A Polymorphic Shellcode Detection Mechanism in the Network*

Hsiang-Lun Huang, Tzong-Jye Liu, Kuong-Ho Chen†, and Chyi-Ren Dow
Department of Information Engineering and Computer Science
Feng Chia University
Taichung, Taiwan, R.O.C.
{m9405100, tjliu, crdow}@fcu.edu.tw
†cyne@pluto.iecs.fcu.edu.tw

Lih-Chyau Wuu
Institute of Computer Science and Information Engineering
National Yunlin University of Science and Technology
Yunlin, Taiwan, R.O.C.
wuulc@yuntech.edu.tw

ABSTRACT
Buffer overflow attack is a major security problem in recent years. The polymorphism technique for shellcode becomes more and more popular along with development of Internet. This paper proposes a method to detect the polymorphic shellcode for Windows operating system. The proposed approach relies on an IA-32 CPU emulator that executes instruction sequences and analyze the behavior of polymorphic shellcode. The experimental results show that the approach is able to detect polymorphic shellcode accurately.

Categories and Subject Descriptors
C.2.0 [**Computer-Communication Networks**]: General-*Security and protection (e.g., firewalls)*

General Terms
Security

Keywords
Buffer overflow, intrusion detection system, polymorphic shellcode.

1. INTRODUCTION
As the popular of Internet, the network attack becomes a big problem. Many software packages today are implemented using C/C++, which may generate some bugs due to the flexible syntax of C/C++ language. The hackers are able to exploit these bugs to attack vulnerable hosts.

Buffer overflow vulnerabilities were in 10 of the 31 advisories published by CERT [9] in 2002 and 17 of the 28 advisories published by CERT [10] in 2003. The well-known examples of the remote code injection are the buffer overflow vulnerabilities such as the Code Red [8] worm in 2001, the W32/Blaster [11] worm in 2003 and the Sasser [17] worm in 2004. These worms attack many network hosts and propagate through Internet. They cause serious threats on Internet.

Several solutions have been proposed to solve the buffer overflow attacks, which can be divided into two classes: host-based solutions and network-based solutions. Host-based solutions are either compile-based such as [13], [14] or hardware-based techniques such as [22]. For network-based solutions such as [2], [4], [5], [6], [24], [27], they detect the malicious codes in the network packets sent by the attackers.

In this paper, we focus on the network-based solution. We propose an algorithm for detecting the malicious codes in the network. For such purpose, analysis concerning the code reusing behavior of polymorphic shellcode was done using emulation. The proposed approach relies on an IA-32 CPU emulator that executes instruction sequences. Experimental results show that the detector is able to execute safely and catch common polymorphism techniques for shellcode accurately.

The rest of this paper is organized as follows. First, we present the related researches about the prevention of buffer overflow attack in Section 2. In Section 3, we discuss the common conception of buffer overflow attack. The layout of the shellcode and obfuscation techniques for shellcode are also discussed. In Section 4, the proposed network-based solution to detect polymorphic shellcode is introduced. In Section 5, the experimental results are introduced. Conclusion is given in Section 6.

2. RELATED WORKS
As was mentioned in the previous section, the solutions to solve the buffer overflow attack can be divided into two classes. First, we discuss host-based solutions. Then, we discuss network-based solution.

2.1 Host-based Solutions
StackGuard [13] is based on the compile technique. It adds many random values, called *canary*, next to the return address. It detects the change of the return address to check if some canary word has been modified when the function is called and before the function returns.

Return Address Defender (RAD) [12] is a simple compiler patch. It creates a safe area to store the return address of a program stack. Then, it automatically duplicates these return addresses to this area. The main idea is to compare the original return address with the duplicate one. An attack is detected if the return address is

* This research was supported by the National Science Council of the Republic of China under the Contract NSC-95-2221-E-035-071.

Infoscale'07, June 6–8, 2007, Suzhou, China.

modified. RAD is also operated on the compile-time. So, programmers do not modify the source code; they just have to recompile the source code.

2.2 Network-based Solutions

Network-based solutions are mostly signature-based. Signature-based network intrusion detection system (NIDS) detects attacks by comparing the packet strings with a signature database that describes the known attacks. If a sequence of bytes match the signature, an alert is generate.

Buttercup [23] is a network-based solution. It attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return addresses for existing buffer overflow vulnerabilities.

Some existing detecting mechanisms focus on detecting *sled* component. A *sled* is a string of bytes that are either NOP instructions or NOP-equivalents instructions. Toth and Kruegel proposed the Abstract Payload Execution (APE) [27] method and P. Akritidis et al. proposed the STRIDE [2]. These methods rely on disassembling the IA-32 CPU instructions to check the existence of a NOPs sled or a NOP-equivalents sled.

In addition, some approaches employed *emulation technique* to detect the buffer overflows attacks. For example, Michalis Polychronakis et al. [24] describe a network-level polymorphic shellcode detection engine by emulating general IA-32 CPU instructions to run the shellcode. This method is to set a payload reads threshold (PRT) to check if the number of memory accessing is greater than PRT or not.

In [4], [5], [6], the authors proposed the network-based code injection attacks by using sandboxing technique. This method traces the system call by using ptarce and Detours on Linux and Windows respectively to detect if the packets have the buffer overflow attacks or not.

3. BACKGROUND OF BUFFER OVERFLOW ATTACK

This section discusses the technique to cause the stack buffer overflow attacks and the layout of the shellcode. We also discuss the techniques to evade the intrusion detection system.

3.1 The Concept of Stack Buffer Overflow Attacks

If a programmer declares a buffer and accesses it without checking its boundary, a bug is generated such that attackers can exploit it to attack and change the normal execution flow to execute attackers' code. Shown in Figure 1 are the memory maps that are before and after the attack.

Since programmers did not check the boundary of the buffer, attackers can inject code into the buffer. Since the size of the injecting code is larger than the buffer size, overflow is guaranteed to be caused, and injected code will overwrite local variables in higher memory addresses that contain the stack pointer frame and the return address which, if modified, will change the normal execution. Since the return address will be pop to change the program counter register after the function call is finished, the program counter will point to the address that is overwritten by the attacker. This attack is called the *Buffer Overflow Attack*. A tutorial for the buffer overflow attacks was provided by Aleph One [3].

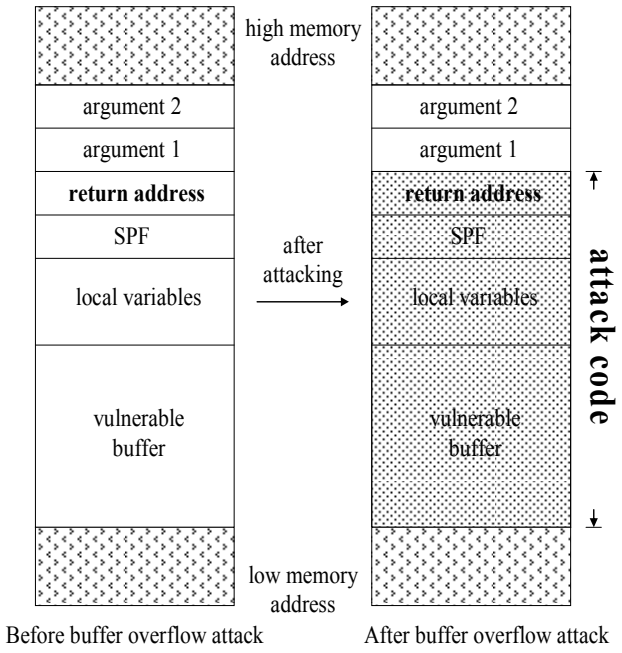

Figure 1. Memory maps of buffer overflow attack.

3.2 The Architecture of Attack Codes

Figure 1 is the actions that attackers want to perform after attacking. An attack code attempts to spawn a shell (hereby called *shellcode*) to gain the complete control of the target system. Attackers can construct a shellcode to perform arbitrary actions under the privileges of the buggy service. Figure 2 shows the layout of shellcode. Although the location of the code injection, relative to the start of the vulnerable buffer, is known to the attackers, it is only approximately known because the location varies between systems, even for identical executable programs. So attackers need to place a sequence of NOP (no-operation) instructions to increase the probability of the execution of the shellcode. This sequence is usually called a sled. The length of a sled is usually ranged around a few hundred bytes.

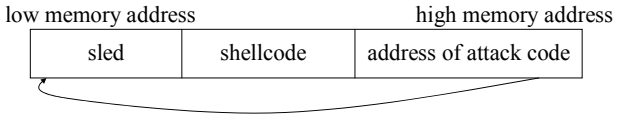

Figure 2. The layout of shellcode.

After the overwritten return address is popped, if attackers do not place the sled, the probability of error is very high because the overwritten return address is unable to jump to the accurate location to execute the shellcode. If attackers append the shellcode fragment to the sled, the overwritten return address may jump to execute NOP instruction. After executing a sequence of NOP instructions, the program counter will go to the start of shellcode and then perform the actions of attacks. We will describe techniques for sled component in the following subsection.

Table 1. List of IA-32 one-byte nop-equivalents instructions [30].

Code (Hex)	Opcode	Code (Hex)	Opcode	Code (Hex)	Opcode	Code (Hex)	Opcode
27	DAA	4A	DEC EDX	57	PUSH EDI	95	XCHG EBP,EAX
2F	DAS	4B	DEC EBX	58	POP EAX	96	XCHG ESI,EAX
37	AAA	4C	DEC ESP	59	POP ECX	97	XCHG EDI,EAX
3F	AAS	4D	DEC EBP	5A	POP EDX	98	CWTL
40	INC EAX	4E	DEC ESI	5B	POP EBX	99	CLTD
41	INC ECX	4F	DEC EDI	5D	POP EBP	9B	FWAIT
42	INC EDX	50	PUSH EAX	5E	POP ESI	9C	PUSHF
43	INC EBX	51	PUSH ECX	5F	POP EDI	9E	SAFH
44	INC ESP	52	PUSH EDX	60	PUSHA	9F	LASHF
45	INC EBP	53	PUSH EBX	90	NOP	F5	CMC
46	INC ESI	54	PUSH ESP	91	XCHG ECX,EAX	F8	CLC
47	INC EDI	55	PUSH EBP	92	XCHG EDX,EAX	F9	STC
48	DEC EAX	56	PUSH ESI	93	XCHG EBX,EAX	FC	CLD

3.3 One-byte NOP-equivalents Sled

A sequence of NOP instructions is detected easily by intrusion detection systems because this signature is easy to generate. Experienced attackers may use some instructions that do not affect the operation of program to replace NOP instructions. For example, the ADMmutate [19] engine uses this technique with a list of 52 one-byte NOP-equivalents instructions to generate the sled. Table 1 shows the list of 52 one-byte NOP-equivalents instructions under IA-32 architecture [30].

3.4 Multi-byte NOP-equivalents Sled

Attackers can use one-byte NOP-equivalents sledge to evade detection technologies that scan NOP-based. Attackers can also use multi-byte NOP-equivalents sledge to evade detection. However, it is impossible to use any multi-byte NOP-equivalents instructions available in the IA-32 instruction set because a sled must be executable at every offset. Figure 3 shows an example of multi-byte NOP-equivalents sled [1].

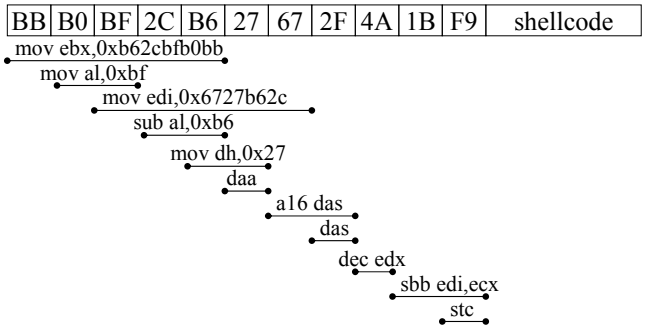

Figure 3. Multi-byte NOP-equivalents sled example [1].

If the execute flow is transferred to the leftmost byte, it will execute following instructions sequentially: "*mov ebx,0xb62cbfb0ff*", "*daa*", "*a16 das*", "*dec edx*" and "*sbb edi, ecx*". Any byte of sled can correspond to the opcode of IA-32 instructions. Therefore, when the execute flow is transferred to any byte of sled, it always executes correctly and the program counter will come to the start of shellcode.

3.5 No Sled

In addition, dark spyrit [16] had presented a method that use known locations of "JMP ESP" in system memory to transfer instruction pointer to the start of shellcode. This method is to find the system memory address of existing code within a dynamically linked library or the static program binary. This code is disassembled to a call such as "JMP ESP". The attack is called the *register spring* [15].

Attackers can use the memory address of code of register spring to overwrite the original return address. Therefore, attackers do not need to place sled because the vulnerable program will execute the shellcode directly when the program counter jump to memory address of "JMP ESP" and execute this instruction. The operation is shown in Figure 4.

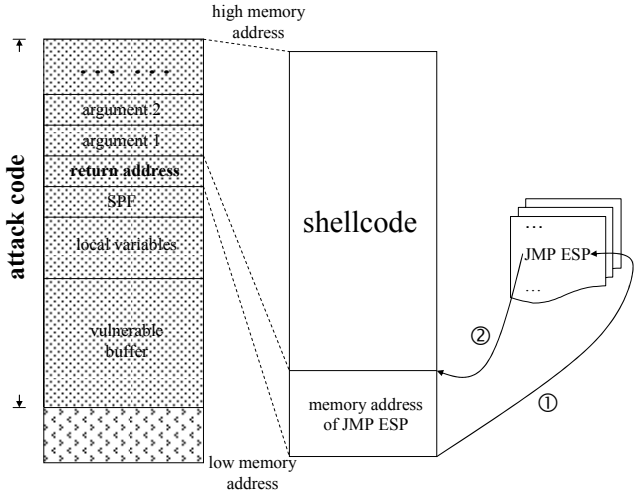

Figure 4. The operation of register spring.

In Figure 4, the first step transfers the execution flow to system address of the "JMP ESP" instruction and pop; the current ESP register points to the start of the shellcode. The second step is to execute the "JMP ESP" instruction. After executing, the execution flow will sprint to shellcode and execute it.

3.6 Polymorphism and Obfuscation Techniques

Obfuscation techniques for shellcode will be discussed in this subsection.

Plain shellcode usually needs to face some challenges. First, it maybe has some different restricted characters such that every buffer overflow point is different. Most buffer overflow cases are

to exploit some function that terminates until encountering a terminating character such as *strcpy*, *strcat*, *strncpy*, *strncat*, etc. Therefore, shellcode cannot contain terminating character. It is because that the null-termination byte make the shellcode cannot completely send to vulnerable program. Second, the plain shellcode may be detected by intrusion detection system.

Experienced attackers may use some tricks to solve the challenges of shellcode. Most popular ways are to encrypt the shellcode. It is called the *Polymorphic Shellcode*. Attackers can select one value and use this value to exclusive-or (XOR) each byte of shellcode to eliminate the null-termination byte or encrypt the shellcode. Therefore, using this way can cause NIDS obfuscate because the pattern of the malicious code are encrypted.

Figure 5 shows an example of polymorphic shellcode. Line 0005 and line 0007 are FPU instructions. When line 0007 is executed, the FPU instructions pointer will be saved and this pointer points to the last FPU instruction before *fsetenv/fnsetenv* [18] instruction. This pointer stores the value 0005 after executing fnsetenv instruction. A series of actions in Figure 5 are to get the program counter to be the base address to relocate address. The detail is discussed in "History and Advances in Windows Shellcode" [26].

```
0000   33 C9                    xor      ecx,ecx
0002   83 E9 DD                 sub      ecx,0FFFFFFDDh
0005   D9 EE                    fldz
0007   D9 74 24 F4              fnstenv  [esp-0Ch]
000B   5B                       pop      ebx
000C   81 73 13 F1 F1 59 06     xor      dword ptr [ebx+13h],659F1F1h
0013   83 EB FC                 sub      ebx,0FFFFFFFCh
0016   E2 F4                    loop     000C
0018
. . .  < encoded payload >
```

Figure 5. An example of decrypt part of shellcode.

4. THE PROPOSED APPROACH

The proposed approach detects the malicious code in packet streams by attempting to execute the network packets in a virtual environment, in which executable codes, if any, would be detected. There are many kinds of virtual environment, e.g., Xen [29], QEMU [25], Bochs [7], and VMware [28], which are standalone virtual machine environment. In this paper, we develop a simple emulator to emulate some general IA-32 CPU instructions. This way, malicious codes can be executed safely on the detector and do not affect the target operating system. Figure 6 shows the proposed detection algorithm.

```
01    for ( pos = Buf_Start ; pos < Buf_Len; pos++ )
02    {
03        reset_register( );
04        Init( );
05        ExecutedCount = 0;
06        for ( IP = pos ; IP < Buf_Len && ExecutedCount < 2048; IP += Len )
07        {
08            Len = Disasm_and_Emulate( buf [ IP ] );
09            if ( CodeReuse && MemoryAccess && ExecutedCount >= 5 )
10                return TRUE;
11            else if ( MemoryAccessViolation )
12            {
13                ExecutedCount = 0;
14                break;
15            }
16        }
17    }
18    return FALSE;
```

Figure 6. Pseudo-code for the detection algorithm.

Since register spring technique was presented by dark spyrit [16], the solutions that detect sled component may be bypassed. As the polymorphic engines are gaining their popularities, here the primary focus is on the detection of polymorphic shellcode. We aim at decrypt component of shellcode to analyze its behavior.

Network packet streams are sequences of hexadecimal signs which cannot be intuitively distinguished between data and code; since the entry of shellcode is unknown, the system will emulate the instruction from the start of buffer. Line 1 in Figure 6 is a loop to select a position to be the entry of shellcode. In first loop, the *reset_register* function is to assign the negative one to all general-purpose registers. The Init function is to initiate all state of the virtual process. Line 2 is the second loop to perform a series of actions of fetching, decoding and emulating. The *Disasm_and_Emulate* function in line 8 is implemented to disassemble and emulate the packet stream. It will return the length of current instruction. Emulator will break and shift to next byte from original entry to be a new entry and fetch and decode instructions from new entry again if hex signs are data instead of code because hex signs that are data may cause memory access violation. The *MemoryAccessViolation* is set nonzero if it execute from a wrong entry. Therefore, we shift each byte to be the entry of shellcode could find the correct entry accurately if the network stream contains the polymorphic shellcode.

The layout of polymorphic shellcode is to place a decoder before the shellcode. The decoder decrypts the encoded shellcode. There are two signatures in decryption process. First, in order to decrypt the encoded shellcode, the decoder will execute the same code repetitively, called *code reuse*. Second, the decoder will access memory in decryption process. Therefore, we get these two signatures to be detection condition.

In *Disasm_and_Emulate* function, we emulate IA-32 instructions to execute the contents of the received packet streams. Code that is written by attackers can be executed correctly. In contrast, data cannot disassemble and will get error when it executes. So, we will check if the behavior of code is a code reuse or not. The proposed system will alert an alarm if the detector detects a code reuse or a loop behavior.

However, common data maybe has the code reuse behavior as same as the real code. So, we set a threshold to decrease false positive. The threshold is the counter of continuous executable instructions. The minimal number of executable decryption instructions is 7 in Figure 7.

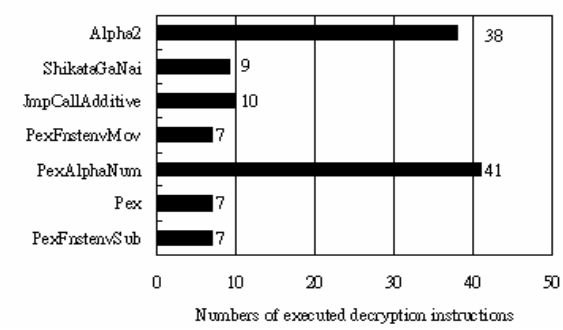

Figure 7. Numbers of executed decryption instructions of each encryption engines.

We take the *Alpha2*, *ShikataGaNai*, *JmpCallAdditive*, *PexFnstenvMov*, *PexAlphaNum*, *Pex* and *PexFnstenvSub* shellcode encryption engines of the Metasploit Framework [20] to count their respective amount of executed decryption instructions.

On the other hand, the execution of the polymorphic shellcode produced by *Alpha2* encryption engine or *PexAlphaNum* encryption engine takes more then 10 instructions due to these two engines produce alphanumeric shellcode.

It was found that the minimal instruction count of decryption part of polymorphic shellcode is 5. Figure 8 is an example of minimal instruction counts which contains 5 instructions. It was assumed that the polymorphic shellcode is to encrypt a block plain shellcode, not just to encrypt certain instructions. A decoder must have at least 5 instructions to decode a block of encoded payloads because it uses two instructions (line 0000 and line 0005) to get the program count to be the base address for relocate address. One instruction would initiate register ECX (line 0006) to set the length of plain shellcode because this case decodes payloads from the end of polymorphic shellcode. Then, one instruction (line 000B) would decode the encoded payloads. At the end, LOOP instruction (line 0010) would be used to jump to decode the encoded payloads repetitively. So we choose the value of 5 to be the threshold of *ExecutedCount*.

Hackers may use the current value of register ECX at the time of buffer overflow. Therefore, the decoder could eliminate the action of initiating in line 0006. This trick is only able to run when the value of register ECX is somewhat greater than the length of plain shellcode. If the value of register ECX is far greater than the length of plain shellcode, the operating system in vulnerable host will cause failure. Therefore, the risk of attacking failure is too high for attackers. So we do not consider this case.

```
0000   E8 00 00 00 00          call    0005
0005   5B                      pop     ebx
0006   B9 23 01 00 00          mov     ecx,0123h
000B   80 74 0B 0C 77          xor     byte ptr [ebx+ecx+0Ch],77h
0010   E2 F9                   loop    000B
0012
 .  .  .  < encoded payload >
```

Figure 8. An example of minimal instruction counts of decoder part of polymorphic shellcode.

If code reuse behaviors are detected in the contents by *Disasm_and_Emulate* function, the value *CodeReuse* will be set to nonzero and the value *ExecutedCount* is to count the number of executed code. Once the value *CodeReuse* is set nonzero and the value *ExecutedCount* is equal or greater than 8, the detector will set an alarm to inform the network administrator.

In order to avoid the detector occurring infinite loop or hang up, we use the execution threshold (XT) that Michalis Polychronakis et al. [24] proposed to be our threshold. If *ExecutedCount* is greater than 2048, the detector breaks the second loop.

5. THE PROPOSED ARCHITECTURE AND EXPERIMENTED RESULTS

The proposed approach modifies the disassembler of *OllyDBG* versions 1.04 [21] to become the IA-32 emulator. This package is an open source and includes the source code of 32-bit disassembler. We use this disassembler to decode instructions first. Then, we add function of emulation within disassembler to emulate IA-32 instruction set. We have implemented some general-purpose instructions and FPU instructions, but no MMX, SSE and SSE2 instructions. The emulator stops when it encounters an unimplemented instruction. The main purpose of the emulator is only to decode it and go to the next instruction.

We evaluate the ratio of accuracy of detection and its false positive ratio. The proposed system runs on a PC with a 3.0GHz Pentium 4 processor, 1GB RAM and the system operates in Windows XP Service Pack 2.

We generate the testing polymorphic shellcode by using the encoders of the Metasploit Framework [20]. Then, we take the polymorphic shellcode into a sequence of random values that we generate by the random number generator. The size of each testing data is 1024 bytes and the total numbers of testing samples are 1000. The encoders that we use in the experiment include *PexFnstenvSub*, *Pex* and *PexFnstenvMov*.

Table 2 shows the results of detection with polymorphic shellcode. The proposed detector catches the polymorphic shellcode that encode by above three encoders.

Table 2. The results of detection with polymorphic shellcode.

Encoder Name	Detected
PexFnstenvSub	Yes
Pex	Yes
PexFnstenvMov	Yes

In addition, we also test the false positive ratio. We generate a sequence of random data without polymorphic shellcode and take these data to be tested by the detection system. Table 3 shows the results of false positive, which is 5.2% of the total testing samples without *ExecutedCount* threshold. This is because the random data may contain the short branch instructions such as the opcode is E2 F4 (LOOP short address) or EB F4 (JMP short address). Like these short branch instructions, it will cause the false positive if the address of branch is before current branch instruction and fetch, decode instruction from the address which branched is not affect decode of original branch instruction. It has code reuse behavior in this situation and cause false positive. We may decrease the probability of false positive if we set the *ExecutedCount* threshold.

Table 3. The results of false positive without polymorphic shellcode.

Testing Random Data	
False Positive without ExecutableCount Theshold	***False Positive with ExecutableCount Theshold***
5.2 %	0 %

Figure 9 shows the performance of the detection algorithm. We insert the polymorphic shellcode into the random data. The x-axis is the percentage of the polymorphic shellcode in 1000 samples. The y-axis is the average detection time that the detector checks one sample. The performance can be improved. However, the accuracy ratio of detection and the false positive that discussed early are excellent.

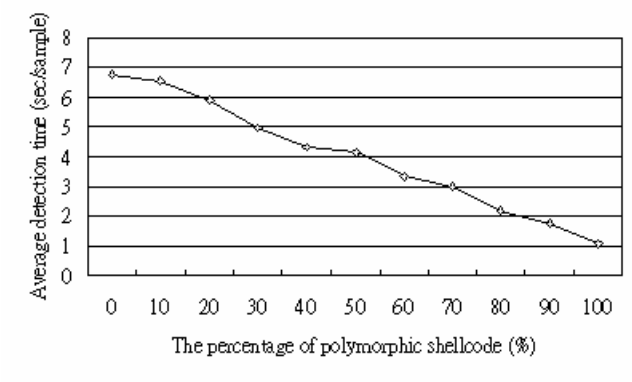

Figure 9. The performance of the detection algorithm.

6. CONCLUSION

The proposed system focuses on polymorphic shellcode. The proposed system detects the shellcode before executing actual payload. Currently, the proposed system cannot detect plain shellcode because it may uses some techniques to get address of system dynamic link library or call system API address directly. The proposed system is a network-based solution. It cannot get the operating system information.

In addition, the performance is not good due to shift each byte to check whether contain the polymorphic shellcode or not. Although most shellcode has no terminating character, the shellcode contains the terminating character in special case. In order to detect all types of shellcode, we need to survey and find the signature to increase performance. We will combine the network intrusion detection system with our algorithm to detect real network data.

In this system, we propose a network-based solution to analyze the behavior of polymorphic shellcode. The proposed detection algorithm is simple and clear. It detects accurately whether contain the shellcode or not. In addition, we can solve the problems of self-modify that bypass some previous solutions.

7. REFERENCES

[1] Advances in Exploit Technology. Go online to http://www.metasploit.com/confs/core05/core05_metasploit.pdf, 2005.

[2] Akritidis, P., Markatos, E. P., Polychronakis, M., and Anagnostakis, K. G. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference*.

[3] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, vol. 14, no. 49, Go online to http://www.phrack.org/archives/49/P49-14, Nov. 1996.

[4] Andersson, S., Clark, A., and Mohay, G. Network based buffer overflow detection by exploit code analysis. In *Proceedings of AusCERT Asia Pacific Information Technology Security Conference (AusCERT2004): R&D Stream*, Gold Coast, Australia, 2004. University of Queensland. ISBN: 1-86499-774-5.

[5] Andersson, S., Clark, A., and Mohay, G. Detecting network-based obfuscated code injection attacks using sandboxing. In *Proceedings of AusCERT Asia Pacific Information Technology Security Conference (AusCERT2005): Refereed R&D Stream*, Gold Coast, Australia, 2005. University of Queensland. ISBN: 1-86499-799-0.

[6] Andersson, S., Clark, A., Mohay, G., Schatz, B., and Zimmermann, J. A framework for detecting network-based code injection attacks targeting Windows and UNIX. In *Proceedings of the 21st Computer Security Applications Conference*, Dec. 2005.

[7] Bochs: the Open Source IA-32 Emulation Project (Home Page). Go online to http://bochs.sourceforge.net/.

[8] CERT Coordination Center. CERT Incident Note IN-2001-08 Code Red Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. Go online to http://www.cert.org/incident_notes/IN-2001-08.html, June 2001.

[9] CERT Coordination Center. CERT Coordination Center Advisories for 2002. Go online to http://www.cert.org/advisories/#2002, 2002.

[10] CERT Coordination Center. CERT Coordination Center Advisories for 2003. Go online to http://www.cert.org/advisories/#2003, 2003.

[11] CERT Coordination Center. CERT Advisory CA-2003-20 W32/Blaster worm. Go online to http://www.cert.org/advisories/CA-2003-20.html, Aug. 2003.

[12] Chiueh, T. C. and Hsu, F. H. RAD: a compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pp. 409-417, Apr. 2001.

[13] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, San Antonio, Texas, pp. 63–78, Jan. 1998.

[14] Cowan, C., Beattie, S., Johansen, J., and Wagle., P. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Usenix Security Symposium*, Aug. 2003.

[15] Crandall, J. R., Wu, S. F., and Chong, F. T. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusion and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[16] Dark spyrit AKA Barnaby Jack. Win32 Buffer Overflows. Go online to http://www.phrack.org/archives/55/P55-15, Sept. 1999.

[17] F-Secure Corporation. F-Secure Virus Descriptions: Sasser. Go online to http://www.f-secure.com/v-descs/sasser.shtml, May 2004.

[18] Intel® 64 and IA-32 Architectures Software Developer's Manuals vol. 1-3. Go online to http://www.intel.com/products/processor/manuals/index.htm.

[19] K2. ADMmutate. Go online to http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[20] Metasploit project. Go online to http://www.metasploit.com/.

[21] OllyDBG disassembler. Go online to http://www.ollydbg.de/disasm.zip.

[22] Özdoganoglu, H., Vijaykumar, T. N., Brodley, C. E., Kuperman, B. A., and Jalote, A. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. In *Proceedings of the IEEE Transactions on Computers*, pp. 1271- 1285, Oct. 2006.

[23] Pasupulati, A., Coit, J., Levitt, K., Wu, S. F., Li, S. H., Kuo, J. C., and Fan, K. P. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *IEEE/IFIP Network Operation and Management Symposium*, May 2004.

[24] Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Jul. 2006.

[25] QEMU (Home Page). Go online to http://fabrice.bellard.free.fr/qemu/.

[26] Sk. History and advances in windows shellcode. Phrack Magazine, vol. 7, no. 62. Go online to http://www.phrack.org/archives/62/p62-0x07_Advances_in_Windows_Shellcode.txt, Jun. 2004.

[27] Toth, T. and Krügel, C. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advances in Instrusion Detection (RAID)*, pp. 274-291, 2002.

[28] VMware: Virtualization, Virtual Machine & Virtual Server Consolidation (Home Page). Go online to http://bochs.sourceforge.net/.

[29] XenSource: Delivering the Power of Xen (Home Page). Go online to http://www.xensource.com/.

[30] XFOCUS team. NOP Equivalent opcodes for shellcodes - Canonical List. Go online to http://www.xfocus.org/articles/200203/363.html, Mar. 2002.