References

[1] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Network-level polymorphic shellcode detection using emulation," 3rd International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2006.
[2] R. Paleari, L. Martignoni, G.F. Roglia, and D. Bruschi, "A fistful of

red-pills: How to automatically generate procedures to detect CPU emulators," Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT), 2009.

[3] T. Toth and C. Krgel, "Accurate buffer overflow detection via abstract payload execution," 5th International Symposium on Recent Advances in Intrusion Detection (RAID), 2002.

[4] R. Chinchani and E. van den Berg, "A fast static analysis approach to detect exploit code inside network flows," 8th International Symposium on Recent Advances in Intrusion Detection (RAID), 2005.

[5] K.Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "SHELLOS: Enabling fast detection and forensic analysis of code injection attacks," Proc. 20th USENIX Security Symposium, 2011.

[6] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Comprehensive shellcode detection using runtime heuristics," Proc. 26th Annual Computer Security Applications Conference (ACSAC), 2010.

[7] A. Vasudevan, and R. Yerraballi, "Stealth breakpoints," Proc. 21st Annual Computer Security Applications Conference (ACSAC), 2005.

[8] L. Lamport, Proving the Correctness of Multiprocess Programs, IEEE Transactions on Software Engineering, 1977.

[9] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Emulation-based detection of non-self-contained polymorphic shellcode," 10th International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.

[10] Rapid7 LLC, Penetration Testing Software: Metasploit, http://www.metasploit.com/, 2012.

[11] ZTI, LanTraffic V2, http://www.zti-telecom.com/, 2012.

[12] L. Wang, H.-X. Duan, and X. Li, "Dynamic emulation based modeling and detection of polymorphic shellcode at the network level," Science in China Series F: Information Sciences, 2008.