LETTER

Tracing Stored Program Counter to Detect Polymorphic Shellcode

Daewon KIM[†a)], Ikkyun KIM[†], Jintae OH[†], *Nonmembers, and* Jongsoo JANG[†], *Member*

SUMMARY The shellcode use of the polymorphic form has become active as the de facto method for avoiding signature based network security system. We present a new static analysis method for detecting the decryption routine of the polymorphic shellcode. This method traces the processes by which the decryption routine stores the current program counter in a stack, moves the value between registers and uses the value in order to make the address of the encrypted code accessible. Most of decryption routines have the feature which they use the program counter stored on a stack as the address for accessing the memory that the encrypted code is positioned.

key words: *network, security, shellcode, polymorphism*

1. Introduction

The main goal of an attacker is to obtain the chair control of the remote host. This is possible because there exists a vulnerable service which changes the control flow of the remote host and in which an attacker arbitrarily can execute the malicious code. The general method for obtaining the chair control of the remote host is against the vulnerable service through the shellcode transmission. Recent network-based attack detection technologies more and more broaden the application areas, however, most of them have the basic limit which is the signature-based. The shellcode in which the polymorphic technique is used due to this limit has a difficulty in detection.

To overcome this limit, through some researches the binary analysis has been attempted toward the payloads of target packets to detect the polymorphic shellcode. With the first of these attempts, various static analysis methods were used. However, there was a difficulty by static analysis resistant methods [1]–[3] including the disassembly thwarting technique and self-modifying code technique in detection.

Not to be influenced in the static analysis resistant techniques, recent researches have been studied as the dynamic analysis methods [4], [5] that use CPU emulation and the hybrid [6] that uses the static-dynamic analysis. There is the advantage that the former can find most of polymorphic shellcodes; however, because instructions are one by one performed as CPU actually computes, it has the disadvantage that the operation overhead high. The latter firstly detects the GetPC code used as the object finding out the address of the encrypted code through the linear or the recursive disassembly. As the next step, after the tree of instruction sequence is found with the beginning of the decryption routine through static analysis, by emulating from the start instruction of tree it detects the feature of the decryption routine. As to this method, there is the advantage that the operation overhead is lower than the former; however, there is the disadvantage that the polymorphic shellcode which does not use GetPC code cannot detect.

The method of this paper shows that it can detect the polymorphic shellcode in which the disassembly thwarting and self-modifying code techniques are used through the static analysis method. In result, compared with the hybrid method, this method can achieve the similar detection performance and the little overhead. This method performs the disassembly per every byte to detect the *seed* instruction for GetPC so that it is not influenced by the disassembly thwarting technique. Moreover, before the self-modifying code is just operated, the feature of the decryption routine is analyzed and the decryption routine is detected.

Because attackers are difficult to forecast the address of the encrypted code of the polymorphic shellcode injected into the remote host, they store the current program counter value through the decryption routine on a stack and use as the address for accessing the memory of the encrypted code. The method of this paper statically traces the movement of the program counter value between registers.

2. The Proposed Method

2.1 The Overview

The whole idea of this method detects whether the program counter value which the decryption routine stores through the static analysis method is used in accessing the memory or not. In the first step, the method finds the *seed* instruction playing the role of storing the program counter value on a stack. As the second, the method detects a register loading the value. The third step is to trace the relation between the register and others. Finally, if the loaded program counter value is used for accessing a memory, the input data is determined to the decryption routine of polymorphic shellcode.

2.2 The Decryption Routine Detection

• Step 1: The *seed* instruction detection for GetPC

Manuscript received January 7, 2008.
Manuscript revised February 29, 2008.

[†]The authors are with Information Security Research Division, Electronics and Telecommunications Research Institute, 161 Gajeong-dong, Yuseong-gu, Daejeon, 305–350, Korea.
a) E-mail: dwkim77@etri.re.kr

DOI: 10.1093/ietisy/e91–d.8.2192

The first step to find the decryption routine is the *seed* instruction detection for GetPC. The instruction stores the current program counter value on a stack and it is necessary code to find the access address of encrypted code and to use the self-decrypting technique. If an attacker already knows the information about the specific register value when the polymorphic shellcode is put on in memory of the remote host, the instruction is unnecessary [5]. However, it is not in an attacker the easy task to predict a situation. Therefore, by using the instruction, in a general way, an attacker draws up the decryption routine.

There are the *call*, *fsave*, *fnsave*, *fstenv* and *fnstenv* with *seed* instructions for GetPC which can be used. This method performs every byte disassembly and does not miss the instructions. The virtual stack space is created and the in-between of the space is set as the virtual stack pointer if the instruction is detected. The current program counter value is stored in the virtual stack that the virtual stack pointer is pointing. In the case of *call*, the value is stored in the position of current virtual stack pointer and, in the case of *f* series instruction, the target position of the virtual stack pointer is calculated through the static analysis and the value is stored in the calculated position.

If the *f* series have no relation with the stack operation, the detected instruction is determined to be not included in the description routine. Because an attacker is difficult to know the memory and all kinds of the register values of the remote host, except for a stack the possibility of making the arbitrary memory accessible is due to be low.

• Step 2: The register detection loading a program counter

A description routine loads the program counter value stored in the virtual stack space into the specific register. If the instruction which accesses the memory which is not a stack before loading the program counter value from a stack shows up, it is not the description routine. It is due to the attacker's knowledge limit about the remote host.

In a description routine, many instructions may be inserted as dummy with from time to time in order to avoid general polymorphic detection methods. Therefore, it needs to trace from the virtual stack position in which the program counter is stored whether the value is loaded with the specific register or not. Firstly this method traces the location of the virtual stack pointer by the *push/pop* and the *inc/dec/sub/add* that are frequently used as the basic operation instructions. These traces can be simply performed with the static analysis.

• Step 3: The connection relationship tracing between the other registers and the register in which the program counter value is stored.

The encrypted original code is accessed by the program counter value loaded in the STEP2 with the specific register. Therefore, if it is used in the instruction in which the register accesses the memory, finally it is determined to a polymorphic shellcode including the description routine.

The program counter value may be moved to the others in the register which stored the program counter value for the first time in order to hide that the value is used in the memory access. Like this case, if it is expressed as the connection relation between the registers that the program counter value is moved and there is among the connected registers that it is used in the instruction accessing the memory, it means that the program counter value pointing at the detected *seed* instruction is used for the access of encrypted code.

Like STEP2, an attacker may use dummy instructions to confuse the analysis. The register in which the program counter value was stored as the first may be again performed with *push* and it can be popped to the other register. Moreover, the value may be moved to the other register through the arithmetic or the logical operation instruction. In the case of former, the connection relation can be founded out through the stack trace like STEP2. In the case of latter, firstly the operand part of an instruction is classified an input and output. If the register in connection relation exists in an input and the new register exists in an output, the new register is included in the connection relation. For example if in STEP2 *ecx* is detected and current instruction is *mov eax, ecx*, *eax* has a correlation with the *ecx*.

Finally, in this way, if one among the connected registers is used in the memory access calculation, our method determines that the decryption routine of the polymorphic shellcode exists. For example, the instruction is like *xor* [*eax* + 15], 12.

2.3 The Detection Example of Real Decryption Routine

Table 1 presents, in a Honey net, on November 27, 2007, a complicated decryption routine of the polymorphic shellcode detected by our method.

Table 1 The decryption routine detected in the honey net.

1225	E80F000000	call 00001239	126a	83C101	add ecx, 1
122a	E862000000	call 00001291	126d	83EF01	sub edi, 1
122f	6764FF360000	push dword ptr fs:[0]	1270	83FF00	cmp edi, 0
1235	5D	pop ebp	1273	77 EE	ja short 00001263
1236	8D6D08	lea ebp, [ebp+8]	1275	59	pop ecx
1239	29C0	sub eax, eax	1276	67648B260000	mov esp, fs:[0]
123b	FEC8	dec al	127c	64678F060000	pop dword ptr fs:[0]
123d	08C0	or al, al	1282	C9	leave
123f	7404	je short 00001245	1283	894C2418	mov [esp+18], ecx
1241	75F8	jnz short 0000123B	1287	61	popad
1243	EB67	jmp short 000012AC	1288	FFE1	jmp ecx
1245	29F6	sub esi, esi	128a	90	nop
1247	29C9	sub ecx, ecx	128b	90	nop
1249	B175	mov cl, 75	128c	90	nop
124b	46	inc esi	128d	59	pop ecx
124c	49	dec ecx	128e	51	push ecx
124d	75FC	jnz short 0000124B	128f	C3	retn
124f	E839000000	call 0000128D	1290	90	nop
1254	81C14A000000	add ecx, 4A	1291	8B442410	mov eax, [esp+10]
125a	51	push ecx	1295	8F80B8000000	pop dword ptr[eax+B8]
125b	31FF	xor edi, edi	129b	31C0	xor eax, eax
125d	81CF3C240000	or edi, 243C	129d	C3	retn
1263	8A01	mov al, [ecx]	129e	90	nop
1265	6629F0	sub ax, si			
1268	8801	mov [ecx], al			

The *call* of the address 0x1225 is detected as the *seed* in the STEP1. At this time next instruction address 0x122a of current program counter is stored in the virtual stack space. Current program counter is updated to the address 0x1239 by the recursive disassembly. In the STEP2, the recursive disassembly has been being from the address 0x1239 till 0x124d in order to detect the register loading 0x122a. In Table 1, these instructions may be performed for some purposes. However, these can be also used to carry out the role of the iterative statement which there is no with meaning that this part hangs an overload on the detection system in which an attacker uses the method for emulating.

Our method moves by the destination address through the recursive disassembly firstly, if it meets the conditional jump. However, because of moving by the next instruction if the method again meets the condition jump of the same address, it does not come off in these traps. In the address 0x124f, because the *seed* instruction again came out, current detection try is over.

The *call* of address 0x122a is detected in the STEP1 because the *seed* is detected per a byte. The instruction of address 0x1291 moves the value of the stack area in the specific register. After that, our method records an *eax* as the state where the instruction of the address accesses the stack area. The instruction of the address 0x1295 stores the program counter value into [*eax* + *B*8]. However, because the tracing is impossible by the static analysis, current detection try is over.

The next detected *seed* is the *call* of address 0x124f. Our method sets the virtual stack pointer up as the middle position of stack and stores 0x1254 in the virtual stack space. In the address 0x128d, the STEP2 in which the value is loaded in an *ecx* is detected. It is now altogether traced as the movement of the value relating to an *ecx* by the method of STEP3. By the *retn* of address 0x129d, current program counter is updated to the next instruction address 0x1254 stored in advance when the *call* was detected. By analyzing next instructions, since it is confirmed that 0x1254 is used in the calculation accessing the memory in the *mov al*, [*ecx*] of address 0x126e, finally our method can decide that these instructions are included in the decryption routine of the polymorphic shellcode.

3. Evaluations

In the prototype program used in this evaluation, the branch stack is added to process recursively the branch instruction. If the decryption code is finally not detected starting from a branched address, the detection process is continued from the address of next instruction, which is pushed on the branch stack, of the branch instruction.

In Table 2, we evaluated the detection rate against the famous polymorphic engines referred in recent papers. For each engines, 10 polymorphic instances were generated.

In the case of processing cost this method has not been optimized yet, however, our implementation demonstrates a linear relationship between the input data size, and running time. Moreover, because of performing only the simple static analysis, it is a reasonable fact that the performance of this method is better than the other methods that the emulation technique is included.

Table 2 The detection evaluation against polymorphic engines.

Engines		Detection	Detectable Method
ADMmutate		All detected.	
Clet		“	
Alpha2		“	
TAPiON		No detected.	Emulation needed.
Metasploit	Shikata_Ga_Nai	All detected.	
	jmp_call_additive	“	
	call4_dword_xor	“	
	fnstenv_mov	“	
	Countdown	“	
	Nonalpha	“	
	alpha_mixed	“	
	nonupper	“	
	alpha_upper	“	
	unicode_mixed	“	
	avoid_utf8_tolower	No detected.	Emulation needed.

4. Conclusions

We have considered the overhead problem of detecting the polymorphic shellcode with the static analysis resistant techniques. The starting point for this work is to detect whether the program counter value which the decryption routine stores through the static analysis method is used in accessing the memory or not.

Through our analysis of the real detected polymorphic shellcode and other evaluations, the method of this paper shows that it can detect the polymorphic shellcode in which the disassembly thwarting and self-modifying code techniques are used. Moreover the performance property is also better than other methods.

Acknowledgments

This work was sponsored by the Korea Ministry of Information and Communication under the Zero-day Attack Signature Management INfrastructure (ZASMIN) Project.

References

[1] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” Proc. International Symposium on Recent Advances in Intrusion Detection (RAID), Sept. 2005.

[2] X. Wang, C. Pan, P. Liu, and S. Zhu, “SigFree: A signature-free buffer overflow attack blocker,” Proc. 15th USENIX Security Symposium, pp.225–240, July 2006.

[3] R. Chinchani and E. Berg, “A fast static analysis approach to detect exploit code inside network flows,” Proc. 8th International Symposium on Recent Advances in Intrusion Detection (RAID), pp.284–308, Sept. 2005.

[4] M. Polychronakis, K. Anagnostakis, and E. Makatos, “Network-level polymorphic shellcode detection using emulation,” Proc. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2006.

[5] M. Polychronakis, K. Anagnostakis, and E. Makatos, “Emulation-based detection of non-self-contained polymorphic shellcode,” Proc.

International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.

[6] Q. Zhang, D.S. Reeves, P. Ning, and S.P. Lyer, "Analyzing network traffic to detect self-decrypting exploit code," Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2007.