Malicious Shellcode Detection with Virtual Memory Snapshots

Boxuan Gu, Xiaole Bai, Zhimin Yang, Adam C. Champion and Dong Xuan
Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
{gub,baixia,yangz,champion,xuan}@cse.ohio-state.edu

***Abstract*—Malicious shellcodes are segments of *binary* code disguised as normal input data. Such shellcodes can be injected into a target process's virtual memory. They overwrite the process's return addresses and hijack control flow. Detecting and filtering out such shellcodes is vital to prevent damage. In this paper, we propose a new malicious shellcode detection methodology in which we take *snapshots* of the process's virtual memory *before* input data are consumed, and feed the snapshots to a malicious shellcode detector. These snapshots are used to instantiate a runtime environment that emulates the target process's input data consumption to monitor shellcodes' behaviors. The snapshots can also be used to examine the system calls that shellcodes invoke, these system call parameters, and the process's execution flow. We implement a prototype system in Debian Linux with kernel version 2.6.26. Our extensive experiments with real traces and thousands of malicious shellcodes illustrate our system's performance with low overhead and few false negatives and few false positives.**

I. Introduction

A. Motivation

According to the US-CERT database [1], buffer overflows are one of the most critical and common software vulnerabilities. Attackers routinely exploit these vulnerabilities to inject malicious shellcode and gain control of hosts. Unlike self-contained pieces of malware, malicious shellcodes are segments of binary code disguised as normal input data. They are injected into a target process's virtual memory and hijack the process control flow. It is important to filter out messages that contain such shellcodes before they cause damage.

Many intrusion detection approaches have been proposed to detect such malicious shellcodes. Based on *whether* they check input data before consumption, we classify them into two categories: *d*etection *d*uring input data *c*onsumption-based approaches (*DDC*-based approaches for short) and *d*etection *b*efore input data *c*onsumption-based approaches (*DBC*-based approaches for short). DDC-based approaches, which conduct detection when the target process *actually* consumes the input data, include flow tracking [2]–[4], randomization [5]–[7], compiler extensions [8]–[13], OS extensions [14], [15] and hardware modifications [16]. In spite of their strong detection capabilities, these approaches are often heavyweight and they may not detect malicious shellcode in a timely manner [17]. DBC-based approaches, in which the input data are checked *before* the target process consumes them, can be subdivided into two categories: *static analysis* [18]–[21] and *dynamic analysis* [22], [23]. *Static analysis* uses code-level patterns to detect malicious shellcodes in input data whereas *dynamic analysis* conducts network-level emulation. A detailed discussion on these works is given in Section II.

In this paper, we focus on DBC-based approaches, which are lightweight. However, such existing approaches have limited detection capability. Attacks exploit knowledge of the target process's runtime information to evade detection by DBC-based approaches via techniques such as polymorphism and metamorphism. We will present two representative examples in Section II that illustrate these attacks. The first example uses runtime information generated by the target process and the second uses dynamic information generated by the attack on the fly. Existing DBC-based approaches fail to detect these attacks. In general, these approaches conduct detection by scanning input data but they rarely utilize the target process's runtime information. We aim to remedy this situation by introducing a new detection methodology to detect *malicious shellcodes*. We do *not* focus on detecting self-contained malware programs.

B. Our Contributions

We highlight our contributions in this paper as follows.

– We propose a new malicious shellcode detection methodology in which we take *snapshots* of the target process's virtual memory immediately before input data are consumed and feed these snapshots to a *lightweight* DBC-based malicious code detector that we design and implement.

– We propose using these snapshots to instantiate a runtime environment that emulates the target process's *input data consumption*. This environment facilitates monitoring shellcodes' behaviors. We use the snapshots to examine *system calls* invoked by (executable) input data and the parameters thereof as well as the process's execution flow to detect malicious shellcodes.

This work was supported in part by the US National Science Foundation (NSF) under grants No. CNS-0916584, CAREER Award CCF-0546668, and the Army Research Office (ARO) under grant No. AMSRD-ACC-R 50521-CI. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

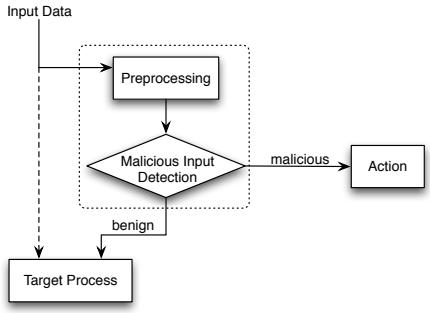

Fig. 1. In DBC-based intrusion detection approaches, the input data are checked before they are consumed by the target process. Such state-of-the-art approaches do not use the target process's runtime information.

– We implement a prototype system based on the above methodology in Debian Linux with kernel version 2.6.26. Our system is adaptive and extensible. It is designed to be loaded into the target process's address space statically or dynamically. Our system can efficiently fetch and use the target process's virtual memory snapshots for shellcode detection.

– We conduct extensive experiments based on real traces and thousands of malicious shellcode samples. The experimental results illustrate our detection's low runtime overhead as well as its high detection accuracy, which exceed those of state-of-the-art DBC-based approaches.

Note that unlike emulators such as QEMU [24] that emulate an entire OS and processes running therein, our system only emulates instructions decoded from input data. In addition, our detection methodology has broad adaptability. Other dynamic analysis techniques can be developed based on virtual memory snapshots and DBC-based static analysis approaches can apply our methodology to augment their shellcode detection capabilities.

Paper Organization. The rest of this paper is organized as follows. We introduce related work and its limitations in Section II. We present our system design in Section III. We present our implementation and evaluation thereof in Section IV. We conclude in Section V.

II. Related Work

Intrusion detection approaches can be classified as *D*etection *B*efore Input Data *C*onsumption-based (DBC-based) and *D*etection *D*uring Input Data *C*onsumption-based (DDC-based).

A. DBC-Based Approaches

In this intrusion detection approach category, input data are checked as *executables* before the target process consumes them as shown in Fig. 1. We subdivide this category into two subcategories: *static analysis* and *dynamic analysis*.

Static Analysis. In static analysis, input data are first disassembled and then screened via code-level pattern analysis and matching. Patterns can be complicated signatures or simple heuristics obtained from studying known malicious codes. In [18], Toth and Kruegel proposed identifying exploit code by detecting NOP sleds, an approach that the Snort IDS preprocessor also uses [25]. However, attacks can bypass this detection by either *not* including NOP sleds or by using polymorphic techniques [26]–[28]. Chinchani and van den Berg proposed a rule-based scheme in [26]. Wang et al. proposed SigFree [29] that checks if packets contain malicious codes using "push and call" patterns and the number of useful instructions in the longest possible execution chain.

```
 1  push esp            ; esp points here
 2  pop  eax            ; [eax] <- 0xBFFF0000
 3  sub  eax,0x2d2d2d6c ; [eax] <- 0x92D1D294
 4  sub  eax,0x5858557a ; [eax] <- 0x3A797D1A
 5  sub  eax,0x7a7a7a7a ; [eax] <- 0xBFFF02A0
 6  push eax
 7  pop  esp            ; [esp] <- 0xBFFF02A0
 8  and  eax,0x2321252d ; [eax] <- 0x20012020
 9  and  eax,0x44424242 ; [eax] <- 0x00000000
10  sub  eax,0x2d2d2d2d ; [eax] <- 0xD2D2D2D3
11  sub  eax,0x252d252d ; [eax] <- 0xADA5ADA6
12  sub  eax,0x655e6761 ; [eax] <- 0x48474645
13  push eax            ; M[0xBFFF029c] = "EFGH"
14  sub  eax,0x2d2d2d2d ; [eax] <- 0x1B1A1918
15  sub  eax,0x5e5e5e5e ; [eax] <- 0xBCBBBABA
16  sub  eax,0x78787879 ; [eax] <- 0x44434241
17  push eax            ; M[0xBFFF0298] = "ABCD"
...
```

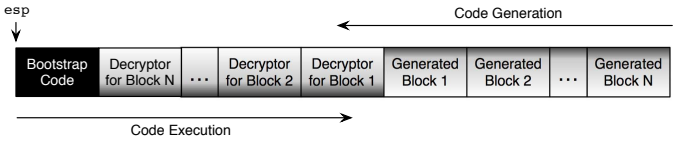

Fig. 2. Original Shellcode Execution Trace Produced by "Encode" Engine.

In general, static analysis is efficient. However, it has limitations regarding accuracy and completeness. In [30], Bayer et al. point out that, in general, determining program behavior via static analysis is undecidable, and binary obfuscation often effectively thwarts both the disassembly and code analysis steps.

Dynamic Analysis. Instead of static code-level patterns, dynamic analysis detects malicious input data by exploiting information generated during program execution on those data. The state-of-the-art dynamic analysis approach is network-level emulation, which Polychronakis et al. proposed [22], [23]. Network-level emulation disassembles input data into several possible execution chains and then emulates the execution of each chain. If any chain exhibits malicious behavior during emulation, the input data are classified as malicious.

Even though network-level emulation can achieve better detection completeness than static analysis, it is still prone to evasion. The problem is that it has insufficient context information about the target process and thus it must make assumptions during initialization and execution emulation. We will introduce two representative examples that can bypass it.

Before we give the examples, we provide some background information to aid understanding. Our examples are derived from shellcodes that are produced by an adapted version of the "Encode" shellcode engine [31]. The attack mechanism of such shellcode is illustrated in Fig. 2. The shellcode exploits a stack buffer overflow to which `esp` points after the shellcode is injected into the target process's address space. After activation, this shellcode decrypts a series of encrypted payloads, each of which contains 4 bytes. Due to the stack's LIFO nature, the decrypted payload is generated *backwards*, starting with its *last* four bytes. When the final decrypted block is pushed

```
 1  push esp            ; esp points here
 2  pop  eax            ; [eax] <- 0xBFFF0000
 3  sub  eax,0x2d2d2d6c ; [eax] <- 0x92D1D294
 4  sub  eax,0x5858557a ; [eax] <- 0x3A797D1A
 5  sub  eax,0x7a7a7a7a ; [eax] <- 0xBFFF02A0
 6  push eax
 7  pop  esp            ; [esp] <- 0xBFFF02A0
 8  and  eax,0x2321252d ; [eax] <- 0x20012020
 9  and  eax,0x44424242 ; [eax] <- 0x00000000
10  mov  ebx,0xa0ef(ebp); 0xa0ef is known
11  cmp  ebx,0x252d252d ; compare [ebx], 0x252d252d
12  jz   +3             ; conditional jump
13  lidt eax            ; privileged instruction
14  sub  eax,ebx        ; [eax] <- 0x48474645
15  push eax            ; M[0xBFFF029c] = "EFGH"
16  add  ebx,0x04350435
17  add  ebx,0x04cb04cb
18  cmp  ebx,0x2d2d3d3d ; compare [ebx], 0x2d2d2d2d
19  jz   +3             ; conditional jump
20  lidt eax            ; privileged instruction
21  sub  eax,ebx        ; [eax] <- 0x44434241
22  push eax            ; M[0xBFFF0298] = "ABCD"
```

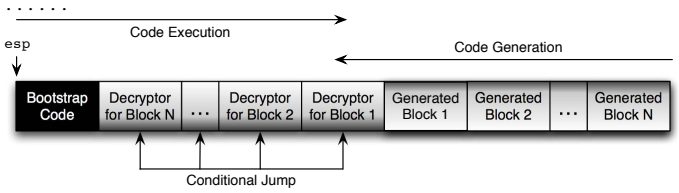

Fig. 3. Shellcode for Example 1.

```
 1  push esp            ;
 2  pop  eax            ; [eax] <- 0xBFFF0000
 3  sub  eax,0x2d2d2d6c ; [eax] <- 0x92D1D294
 4  sub  eax,0x5858557a ; [eax] <- 0x3A797D1A
 5  sub  eax,0x7a7a7a7a ; [eax] <- 0xBFFF02A0
 6  push eax
 7  pop  esp            ; [esp] <- 0xBFFF02A0
 8  and  eax,0x2321252d ; [eax] <- 0x20012020
 9  and  eax,0x44424242 ; [eax] <- 0x00000000
```

Message 1

```
10  sub  eax,0x2d2d2d2d ; [eax] <- 0xD2D2D2D3
11  sub  eax,0x252d252d ; [eax] <- 0xADA5ADA6
12  sub  eax,0x655e6761 ; [eax] <- 0x48474645
13  push eax            ; M[0xBFFF029c] = "EFGH"
```

Message 2

```
14  sub  eax,0x2d2d2d2d ; [eax] <- 0x1B1A1918
15  sub  eax,0x252d252d ; [eax] <- 0xBCBBBABA
16  sub  eax,0x655e6761 ; [eax] <- 0x44434241
17  push eax            ; M[0xBFFF029c] = "ABCD"
```

Message 3

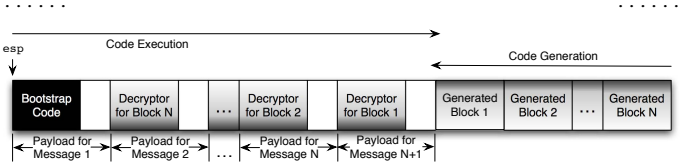

Fig. 4. Shellcode for Example 2.

onto the stack, the shellcode's control flow "meets" the newly built payload and the control flow is redirected to the decrypted instructions. Though is difficult for static analysis to detect such attacks, network-level emulation can detect them [23]. Now we are ready to introduce the examples.

– *Example 1:* Fig. 3 shows the IA-32 assembly language code for the first type of attack. Assume the attacker has some knowledge of the target process runtime information. More specifically, he knows that a constant value is stored somewhere in memory that can be used as well as the value in `ebp`, which is true for almost all stack-based buffer overflows [32]. After zeroing `eax` (in instruction 9), the decryption block process begins, again using separate decryption blocks (10–15, 16–21, . . .) for each four bytes of the encrypted payloads. However, the decryption blocks in this trace rely on a constant value stored in address `0xa0ef + (ebp)`. This value is put into `ebx` (instruction 10), and then compared with a desired value (instruction 11). If this value differs from the one the attacker wants, decryption stops and a privileged instruction is executed (instruction 13). Otherwise, decryption continues. As network-level emulation does not have *real* runtime information about the target process, all eight (emulated) general-purpose registers are initialized to hold the absolute address of the first instruction of each execution chain. An incorrect `ebp` will result in the emulated execution of instruction 13. Since `lidt` is a privileged operation, which malicious shellcodes should not contain, emulation will stop and consider the execution chain containing this instruction benign.

– *Example 2:* Chung and Mok proposed a new type of attack, the *swarm attack*, which can defeat existing DBC-based approaches that detect malicious shellcodes *one message at a time* [32]. In swarm attacks' methodology, shellcodes can be injected via *several* messages instead of *one* message. The authors point out that it is possible to create the decoder inside the attacked process's address space using multiple instances thereof with each instance of the attack writing a small part of the decoder at the designated location [32]. Fig. 4 shows the IA-32 assembly language code of an attack using this methodology. In this case, the entire shellcode is separated into several blocks, each of which (except the first) is a decryptor and can *only* generate *one* new instruction if it is executed. If the entire shellcode is sent to a target process using *one* message, then network-level emulation will easily detect it [23] because the number of newly generated instructions during emulation of the malicious shellcode will exceed a threshold that is set in advance. According to the swarm methodology, if each block is injected into a target process using one message via one attack instance, the entire malicious shellcode can evade detection by network-level emulation because the threshold for the number of newly generated instructions during the emulation of a shellcode is too large [22], [23].

The above two examples represent two types of attacks that can thwart state-of-the-art DBC approaches. The first type exploits runtime information generated by the *target process*. The second one uses information generated by the *previous attacks*.

B. DDC-Based Approaches

In this intrusion detection approach category, detection is conducted *while* processes consume input data. If malicious runtime behavior is detected, the process often stops and sounds an alarm. The input data and the log of target process states are stored for further analysis. Flow tracking [2]–[4] uses a taint-based method to check whether the input data are malicious. Randomization can effectively defend against attacks exploiting memory errors through address space randomization (ASR)

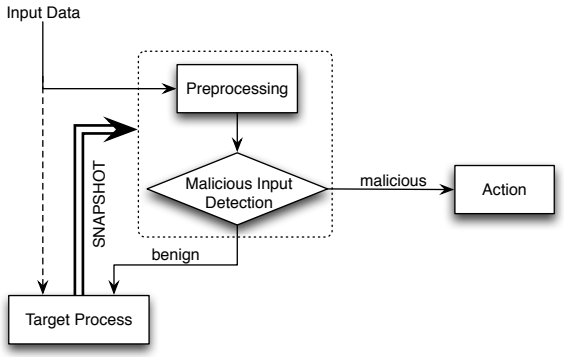

Fig. 5. We propose a new DBC-based intrusion detection methodology that feeds a virtual memory snapshot of the target process to the detector.

[5]–[7], instruction set randomization (ISR) [33], [34] and data space randomization (DSR) [35]. OS extension approaches insert checkpoints in OS kernels or libraries to determine if calls to vulnerable library functions are safe [14], [15]. In general, DDC-based approaches have good detection completeness due to their extensive use of context information. However, troubleshooting is inefficient [17], [29].

Compared with DDC-based approaches, DBC-based approaches are efficient in troubleshooting but their detection completeness is not as good as DDC-based approaches' detection completeness. This is because DBC-based approaches do not use the runtime virtual memory information of target processes. We design a new DBC-based detection methodology that overcomes this limitation.

III. System Design

A. Design Rationale

We propose taking snapshots of the target process's virtual memory before input data are consumed and feeding these snapshots to our DBC-based detection system. This idea is illustrated in Fig. 5, where the DBC-based detector has a *new* input that is the virtual memory runtime information of the target process.

A virtual memory snapshot records the state of a process's virtual memory at this time, including the target process's address space and register values. In our system, snapshots are used in the following two ways:

– *Instantiating Virtual Execution Environment.* Immediately before the target process consumes data, its control flow is directed to our system. A virtual memory snapshot is taken at this moment and the detection procedure is immediately activated, instantiating a virtual environment where the input data are executed and monitored. Snapshots are used to initialize this environment and provide two benefits. First, snapshots are critical for observing the input data's *real* behaviors, as they illustrate *real* execution flow. To accurately reveal shellcodes' behavior, the environment should be able to mimic the process's consumption of input data. In malicious shellcodes, process state information can be used to redirect the execution flow, e.g, for encryption or decryption as Example 1 illustrates. Without precise virtual memory information about the process, shellcodes' execution flow can be changed or even interrupted.

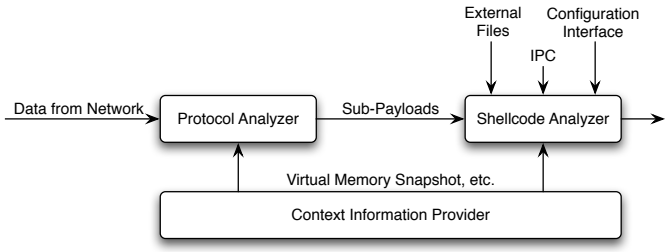

Fig. 6. Our System Modules.

Second, since the virtual memory snapshot is compact and easy to obtain, our system's virtual environment is *lightweight*, unlike the environments that *complete* system emulators such as QEMU create [24]. In existing DBC-based approaches, it is hard to observe *real* shellcode behavior in detection systems since the target process's virtual memory information is either assumed or ignored [22], [23], [29].

– *Facilitating System Call-based Detection.* One type of shellcode behavior that virtual memory snapshots facilitate is system call invocation, which is valuable for detection. Malicious shellcode must switch to kernel mode to launch OS-related operations by invoking system calls. No matter how well malicious shellcode disguises itself, it will eventually use system calls to launch attacks. Different system call operations are distinguished by system call numbers and parameters, which are normally stored in the registers. For example, in the IA-32 architecture, the system call number is placed into `eax` and its parameters are placed in `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp` as necessary. Existing DBC-based approaches, including both static analysis and dynamic analysis, cannot use system call invocations as detection criteria as they lack such necessary register information [22], [23], [36]. As we use virtual memory snapshots, we can accurately distinguish among different system calls to improve detection capability as well as accuracy.

In the following, we first introduce our system architecture followed by its workflow.

B. System Architecture

In this section, we first introduce our entire system architecture and then we present its key module—the shellcode analyzer.

1) Architecture: There are three modules in our system as shown in Fig. 6. The first module is the *Protocol Analyzer*, which extracts headers from input messages and separates the payload into several sub-payloads (fields) as necessary according to upper-layer specifications. The second module is the *Shellcode Analyzer*, which is responsible for detecting malicious shellcodes with virtual memory snapshots. It can receive sub-payloads from the protocol analyzer, an external file, or another process within the same host through local IPC. The third module is the the *Context Information Provider*, which is an interface between the *emulated* execution environment and the *real* one. This component assists the protocol analyzer in

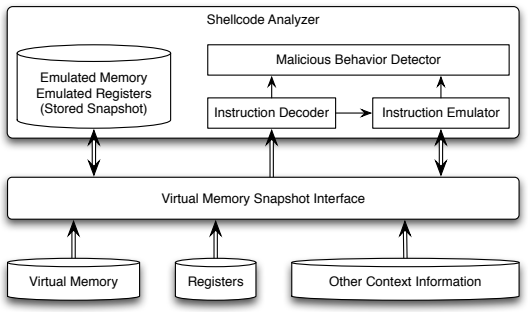

Fig. 7. The Shellcode Analyzer Architecture.

producing correct sub-payloads and the shellcode analyzer in conducting detection.

2) Key Module – Shellcode Analyzer: The shellcode analyzer architecture is shown in Fig. 7. This module consists of an instruction decoder, an instruction emulator, a malicious behavior detector, an emulated memory system, and a set of emulated registers. The instruction decoder iteratively decodes buffer contents into instructions and sends each one to the emulator. For each instruction the emulator receives, it emulates the execution thereof, for which the emulated memory system and registers provide a runtime virtual environment. This environment is instantiated by the virtual memory snapshot that is taken through the virtual memory snapshot interface. During emulation, virtual memory or register access is directed to the emulated memory system or registers.

C. Workflow

In this section, we first overview our entire system workflow, then we present our shellcode analyzer workflow.

1) Overview: Recall that our system has three modules: the protocol analyzer, the shellcode analyzer, and the context information provider. After receiving a message from a network, our detection system first uses the protocol analyzer to extract the message header and then separates the payload into several sub-payloads as necessary. Then the protocol analyzer feeds the payload (or sub-payloads) to the shellcode analyzer. The shellcode analyzer's instruction decoder treats the payload (or each sub-payload) as a byte-indexed sequence of input data. It decodes the data into different instruction sequences according to different start positions and sends them to the emulator. The virtual memory snapshot is taken at this moment and used to instantiate the emulation environment. The emulator executes each instruction sequence to determine whether there is any malicious behavior. If any such behavior is detected, the sequence is considered malicious, and if this occurs, the shellcode analyzer will conclude that the there is malicious shellcode in the payload (or sub-payloads) and the message is considered malicious.

Our detection system is not intended to run as an independent process. After a target process is created, our system can be loaded into its address space either statically or dynamically. When a target process receives a message from a network interface, our system is activated and the control flow is directed to our system.

```
1  #define MALICIOUS    1
2  #define BENIGN       0
3  #define MALICIOUS_SEQUENCE    1
4  #define BENIGN_SEQUENCE       0
5
6  int ShellcodeAnalyzer(base_addr, base_size)
7  {
8      for (i = 0; i< base_size; i++)
9          if (MaliciousInstructionSeq(base_addr + i))
10             return MALICIOUS;
11     return BENIGN;
12 }
13
14 int MaliciousInstructionSeq(addr)
15 {
16     InitializeEmulationEnvironment();
17     instruction = InstructionDecoder(addr);
18     if (End(instruction)) return BENIGN_SEQUENCE;
19     instruction.exe_depth = 1;
20     while (instruction)
21     {
22     if (MaliciousSystemCall(instruction))
23            if(instruction.exe_depth > exe_depth_threshold)
24                   return MALICIOUS_SEQUENCE;
25        InstructionEmulator(instruction);
26        UpdateEmulationEnvironment();
27        target = ComputeTarget(instruction);
28        prev_instruction = instruction;
29        instruction = InstructionDecoder(target);
30        if (End(instruction)) break;
31        SetExecutionDepth(instruction, prevInstruction);
32     }
33     return BENIGN_SEQUENCE;
34 }
```

Fig. 8. Shellcode Analyzer Workflow.

2) Shellcode Analyzer Workflow: The shellcode analyzer workflow is shown in Fig. 8. From each position of the input data, the shellcode analyzer uses the virtual memory snapshot to emulate the execution of the decoded instruction sequence. Snapshots facilitate observing *real* behaviors of the input data by illustrating the *real* execution flow. There are two input parameters for `ShellcodeAnalyzer()`. The first is `base_address`, the starting address of the input data to be analyzed. The second is `base_size`, the size of the input data.

Precise virtual memory information provided by the snapshot ensures the shellcode's execution flow is not interrupted. If an instruction sequence is malicious shellcode, all of the instructions it uses are eventually reached during emulation via the execution flow, including the shellcode's, those generated by previous messages, and those from the libraries loaded into the target process's address space. Thus emulation within the shellcode analyzer with a virtual memory snapshot can be used to accurately observe the behaviors of polymorphic and metamorphic shellcodes as well as shellcodes used in swarm attacks.

The key function of the shellcode analyzer is `MaliciousInstructionSeq()`, which detects a malicious instruction sequence. The workflow of `MaliciousInstructionSeq()` is shown in lines 14–34 in Fig. 8. At the beginning of `MaliciousInstructionSeq()`, the emulation environment is instantiated by taking a virtual memory snapshot. The `while` loop from line 20 to line 32 in Fig. 8 emulates a sequence of instructions, which continues until one of the following occurs: (1) a malicious behavior is detected; (2) a

privileged or invalid instruction is encountered;[1] (3) an illegal memory access occurs; (4) the number of executed instructions exceeds a threshold.

`MaliciousInstructionSeq()` returns `BENIGN_SEQUENCE` for conditions (2)–(4) and `MALICIOUS_SEQUENCE` for condition (1). For this condition, a *malicious behavior* is defined in our system by a *malicious* system call invocation. In Linux and MS Windows systems, not all system calls can compromise the target host's security. This depends on a system call's number and its parameters, which are stored in registers before a system call instruction is executed. Because of the snapshot, the system call number and its parameters can be accurately obtained to determine if the system call invocation is intended to compromise host security. For example, in Linux, system call number `11` corresponds to the system function `execve`, which executes a program. During emulation of an instruction, if it is a system call instruction and the value of the emulated `eax` is `11`, then the system call number is `11`. After checking its parameters stored in other emulated registers, if its first parameter is `/bin/sh`, then we can conclude that the instruction tries to open a root shell that can be used to compromise the host's security. In this case, the system call instruction will be considered to be a *malicious* system call.

Note that malicious shellcode normally uses several instructions to initialize system call parameters. We also use the `exe_depth` of an instruction that invokes a system call to decrease false positives, which is shown in line 23 of Fig. 8. An instruction's `exe_depth` is defined as the number of instructions from the starting point to it during emulation of an instruction sequence. For example, suppose that a statement S in a `for` loop is executed 100 times. Then the execution depth of S is `2` (the `for` statement and S). We further discuss the execution depth threshold in Section IV.

D. Detection Example

In this section, we use the example shown in Fig. 3 to illustrate our system's entire detection procedure. We assume that this malicious shellcode is inserted into the body of an HTTP request message and that it aims to open a root shell in an HTTP server with an exploitable memory error. When this message is received, the server's control flow is directed to our system. The protocol analyzer extracts the payload from the request body and calls the function `ShellcodeAnalyzer()` to analyze it. The instruction decoder will decode it into several instruction sequences, and feed each one to the function `MaliciousInstructionSeq()`. Immediately before emulating the execution, a virtual memory snapshot is taken. At this moment, suppose that the value of the memory unit pointed by `0xa0ef(ebp)` is `0x252d252d`.

Suppose that the function `MaliciousInstructionSeq()` is emulating an instruction sequence that contains malicious shellcode, and the instruction being emulated is `mov ebx,0xa0ef(ebp)` as shown in line 10 of Fig. 3. After emulating the instruction, the value of the emulated `ebx` is `0x252d252d`. After emulating the next two instructions, because the value of the emulated `ebx` is `0x252d252d`, the control flow will reach the instruction on line 15 of Fig. 3, namely, `push eax`. After emulating it, the first decrypted instruction will be written into the emulated memory system. Similarly, all of the encrypted instructions are decrypted and stored in the emulated memory system.

When the control flow later reaches the first decrypted instruction, all the decrypted instructions are read from the emulated memory system. Finally, the control flow reaches a system call instruction, and then the function `MaliciousSystemCall()` is called. By checking the emulated registers' values, we discover that the system call instruction tries to open a root shell. In this case, this system call instruction will be considered to be a malicious system call instruction. In addition, its *execution depth* exceeds the preset threshold. Thus the emulated instruction sequence is considered malicious as are the analyzed payload and the request message. After the control flow returns from our detection system, the server will discard the message.

IV. Implementation and Experimental Evaluation

A. Implementation

Our detection system is implemented in Debian Linux with kernel version 2.6.26. We build our prototype system in C using `gcc` 4.3.2. The prototype system focuses on the IA-32 architecture and Linux operating systems. The core component of our prototype system is the instruction emulator. In the following, we present its implementation.

Our instruction emulator can *interpret* all IA-32 instructions and *emulates* a subset thereof, including all general-purpose instructions and the FPU instructions that are used to obtain injected shellcodes' absolute addresses [22], [37]. In addition, we also emulate some system instructions such as `rtdsc`. This subset contains all instructions used by known malicious shellcodes in useful computation of malicious attacks [22]. We consider our implemented subset sufficient, though extensions are feasible. When an unimplemented instruction is encountered in emulation, if it is not a privileged instruction, the control flow skips it and goes to the next instruction; otherwise, emulation stops. When encountering an system call instruction, namely, `sysenter` and `int 0x80`, we check if it is one of 36 system calls that can be used to compromise the Linux system security [38]. Besides these "malicious" system calls, we also use the `exe_depth` threshold to determine if the instruction truly tries to compromise the host's security; we set the threshold to 14 since most unencrypted malicious shellcodes have at least 14 instructions [23], [29]. Recall that an emulation termination condition is that the number of the executed instructions reaches a threshold. According to current research, it suffices to set the threshold to `7000` in order to detect malicious shellcodes in the input stream [22], [23]. In our system, we also use this threshold.

[1]Privileged instructions can only be executed in kernel mode while a shellcode normally runs in user mode. If a shellcode contains a privileged instruction, it leads to an exception.

Data Set	# Requests	# Responses	Size (MB)
1	56,002	56,002	628.5
2	64,916	64,916	840.5
3	31,229	31,229	341.2
4	52,003	51,965	721.3
Total	204,150	204,112	2,531.5

Fig. 9. Collected Data Sets.

Data Set	Request Messages		Response Messages	
	# Malicious	False Positives	# Malicious	False Positives
1	0	0	41	0.000732
2	0	0	43	0.000662
3	0	0	27	0.000865
4	0	0	45	0.000866

Fig. 10. False Positives for the HTTP Traces.

Besides this core component, we also implement an emulated memory system and a virtual memory snapshot interface. The snapshot interface contains procedures to access the host's virtual memory snapshot and context information. Moreover, we use the Bastard project's `libdisasm`, version 0.23-pre [39] to construct our instruction decoder. In addition, during emulation of instructions, our detection system also collects the data that could be leveraged to implement the detection rules that are proposed by other papers [22], [23].

We have integrated our system into `glibc` and we also modify some read functions thereof. When these functions are called, they call our detection system to detect input data before returning to the calling procedure. If an application is statically linked to `glibc`, it must be rebuilt. However, when an application is dynamically linked to `glibc`, rebuilding is unnecessary, as our system will be dynamically loaded into the target process's address space. Users can also directly use our detection system in their source codes to check network data before they are consumed. Thus our detection system accurately obtains the virtual memory snapshot that malicious shellcodes utilize and then the detection accuracy of our detection system can be further increased.

B. Experiments

In this section, we use malicious shellcodes and HTTP traces to conduct our experiments, which are divided into two parts. The first part tests our detection system's effectiveness. The second part measures its overhead.

1) Effectiveness: In this part of the experiments, we first describe the collection of the data sets that are used to measure the false negatives and false positives of our detection system, and then we evaluate the results of our experiments.

We collect 51 unencrypted malicious shellcodes from the Internet that target Linux systems. We use these shellcodes in conjunction with the following encryption tools to generate 5000 encrypted malicious shellcodes: the Metasploit project's JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai [40] as well as ADMmutate [28] and TAPiON [41]. These 5000 encrypted malicious shellcodes and the 51 unencrypted malicious shellcodes are used to test our system's false negatives.

Since most Internet traffic is HTTP traffic, we use HTTP traces to test our system's false positive of our detection system. Because it is very difficult to obtain real traces of HTTP messages that contain the entire contents thereof, we enlist four volunteers who collect HTTP messages for six weeks and place them into data sets 1–4. The properties of these data sets are shown in Fig. 9, where *# Requests* and *# Responses* denote the number of request messages and response messages, respectively. The overall size of the traces are about 2.5 GB. The traffic is captured using Fiddler [42]. After obtaining the data sets, we feed our detection system these four data sets to test the false positives thereof.

In both of these two experiments, we run our detection system on a computer and then we feed the data into our system for testing. In the following, we present the evaluation of the experiments of this part.

Fig. 10 presents the false positives on request messages and response messages, respectively. *# Malicious* is the number of messages that our system considers malicious.

All request messages in these data sets are classified as benign messages with very few false positives. Response messages have few false positives. After checking the response messages that are classified as malicious, we find that some of them contain MS Windows executables. Others have large binary objects that have execution chains with malicious system call instructions and the execution depths of these instructions exceed the preset execution depth threshold. If a user receives a response message that is classified as malicious, he can choose to accept it, decline it, or perform a further check based on his knowledge of desirable response messages.

After sending the 5000 encrypted malicious shellcodes to our detection system, we find that it can detect all of them. This result is the same as other DBC-based approaches [22], [23], [36]. Although they have the same number of false negatives as our detection system, our system's false positives are lower than those of other DBC-based approaches.

Current approaches based on network-level emulation use the *`WX` threshold* to detect malicious shellcodes, where the `WX` instructions are those that are executed and are dynamically generated during shellcode emulation [23]. When we set this threshold to `8` as suggested by [23] and use it to detect the 51 unencrypted malicious shellcodes, we find that 31 of them cannot be detected. After studying the traces of these shellcodes, we find that they have no sequences of `WX` instructions of length `8` or larger. We feed these 51 unencrypted malicious shellcodes to our detection system and we find that it can detect all of them. It can detect them because it uses malicious system calls and emulation with a virtual memory snapshot to check whether an message contains a malicious shellcode, and all malicious shellcodes eventually invoke system calls to launch attacks.

Furthermore, our detection system can detect the examples presented in Figs. 3 and 4 but existing DBC-based approaches

Data Set	Request Payload Size (bytes)				Overhead (ms)			
	Min	Max	Average	Median	Min	Max	Average	Median
1	154	3412	799.28	736	2.002	125.144997	13.0219431	12.272
2	132	3864	826.93	827	3.032	120.025002	14.12347399	13.484
3	152	3822	921.02	890	2.107	150.350998	13.80020388	12.386
4	110	3845	840.89	777	1.916	138.990997	12.43439796	11.696

Fig. 11. Net Overhead Incurred by Our System.

cannot do so. The reason is that they do not use the target process's context information to detect malicious shellcode.

2) Overhead: In this section, we evaluate the performance of our system. Our tests consists of two experiments. The purpose of the first experiment is to evaluate the *net* overhead incurred by our detection system. The purpose of the second experiment is to investigate how our system affects the performance of a real HTTP server.

– *Experiment 1.* The first experiment is conducted on a Dell Dimension 5150 machine with an Intel Pentium 4 2.8 GHz processor and 1 GB RAM. In this experiment, we only use the HTTP request messages from the above four data sets to conduct the experiment since the main purpose of our detection system is to protect a server. To evaluate the net overhead incurred by our detection system, we first send a request message to the detection system and record the timestamps before our detection system is invoked and after it returns. The difference between the two timestamps is considered net overhead.

Fig. 11 presents our evaluation results. We believe the results are acceptable for most Internet application servers, though they are usually much more powerful than our test machine.

– *Experiment 2.* In the second experiment, we use two computers to test the performance of our system. One is a Dell Dimension 5150 Web server with an Intel Pentium 4 2.8 GHz processor and 1 GB RAM. The other is an IBM ThinkPad T60 Web client. Both of them are running Debian Linux with kernel version 2.6.26. They are connected by a 100 Mbps Ethernet switch. In the following, we first present the methods and data used in two experiments, and then we evaluate the results of these two experiments.

In the second experiment, we use the modified `glibc` with our detection system to rebuild a real Web server and evaluate our system's influence on the server's performance. We compare the performance of a Web server using our detection system with that of a server not using it. We use `thttpd`, a single-threaded open source Web server from ACME Laboratories that is designed for simplicity, a small execution footprint, and speed [43]. For the server using our system, when it calls `read()` or `recv()` to read from a socket, each function uses our system to analyze the data therefrom. Each function returns `-1` if the data are identified as malicious; otherwise, returns `0` and the data are consumed normally.

The client traffic was generated by Jef Poskanzer's HTTP load program [44] from the IBM machine. Clients send requests from a predefined URL list from which the referenced documents are stored in the server. We modified the original HTTP

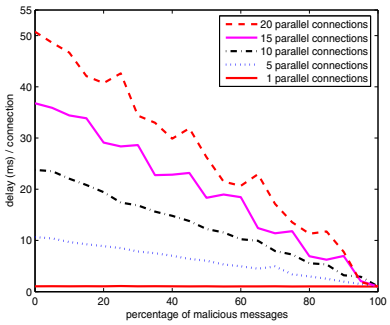

Fig. 12. Server Performance with Detection System.

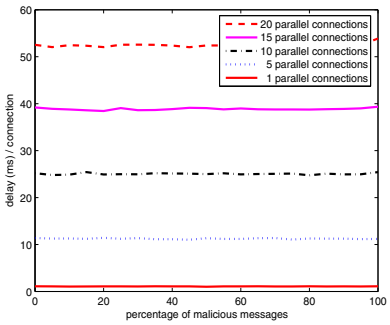

Fig. 13. Server Performance Without Detection System.

load program so the client can inject malicious shellcodes in the requests. As attacks that embed malicious payloads in the HTTP Request-Header have not yet been observed, the original `thttpd` Web server without using our detection system ignores malicious code and returns the requested documents. The server using our detection system returns HTTP status code `400` (bad request) if there is malicious code in the request as the read function returns `-1`.

We measured the average connection response latency by running HTTP load for 1000 fetches. Figs. 12 and 13 show the average connection latency as a function of the percentage of traffic attacking the Web server with and without our detection system, respectively. We randomly choose a shellcode and inject it into requests. We observe that the average latency is stable for the original `thttpd` web server, as malicious code is just ignored. The average latency of the server *with* our system decreases as the percentage of malicious attacks increases, since request errors are returned for connections with shellcodes. Comparing Fig. 12 with Fig. 13, we observe that the average response time in the system with detection is only slightly higher than in the system without detection when there are no malicious shellcodes. When there are malicious shellcodes, our prototype system actually reduces the average response time, which means attack requests have little impact on normal requests. For malicious messages, we observe from Fig. 8 that if any instruction sequence decoded from input data is malicious, then the processing thereof terminates. We also observe that during emulation of a malicious instruction sequence, it will be discovered before the number of executed instructions reach the threshold. Our prototype's experimental results show that our approach has reasonably low performance overhead.

V. Final Remarks

In this paper, we proposed a new detection methodology in which we fed the target process's virtual memory snapshot into a DBC-based detector. These snapshots were used to instantiate a runtime environment that emulated the process's input data consumption. This environment helped monitoring shellcodes' behaviors. To detect malicious shellcodes, we used the snapshots to examine system calls invoked by shellcodes masquerading as input data, the system call parameters, and the process's execution flow. We implemented a prototype system in Debian Linux. Our experiments with real traces showed our system's strong runtime performance with few false negatives and few false positives.

Although our system had few false negatives, one kind of attack may still evade detection. Our system can detect malicious shellcode that starts from some position in a message. If the attack only uses addresses of `glibc` functions to overwrite the current stack frame's return address [45], then the hijacked control flow will be directed to the function, not to a position in the message. This kind of attack can evade detection by our system. However, such attacks are difficult to implement as malicious users need to have *complete* knowledge of the target process and use very advanced techniques.

It is possible to further reduce our system's overhead by: (1) using static analysis techniques to analyze input data before or during emulation; and (2) designing a faster instruction decoder. We will address these directions in our future work.

The virtual memory snapshot is very useful at quickly and efficiently detecting malicious shellcodes in the input stream. We believe it can assist other detection approaches. Using this methodology in static analysis is another interesting direction for our future work.

References

[1] US-CERT Vulnerability Notes Database, http://www.kb.cert.org/vuls.

[2] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proc. NDSS*, 2005.

[3] D. Brumley, J. Newsome, and D. Song, "Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software," in *Proc. NDSS*, 2006.

[4] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proc. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'06)*, 2006.

[5] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent Runtime Randomization for Security," in *Proc. Int'l. Symp. on Reliable Distributed Systems*, 2003.

[6] PaX, http://pax.grsecurity.net/docs/aslr.txt.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proc. USENIX Security*, 2003.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proc. USENIX Security*, 1998.

[9] H. Etoh and K. Yoda, "Protecting from Stack-Smashing Attacks," http://www.trl.ibm.com/projects/security/ssp/main.html.

[10] http://www.angelfire.com/sk/stackshield.

[11] T. Chiueh and F.-H. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *Proc. IEEE ICDCS*, 2001.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proc. USENIX Security*, 2003.

[13] A. Smirnov and T. Chiueh, "DIRA: Automatic Detection, Identification, and Repair of Control Hijacking Attacks," in *Proc. NDSS*, 2005.

[14] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," in *Proc. USENIX Annual Technical Conf.*, 2000.

[15] G. S. Kc and A. D. Keromytis, "e-nexsh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," in *Proc. ACSAC*, 2005.

[16] J. P. McGregor, D. K. Karig, Z. Shi, , and R. B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks," in *Proc. ITRE*, 2003, pp. 243–250.

[17] G. Portokalidis and H. Bos, "Eudaemon: Involuntary and on-demand emulation against zero-day exploits," in *Proc. ACM EuroSys*, 2008.

[18] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," in *Proc. RAID*, 2002.

[19] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *Proc. USENIX Security*, 2003.

[20] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. E. Bryant, "Semantics-Aware Malware Detection," in *Proc. IEEE S&P*, 2005.

[21] A. Lakhotia and U. Eric, "Stack Shape Analysis to Detect Obfuscated Calls in Binaries," in *Proc. IEEE Int'l. Conf. on Source Code Analysis and Manipulation*, 2004.

[22] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level Polymorphic Shellcode Detection Using Emulation," in *Proc. DIMVA*, 2006.

[23] ——, "Emulation-Based Detection of Non-Self-Contained Polymorphic Shellcode," in *Proc. RAID*, 2007.

[24] QEMU, http://www.nongnu.org/qemu/.

[25] Fnord, http://www.cansecwest.com/spp_fnord.c.

[26] R. Chinchani and E. van den Berg, "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows," in *Proc. RAID*, 2005.

[27] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. van Underduk, "Polymorphic Shellcode Engine Using Spectrum Analysis," *Phrack*, 2003, http://www.phrack.org.

[28] S. Macaulay, "ADMMutate: Polymorphic Shellcode Engine," http://www.ktwo.ca/security.html.

[29] X. Wang, C.-C. Pan, P. Liu, and S. Zhu, "SigFree: A Signature-Free Buffer Overflow Attack Blocker," in *Proc. USENIX Security*, 2006.

[30] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic Analysis of Malicious Code," *Journal of Computer Virology*, 2006.

[31] Skape, "Shellcode text encoding utility for 7bit shellcode," http://www.hick.org/code/skape/nologin/encode/encode.c.

[32] S. P. Chung and A. K. Mok, "Swarm Attacks against Network-Level Emulation/Analysis," in *Proc. RAID*, 2008.

[33] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proc. ACM CCS*, 2003.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," in *Proc. ACM CCS*, 2003.

[35] S. Bhatkar and R. Sekar, "Data Space Randomization," in *Proc. DIMVA*, 2008.

[36] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "STILL: Exploit Code Detection via Static Taint and Initialization Analyses," in *Proc. ACSAC*, 2008.

[37] C. Ionescu, "GetPC code," http://securityfocus.com/archive/82/327348/2006-01-03/1.

[38] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting Execution Contexts for the Detection of Anomalous System Calls," in *Proc. RAID*, 2007.

[39] http://bastard.sourceforge.net.

[40] The Metasploit Project, http://www.metasploit.com.

[41] P. Bania, "TAPiON," 2005, http://pb.specialised.info/all/tapion/.

[42] Fiddler, http://fiddler2.com/fiddler2.

[43] thttpd, http://www.acme.com/software/thttpd/.

[44] httpload, http://www.acme.com/software/http_load/.

[45] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proc. ACM CCS*, 2007.