Performance Analysis of Encrypted Code Analyzer for Malicious Code Detection

Daewon Kim, Yongsung Jeon, and Jeongnyeo Kim
Cyber Security Research Department
Electronics and Telecommunications Research Institute
Daejeon, Korea
emails: {dwkim77, ysjeon, jnkim}@etri.re.kr

***Abstract*—Signature-based malicious code detection systems cannot in real-time detect unknowns, such as polymorphic and metamorphic codes, which can be used as zero-day attacks. More serious situation is that many automated engines easily generate new malicious codes without the attacker's special knowledge. We have already proposed a method to detect polymorphic parts of suspicious packets in anomalous network traffic. In this paper, we introduce the experiments and analysis to show the real field effectiveness and performance of our method.**

Keywords-zero-day attack; malicious code; polymorphic code; unknown attack; intrusion prevention system.

I. INTRODUCTION

Static analysis methods [2] to detect polymorphic exploit codes can be avoided by exploits using static analysis resistant techniques, which includes disassembly thwarting and self-modifying code techniques. To catch the techniques, dynamic analysis methods that directly emulate the instructions of packets include full dynamic analysis methods [3], which use a CPU emulator, and a hybrid analysis method [4], which uses both static and dynamic analyses.

Full emulation methods have an advantage in that they can detect most encrypted malicious codes. However, the overhead of emulating instructions makes it difficult to apply to real high-speed networks. A hybrid method offers better performance than a full method because the starting point of emulation can be selected through the support of a static analysis. However, hybrid methods are still insufficient for real networks owing to the complicated operational process of a static analyzer and an emulator.

Our previous work [1] showed that it can detect the decryption routine using the disassembly thwarting and self-modifying techniques. In this paper, we will present more practical examples and experiment results to show real field effectiveness.

The rest of the paper is organized as follows. In Section 2, we overview our method and describe the operation steps. In Section 3, we show our evaluation results. Finally, we conclude the paper in Section 4.

II. ENCRYPTED CODE ANALYZER

In this section, we will present the overview and example of our previous work [1] to help the understanding of our experiment results.

A. Overview

Our encrypted code analyzer detects the loop code instructions in an encrypted exploit code to decrypt the encrypted code itself. Normally, for ease of development, the decryption routine of an encrypted code stores the current program counter value on the stack and uses the value as the address for accessing the memory of an encrypted original code. Our previous work includes four kinds of components.

Firstly, seed detector detects the instructions loading the current program counter, which is a base value, into the stack. The Register Loading Base Value (RLBV) detector traces a register loading the program counter value on the stack. The Memory access Using Base Value (MUBV) detector traces the movements of registers including the base value between instructions. Lastly, loop detector determines the existence of a loop code if the final register traced by the MUBV detector is used for the instructions to access memory.

B. Operation Process

Non-malicious codes normally don't hide their original codes. One of new techniques for avoiding signature-based detection systems is code encryption. Encrypted malicious codes, which are polymorphic codes, have any code routine for decrypting to original codes at run-time. Our analyzer is based on the special patterns of decryption routine codes. If there are non-malicious codes that have similar behaviors to malicious codes, it needs more time and analysis to classify those. The cases are out of scope for our analyzer targeting real-time detection.

The analyzer in Figure 1(a) detects the seed instructions that store the address value related to the current program counter into a stack memory. The address indicates the start address of encrypted codes, which is called the base address. If the seed instructions are detected, the analyzer generates a virtual stack to trace the operations of the stack with the *base* address and in (b) detects a register loading the base address. After that, (c) the analyzer traces the movements of the *base* address from the first register, and (d) checks whether the final register with the base address is used for accessing a

 ISBN: 978-1-61208-444-2

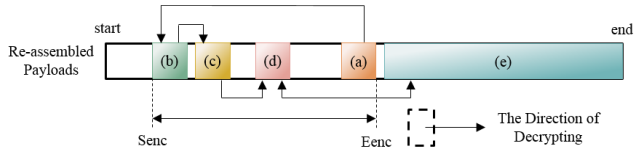

Figure 1. Encrypted code analyzer: (a) seed codes, (b) RLBV codes, (c) MUBV codes, (d) loop codes, and (e) encrypted codes.

valid memory address. The detected codes of (d) decrypt the encrypted codes of (e).

The seed detector finds the instructions shown in Figure 1(a) and creates a virtual stack. The instructions frequently used for a seed are call, fsave, fnsave, fstenv, and fnstenv. As an example, if fnstenv [esp-0c] is detected, the base address is on the top of the stack and is written on the created virtual stack. To decrypt the encrypted codes, the loop codes load the base address into a register using instructions such as pop esi. The RLBV detector traces the position of the base address stored on the real stack using a virtual stack that is operated by instructions such as push/pop, inc/dec/sub/add, and mov. Finally, the detector determines the last register using the base address.

A register with the base address is referenced for accessing the address range of the encrypted code. Attackers can move the base address to other registers to hide the memory accesses referenced by the register with the base address. The MUBV detector expresses the movements between registers as the connection graph for inspecting the register relations. Through this graph, the detector can determine a final register with the base address. The loop detector analyzes the instructions for accessing any range of memory with the base address included in the detected register. One case of instructions is xor byte ptr [ecx+esi-1],93, and our detector analyzes the validity of the address range. If the instruction accesses the memory range near the re-assembled payloads, the detector determines that the payloads are encrypted malicious codes and reports the start-position, Senc, and end-position, Eenc, to our signature generation system. Our previous work [1] described the encrypted code analyzer in greater detail.

III. EXPERIMENTS

Our previous work showed the detection rate, false positive rate, and performance of our encrypted code analyzer. For evaluating the detection rate, we used four kinds of polymorphic generation tools, and thirteen kinds of polymorphic generation engines. Our analyzer archived a 100% detection rate for all polymorphic codes that include disassembly thwarting and self-modifying code techniques.

Figure 2 shows several detection results against the encryption routines that were generated from the use of Linux/x86/shell_bind_tcp exploit. Moreover, the low instruction traversing counts indicate that the detection speed of our analyzer is similar to other static methods.

Figure 3 shows the processing overhead estimated under the system for a 3.2 GHz Pentium 4 processor with 4 GB of RAM on Cent OS (kernel version 2.6.9). The sample is network traffic captured as pcap files in a university that has

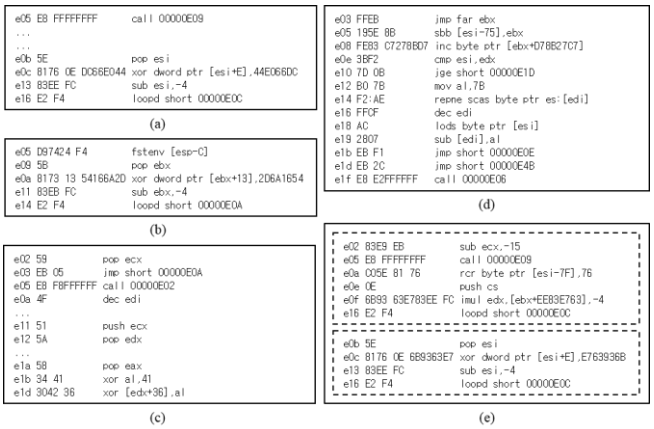

Figure 2. The decryption routines detected by the encrypted code analyzer: (a) Call4DwordXor, (b) FnstenvMov, (c) PexAlpha-Num, and (d) NonAlpha (e) Pex.

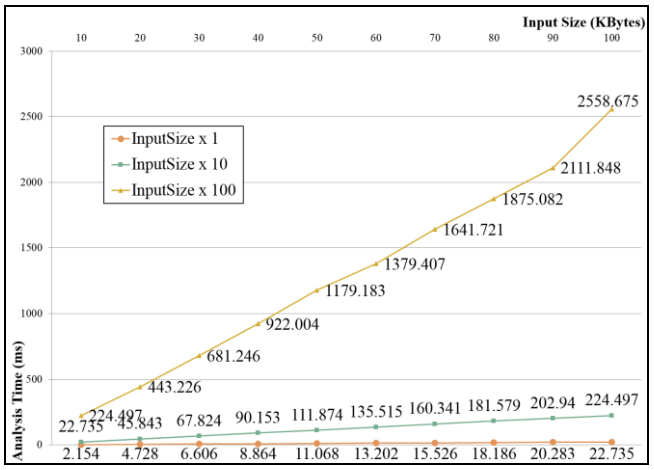

Figure 3. The processing overhead.

policies for clean network. The C function gettimeofday() was used for evaluating each processing overheads. The y-axis analysis time was calculated as the difference of full_time, which includes pcap_parsing+disassemble+detection, and disassemble_time, which includes pcap_parsing+disassemble. The analysis time is the detection time of our analyzer and it means the RLBV+MUBV+loop detector time except for the seed detector.

The results show a linear increasing trend similar to the static methods. Normally, exploit code size is small under a few tens of kilobytes. It means that the result of InputSize x 1 is useful to show a linear overhead. If the bad cases that suspicious traffic is continuously analyzed as back-to-back are considered, other two graphs are useful to show to maintain a linear increasing of analysis time. At present, we guess that the abnormal increase between the last two points, which are 2111.848 and 2558.675, on Input Size x 100 is occurred by any buffer problems between our analyzer and disassembler [5].

 ISBN: 978-1-61208-444-2

IV. CONCLUSIONS

For the detection of polymorphic codes, we have already proposed a new static analysis method for detecting self-contained polymorphic codes using static analysis resistant techniques. In this paper, we overviewed the main functions and presented experiments to show the real field effectiveness of the proposal.

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R-20150518-001267, Development of Operating System Security Core Technology for the Smart Lightweight IoT Devices].

REFERENCES

[1] D. Kim, I. Kim, J. Oh, and H. Cho, "Lightweight Static Analysis to Detect Polymorphic Exploit Code with Static Analysis Resistant Technique," Proc. of IEEE ICC, June 2009, pp. 904-909.

[2] R. Chinchani and E. V. D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows," Proc. of RAID, Sep. 2005, pp. 284-308.

[3] M. Polychronakis, K. Anagnostakis, and E. Makatos, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," Proc. of RAID, Sep. 2007, pp. 87-106.

[4] Q. Zhang et al., "Analyzing Network Traffic to Detect Self-decrypting Exploit Code," Proc. of ACM ASIACCS, Mar. 2007, pp. 4-12.

[5] Libdasm – A Disassembly Library. [Online]. Available from: https://code.google.com/p/libdasm/. 2015.06.16.

 ISBN: 978-1-61208-444-2