Signature Generation Based on Executable Parts in Suspicious Packets

Daewon Kim, Jeongnyeo Kim, and Hyunsook Cho
Cyber Convergence Security Research Department
Electronics and Telecommunications Research Institute
Daejeon, Korea
{dwkim77, jnkim, hscho}@etri.re.kr

***Abstract*—Generally, attackers obtain the control authority of a remote host through the exploit/worm codes with some executable parts. The majority of the codes are still made of the codes which can be executed directly by CPU of the remote host without some decryptions. We focused on the fact that some parts in the exploit/worm codes include the function call related instruction patterns. In some suspicious packets with the exploit/worm codes, the function call instruction parts can be important information to generate the signature of Intrusion Detection System (IDS)/Intrusion Prevention System (IPS) for blocking the packets with the exploit/worm. In this paper, we propose the approach that detects the instruction patterns following the function call mechanism in some suspicious packets and generates a signature including the specific payload positions within the pattern-detected packets. We have implemented a prototype and evaluated it against a variety of the executable and non-executable codes. The results show that the proposed approach properly classifies the executable and non-executable codes and can generate the high-qualified signature based on the analyzed results.**

Keywords-network security; intrusion detection system; intrusion prevention system; malicious code; exploit code; worm code

I. INTRODUCTION

To avoid the signature-based IDS/IPS such as Snort [4], Bro [5] and recent techniques [11], [12], encrypted exploit/worm codes [1]-[3] are gradually increasing. However, in real fields, most of the exploit/worm codes are still non-encrypted codes. Therefore, it is possible to detect and prevent the exploit/worm codes if a distinction can be made between the executable and non-executable codes in network flows with the anomalous and suspicious traffic patterns because normal network services of servers are primarily based on non-executable plain texts and not executable codes [6],[7].

Several researches were published to detect malicious codes in network traffic. Earlybird [8] and Autograph [9] are based on the fact that different instances of the exploit/worm codes would contain common substrings or fingerprints, which would potentially have the code patterns to penetrate vulnerabilities. TRW (Threshold Random Walk) [10] is based on the idea that the exploit/worm codes infected host that is scanning the network randomly will have a higher connection failure rate than a host engaged in legitimate operations. However, for generating signatures, the above researches have difficulty analyzing the logical features of non-encrypted malicious codes because they are based on the simple matching of repeated payload substring and traffic-behavior. As a result, the probability of detection decreases significantly as the size of input data is decreased.

Although not a complete program, the executable part of a non-encrypted malicious code has very logical features. As a malicious code has many action roles, attackers have included many function-based logics in malicious codes. Finally, non-encrypted malicious codes have high probability of including the logical feature following function call mechanisms.

In this paper, we extended our previous work [13] by proposing a signature generation method based on the payload positions detected by our function call detection mechanism. The proposed method calculates the match probabilities of instruction patterns according to the function call mechanism and determines the existence of executable codes in the suspicious packets of anomalous traffic. Finally, the method generates a unique signature with the packet payload including the detected function call instructions.

The rest of the paper is organized as follows: Section II overviews the background and operation according to our method; Section III presents analysis steps of the proposed method; Section IV shows the experimental results; and Section V presents our conclusion and suggestions for future

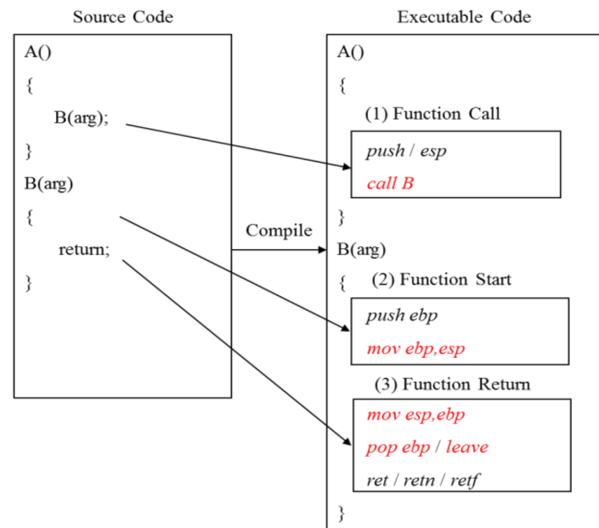

Figure 1. Instruction patterns of function call/return pairs.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-279-0

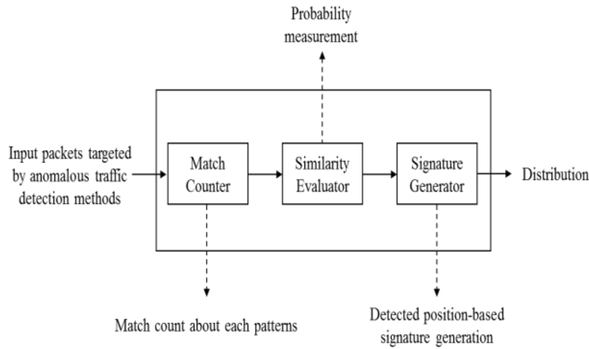

Figure 2. Operational overview.

TABLE I. INSTRUCTION PATTERNS ACCORDING TO FUNCTION CALL MECHANISM

Fn.	Not.	Instruction order to be matched for each notation		
		1	2	3
Call	ec	esp ops.	call(s)	
	pc	push	call(s)	
Start	pm	push ebp	mov ebp,esp	
Return	mpr	mov esp,ebp	pop ebp	ret(s)
	pr	pop ebp	ret(s)	
	lr	leave	ret(s)	

'esp ops.' means instructions that include '*%esp*'.

works.

II. OVERVIEW

The function call instruction patterns are one of the logical features in the executable codes. If a source code with functions is compiled, the function parts are transformed into the instruction patterns with *call*/*return* pairs. In the IA (Intel Architecture)-32, Fig. 1 shows the generated instruction patterns after the function *call*/*return* is compiled.

The proposed method detects the patterns of Fig. 1 and decides in terms of probability whether an executable code exists in the payload of suspicious packets or not. After that, the method generates a signature based on the detected position in the packets. Fig. 2 shows the simple process flows of the method.

In Fig. 2, *Match Counter* measures the trial and match counts of Fig. 1 instructions about the input packets. *Similarity Evaluator* has the pattern match probabilities of executable codes and compares them with the results of *Match Counter*. *Signature Generator* generates a signature including the payload around detected positions.

III. SIGNATURE GENERATION BASED ON FUNCTION CALL INSTRUCTIONS

A. Match Counter

The pattern match counts in the detection window of any instruction range are measured, and moving through the instructions one by one, this measuring is repeated to the end

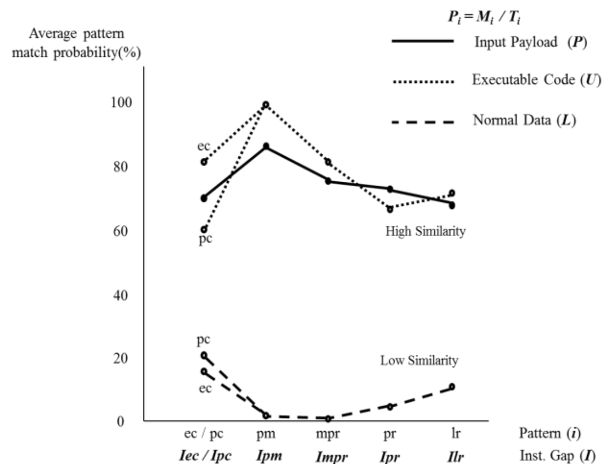

Figure 3. Probability comparison of input payload and real executable code.

of an input payload. In the IA-32, the instruction patterns defined by our method are presented in Table I.

In Table I, the attempt to match patterns is triggered by the gray-highlighted triggering instructions. Other instructions in Table I are inappropriate for triggering detection because they are frequently appeared regardless of the function call mechanism. When *Match Counter* measures the trial and match counts according to the instruction patterns of Table I, the instruction gaps between the instructions of Table I have to be considered. It is because some additional instructions can be made between the instructions of Table I by a compiler. Therefore, the pattern match counts of Table I should be counted in the acceptable instruction gap size.

In the case of a *pc* pattern, for example, if the *call* - which is the instruction number 2 - within the detection window is detected, the trial count of the *pc* pattern is increased by one. If the instructions are compared one by one in the reverse direction of the *call*, and if the *push* - which is the instruction number 1 - is detected, the match count is increased by one. At this time, if the number of instructions tracked as the reverse direction exceeds the pre-defined instruction gap size, the match count is not increased because of the match failure.

B. Similarity Evaluator

After the trial count set T and match count set M are measured on each notation, the match probability set $P_i = M_i / T_i$, where $i = \{ec, pc, pm, mpr, pr, lr\}$, is calculated. Our basic idea considers that P will be similar to the match probability set U of the real executable code if the input packets have executable codes constructed as some functions. Fig. 3 shows an example to describe this idea.

In Fig. 3, the match probability exists in both the executable and non-executable code. It means that the false positive can be large if the total trial count is very small. Therefore, for a more reliable analysis result, the similarity calculation to decide the existence of executable codes in the current detection window should be processed when the total trial

count within the current detection window is larger than the minimum trial count e.

The more similar the input payloads are to the executable code, the closer P would be to U. This could be calculated from the relative similarity set R_i between U and L like the below formula.

$$R_i = \frac{P_i - L_i}{U_i - L_i} \ except\ for\ T_i = zero, \quad (If\ P_i \geq U_i, R_i = 1.0.\ If\ P_i \leq L_i, R_i = zero.) \tag{1}$$

In Fig. 3, each pattern has individual probability gaps between U and L. It means that the decision about the input payload is more reliable if the gap is large. Therefore, the weight set W_i is required to increase the reliability.

$$W_i = \frac{U_i - L_i}{\sum_j U_j - L_j} \ except\ for\ T_i = zero, \tag{2}$$

If the final weighted similarity s is larger than the decision threshold d, the input payload evaluated by our method has high probability of including some executable codes.

$$s = \sum_i (W_i \cdot R_i) \ except\ for\ T_i = zero. \tag{3}$$

C. Signature Generator

Signature generating does not require special techniques in this paper because the signature style is various according to the IDS/IPS. Based on the detection results of our method, it can be the entire payload or the specific-range payload in an input packet. In the case of specific-range, the signature needs to be a continuous range to include the detected all position.

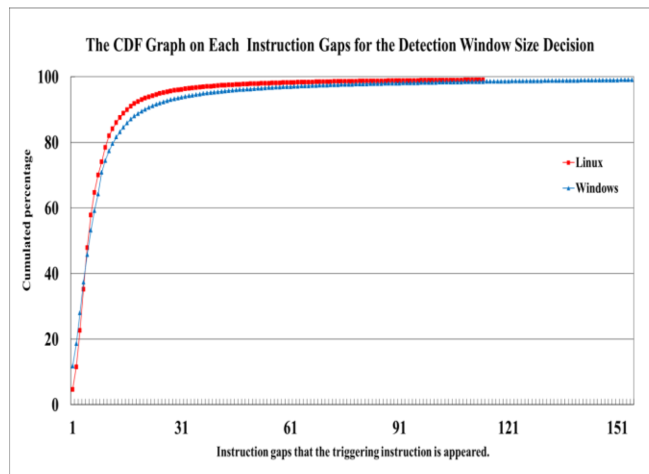

Figure 4. Existence probability of instructions gaps between triggering instructions.

IV. EXPERIMENTS

The test files for the experiments are the IA-32 based 3000 executable files of Windows/Linux and 3000 data files such as .txt, .doc, .ppt, pdf, mp3, gif, etc. In the case of executable files, only <.text> section was used in the experiment.

A. Size of Detection Window

When the detection window moves one byte at a time, the triggering instruction is always required for the analysis. In Fig. 4, when we select the existence probability of triggering instruction as 99%, the detection window sizes were 114 instructions in Windows and 155 in Linux. Therefore, the desired size z can be set as 155, which is about 450 bytes.

B. The Match Probabilities and Instruction Gaps of Executable and Non-Executable Code

Table II shows the experiment results for the match probabilities and the instruction gaps of executable and non-executable code. The determined detection window size of 450 bytes and the results of Table II show that this work proposes a reasonable method for detecting executable codes although the input is only one packet.

C. Executable Threshold and Minimum Trial Count

Figs. 5 and 6 show some parts of experimental results to determine the executable threshold d and minimum trial count e. In Fig. 5, the threshold d of executable codes is

TABLE II. PARAMETERS DETERMINED BY EXPERIMENTS

Notation	I	U	L
ec	1	0.80	0.10
pc	2	0.60	0.20
pm	3	0.98	0.02
mpr	2	0.80	0.01
pr	7	0.75	0.25
lr	2	0.70	0.10

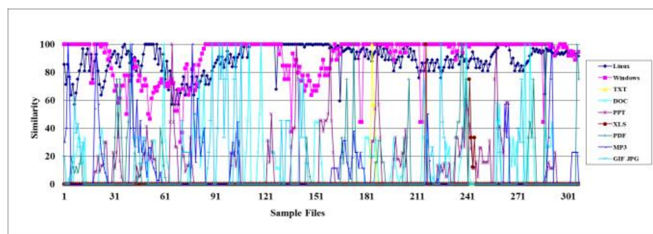

Figure 5. A graph for determining executable thresholds.

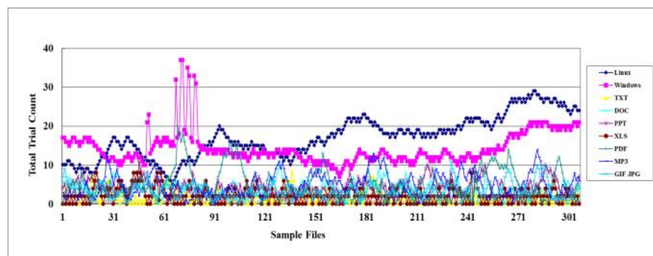

Figure 6. A graph for determining a minimum trial count.

over 60% and in Fig. 6, the minimum trial count e is about 3.

V. CONCLUSION AND FUTURE WORKS

In this paper, we proposed an approach that detects the instruction patterns following the function call mechanism in some suspicious packets and generates a signature including the specific payload positions within the pattern-detected packets. As the experiments shows, the proposed detection method is efficient even for one packet.

Regarding the method of detecting executable codes, our method analyzes in a form that is similar to the pattern-matching of instruction patterns following the function call mechanism. Our method can determine whether the executable codes exist or not in terms of the probability even in small input payload. In current method, we used a Detection Window of several hundred bytes. In next experiment, we will try a method which sequentially searches the payloads in order to detect the triggering instructions without the Detection Window. As a result, we may be able to identify the function call patterns for input payloads of a smaller size.

ACKNOWLEDGMENT

This work was supported by the ETRI R&D program of KCC (Korea Communications Commission), Korea [12-912-01-001, "Development of MTM-based Security Core Technology for Prevention of Information Leakage in Smart Devices"].

REFERENCES

[1] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis, "Network-level polymorphic shellcode detection using emulation," Proc. of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA06), July 2006, pp. 54–73.

[2] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer, "Analyzing network traffic to detect self-decrypting exploit code," Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS07), 2007, pp. 4-12.

[3] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis, "Emulation-based Detection of Non self-contained Polymorphic Shellcode," Proc. of the International Symposium on Recent Advanced in Intrusion Detection (RAID07), 2007, pp. 87-106.

[4] M. Roesch, "Snort: Lightweight intrusion detection for networks.," USENIX LISA Conference, 1999, pp. 229-238.

[5] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-time," Proc. of the USENIX Security Symposium, Jan. 1998, pp. 2435-2463.

[6] R. Chinchani and E. V. D. Berg, "A fast static analysis approach to detect exploit code inside network flows," Proc. of 8th International Symposium on Recent Advanced in Intrusion Detection (RAID05), 2005, pp. 284-308.

[7] X. Wang, C. Pan, P. Liu, and S. Zhu, "SigFree: A Signature-free Buffer Overflow Attack Blocker," Proc. of the 15th USENIX Security Symposium, 2006, pp. 225-240.

[8] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," Proc. of the 6th Symposium on Operating Systems Design & Implementation (OSDI04), 2004, pp. 45-60.

[9] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," Proc. of the 13th USENIX Security Symposium, 2004, pp. 271–286.

[10] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," Proc. of IEEE Symposium on Security and Privacy, 2004, pp. 211-225.

[11] B.-H Chang and C. Jeong, "An Efficient Network Attack Visualization Using Security Quad and Cube," ETRI Journal, vol. 33, no. 5, Oct. 2011, pp. 770-779.

[12] S. A. Taghanaki, M. R. Ansari, B. Z. Dehkordi, and S. A. Mousavi, "Nonlinear Feature Transformation and Genetic Feature Selection: Improving System Security and Decreasing Computational Cost," ETRI Journal, vol. 34, No. 6, Dec. 2012, pp. 847-857.

[13] D. Kim, Y. Choi, I. Kim, J. Oh, and J. Jang, "Function Call Mechanism Based Executable Code Detection for the Network Security," Proc. of the International Symposium on Applications and the Internet (SAINT08), 2008, pp. 62-67.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-279-0