Shellcode Location Based on Register Information Flow

Yu Song¹, Hongyu Kuang¹ and Jian Wang¹

¹ College of Electrical Science, National University of Defense Technology, Changsha, Hunan, 410000, China

*Corresponding author’s e-mail: jwang@nudt.edu.cn

Abstract. Due to the high detection accuracy of dynamic simulation in which network traffic is sent to the emulator as executable code to detect suspicious behaviour, static analysis has become less popular. Earlier static shellcode detection methods are mainly based on statistical method which focus on instruction abstract features or byte patterns. Although these methods can be used for detection, it can’t guarantee the detection effect. It is hard for both of them to locate the position of shellcode. However, a new static analysis based approach is proposed in this paper. The shellcode instructions rely heavily on contextual information, so we use static disassembly to capture the fine-grained malicious mode of registers. In order to enhance the detection performance, several rules are designed based on the characteristics of the shellcode. Experimental results show that our method has certain potential, which may show better effects when combined with other shellcode detection methods.

1. Introduction
With the rapid development of internet technology, the internet has become an inseparable part of our lives, but the risks have followed at the same time. Attackers invade personal privacy and attack our systems by malicious attacks through vulnerabilities, resulting in a large number of security threats.

The detection methods mainly focus on detecting the shellcode which is the key part of intrusion attack [1]. Although modern simulation execution methods can detect shellcode more accurately, static detection methods still play an important role due to its low requirements and high efficiency.

The early shellcode detection system is based on signatures, such as Snort [2] and Bro [3]. They extract common fragments of existing code, construct signature-based detection tags and use a centralized database to detect malicious code. However, they are based on known attacks, which can’t resist new attacks.

In the early shellcode, the return address is an essential part. Buttercup [4] uses heuristic rules to detect shellcode, which identifies possible memory address ranges for existing buffer overflow vulnerabilities. If the packet contains these addresses, a red/yellow flag may be raised. However, the false positive rate is high. According to the fact that the byte sequence probability between the data and the executable code is significantly different. Chen et al. combine markov model and the return address detection to capture the difference [5], so that the false positive rate is greatly reduced.

Daniel compares the opcodes frequencies of the sample set which is consists of malicious and non-malicious files. He founds that the distribution of malware opcodes is significantly different from that of non-malware [6]. Zhao et al. further propose the abstract instruction feature for shellcode detection [7]. They map the network traffic to low latitude space by extracting the instruction type, instruction
length, used register, and then support vector machine is used to classify malicious and non-malicious samples.

With the development of artificial intelligence, more neural networks and other algorithms are applied to shellcode detection. Onotu et al. pre-process the network traffic of fixed-length raw data simply and quickly, and then use neural network to classify samples which is mainly composed of three parts: DLL library, image and normal data [8]. Cheng, etc. obtain the API sequence through simulation, then combined with support vector machine for classification [9]. They use WinDbg to monitor shellcode, while Markov chain is used to extract sequence features. Finally, the experiment shows that their approach has a high detection accuracy as well as a low false positive rate.

To sum up, static analysis method still has unique advantages. Starting from the general characteristics of shellcode, we detect shellcode based on the information flow of internal registers. The rest of this paper is organized as follows: Section 2 describes the basic concepts of register information flow proposed in this paper. Section 3 is the main chapter which describes the overall detection framework and the design of each module. In section 4, we propose two optimization schemes to enhance the detection performance. Section 5 is the experimental part, using harmless and malicious samples to test the proposed method. Finally, in section 6, we summarize the work of this paper and plan for the future work.

2. Proposed method
Plain shellcode is mainly composed of NOP sled, payload and return address in the form of [NOP sled] [Payload] [Return Address]. The encrypted shellcode uses encryption algorithms to encrypt the payload, which is in the form of [NOP sled] [Decoder] [Encrypted Payload] [Return Address]. In order to execute the intrusion code successfully, encrypted payload need to be decrypted first. In the process of decryption, specific code snippet called GetPC is used to locate the current location. However, for both plain and encrypted shellcode, both NOP sled and return address are not essential component.

2.1 Register Information flow
In the existing static detection methods, attention is usually paid to the relationship between instructions or bytes, such as using byte transition probability [5] or abstract instruction features [7] to identify shellcode from network traffic. They capture the relevance of instructions, especially similar instructions or instructions that often appear in pairs. We agree that the logical relationship of instructions is the key to detecting shellcode, but the byte outline and instruction order are difficult to be the decisive factors.

![Spectral images to show shellcode bytes distribution](image)

Figure 1. Spectral images to show shellcode bytes distribution (a) Shikata Na Gai (b) jcadd (c) call4dword (d) fstenv (e) ADMmutate (f) CLET.

Experiment in [10] shows that different shellcode have different characteristics, but figure 1 points out that shellcode encoded by the same encoder share similarities, which can support the above ideas. However, at the end of the paper, the author also concluded that those methods can only be effective at the moment. The encrypted shellcode samples can cover a wide range of spaces, making it difficult for modeling methods to identify the decoder sequence especially when the malicious code is written
separately without tools. And because of the limitations of static methods, traditional methods are difficult to handle most situations.

After analysing a large amount of shellcode samples, we found that the registers seems to be a breakthrough for static detection. To meet the requirements of speed and convenience, the system provides a number of registers for programmers to use. Some have fixed functions such as ESP and EBP, while some are free to use such as EAX, EBX. After observation, we found:

- Malicious code is based on instructions while it is difficult for benign data to be decompiled into a sequence of instructions which have logic.
- Benign data usually failed to be disassemble due to the violation of the rules of use of the registers.
- Although there are multiple ways to reach the same function, the use of registers is compact.

The use of registers in malicious code follows certain rules, and an attacker manipulates registers through programming logic to complete the intrusion. Usually the registers are saved, assigned or cleaned before they can be used to decrypt the encrypted payload, participate in calculations or save parameters.

| :00000000 31DB | xor ebx, ebx |
| :00000002 8D430D | lea eax, dword ptr [ebx+0D] |
| :00000005 C380 | int 80 |
| :00000007 6631C0 | xor eax, eax |
| :0000000A D6BF | cmp mov st[0], st[7] |
| :0000000C 2BC9 | sub ecx, ecx |
| :0000000E B101 | mov cl, 01 |
| :00000010 D97424F4 | fstenv [esp-0C] |
| :00000014 8A | pop edx |
| :00000016 313512 | xor dword ptr [edx+12], eax |
| :00000018 034512 | add eax, dword ptr [edx+12] |
| :0000001B 83C204 | add edx, 00000004 |
| :0000001E 23F8 | loop 00000015 |

Figure 2. A disassembly shellcode example using fstenv as GetPC

The disassembled example in figure 2 can support our view. In this encrypted shellcode sample, register EBX is first cleaned register and then used as an address to assign the value to register EAX, then call system function. The second part cleans register ECX and assigns it. It performs self-localization through the floating-point operation instruction and pop the address to register EDX. Then XOR and ADD calculations are performed by using EDX as the memory address. In this sample, we can find that registers are cleaned and assigned before used, then the value will be "communicated" with the memory or other register, which is the meaning of register information flow proposed in this paper.

3. Detection framework

However, just detecting the register operation can only handle a part of the situation, so we designed several heuristic rules after analysing a large amount of shellcode samples. The detection framework of this paper is shown in figure 3, which contain return address match, GetPC detect, register extract, mode detection and so on.
Figure 3. The detection framework of the proposed method

We use the python disassembly engine capstone to disassemble network traffic and created a register dictionary to record the usage types of all general-purpose registers and special registers which contain EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP. According to the way the registers used in the instruction, we divided all operations into seven categories as shown in table 1.

Table 1. Different types of operation

| Value | Refer              |
|-------|-------------------|
| -1    | No operation      |
| 0     | Saved             |
| 1     | Cleared           |
| 2     | Assigned          |
| 3     | Source value      |
| 4     | Read position     |
| 5     | Write position    |
| 6     | Participate in calculation |

The detection steps are as follows:
Step 1: Traverse the data stream starting from the first byte, and perform step 2 for each location until the end of the stream is reached.
Step 2: Use a window of length N to intercept the traffic data and disassembling the fragments data.
Step 3: End the detection if the return address mode is found, otherwise goes to next step.
Step 4: if there is int 80 or GetPC in disassembly list, set the detection flag to 1 and then focus on the register related to them.
Step 5: Analyse the operation of all instructions one by one, and use a dictionary to record the operation of the corresponding register. If the operation is illegal or the analysis of the fragment has already finished, go to step 6.
Step 6: If the operations of the registers satisfy the malicious mode, judge the sample as shellcode and treat the current window offset as the starting position of the shellcode. Otherwise, return to step 1.

We use the linear disassembly method when disassembly, then perform recursive disassembly and continue analysis from the new position when we encounter CALL or JMP instruction with its jump address fell within a reasonable range. Next we will describe each module in detail.

3.1 Return address match
There are still shellcode that can't give up the return address part, and some encryption algorithms will produce similar features. When detecting the return address, we use a fence of 4 bytes length to split the data in the window. The threshold is set to 4 to determine whether there is a return address mode.

3.2 GetPC detect
In the context of modern intrusion technology, encryption is a common technique. For plain shellcode and encrypted shellcode, int 80 and GetPC play an important role in the code data. So we consider the following two situations:
- Plain shellcode usually contain int 80 system call, before that register EAX have already saved the system call number.
- For encrypted shellcode, call and fstenv are the current popular GetPC methods, so they can be treated as detection targets.

Corresponding to their function, EAX records the system call number and GetPC will pop the EIP address to a register which will used to decrypt the payload, etc. If the GetPC is found, then we focus on the related register in the remaining process. Due to the importance of the register, we define it as a key register which will play an important role in the next detection.

3.3 Register extract
According to the type of instruction, add the corresponding operation to its recording list. For example, EAX is first cleaned and assigned, then AL is participated in the calculation and EAX is used as read address after an instruction. It is recorded as (“EAX”: [1, 2, 6 -1, 4]) (“AL”: [1, 2, 6, 4]).

When recording the register operations, there are two issues. One is that it is impossible to use all registers within one instruction, the other is that the general purpose registers can be divided into two small registers. For example, EAX can be used as AL and AH.

For the above two issues, the following solution are adopted. In order to maintain the original sequence and interval of operations, if any record in the dictionary is changed within one instruction, then supplement other registers’ operation list with no operation type. At the same time, in order to reduce the complexity of the extraction, we extend all 16-bit registers in the instruction to the corresponding 32-bit registers. After the replacement and padding, it can be recorded as (“EAX”: [1, 2, 6 -1, 4]).

3.4 Illegal rules
When extracting the register operations, process should be stopped when any of the following illegal operations is encountered. Then send register operation dictionary to the malicious mode detection module and get the final judgment result. When current fragment is successfully extracted from start to finish, it will also follow these steps. Here are the illegal rules:
- Current instruction is a privileged instruction or illegal instruction which is shown in table 2.
- The jump address of the CALL and J class instructions is out of range, including JO, JS, JE, JNP, JMP, JECXZ and other similar instructions. In this paper, we limit the jump address to be within the data range, and the jump distance is not greater than 90.
- Register is used as memory address without assignment.
- Register is used as source value without assignment.
- Any direct operation on register ESP in addition to save operation and assign its values to other registers.
A large fixed immediate is used as memory addresses, while it is legal for a small one to be used as offset address.

| Instruction | Instruction | Instruction |
|-------------|-------------|-------------|
| LGDT        | INVD        | RDPMC       |
| LLDT        | WBINVD      | RDTSC       |
| LTR         | INVLPG      | Int3        |
| LIDT        | HLT         | CLI         |
| LMSW        | WRMSR       | STL         |
| CLTS        | RDMSR       | RET         |

However, it is also illegal when the register is assigned by an immediate value and then used as memory address. To do this, we maintain a dictionary of register sources to help us check the operation, where (-1-unassigned), (0-initialization), (1-immediate assignment), (2-getpc assignment). But registers ESP, EBP are a little different, their values are updated in real time while the program is running. Another crucial point is that the registers will inherit the source attributes from its origin register.

In the experiment we found that the analysis of benign traffic is often terminated due to the illegal instructions or illegal use of the register, which can greatly reduce the pressure of the detector, and effectively avoid false positives.

3.5 Malicious mode detection
In mode detection module, the input is the register operations dictionary. For any register, it must be cleaned or assigned before it can be used as address, source value or participate in calculation. We analyse the operating modes of the eight registers one by one and use the following rules as criteria

1. The number of registers that satisfy the malicious mode is more than one.
2. The related register satisfy the malicious mode while the GetPC flag is set to 1.
3. The fragment data can successfully execute to instruction int 80 while register EAX has already been assigned
4. Number of continuous operations of any register is larger than the threshold, in this paper we set the threshold to 5.

In order to improve efficiency, if the number of operations of the register is less than 4 (including the padding value), we regard it as harmless data and continue detecting.

4. Optimization
We extract malicious patterns from the disassembly results. However, disassembling from all locations results in significant overhead without equal benefits. According to [6], we analyse the opcodes of a large number of samples, and obtained the frequency distribution as shown in table 3.

From the table we can know that PUSH, MOV and XOR are the three most used instructions in shellcode. Whether in plain shellcode or encrypted shellcode, the above instructions are usually used to save original data, clean registers or initialize registers before operation. In addition, many basic functions are based on the above instructions, so in this paper we choose the disassembly position where start with these instructions. Moreover, SUB and GetPC are also taken into consideration, which can avoid missing detection opportunities. Because SUB is also a way to clean up, and GetPC code can be used without any pre-treatment.

| Opcode | frequency (%) | Opcode | frequency (%) | Opcode | frequency (%) |
|--------|---------------|--------|---------------|--------|---------------|
| push   | 20.016        | mov   | 11.1634       | push   | 20.8196       |
We found that some encryption algorithms pass register values through PUSH and POP, etc., and the detector can’t capture the associations between registers. Such confusion make it difficult for the detector to capture the real relationship between instructions, which leading to the increasing of the false negative rate.

Figure 4. Special sample of the exchange of register value

As shown in figure 4, the GetPC code uses EDI instead ESP, then uses EBP in the middle to pass the address value to ECX. However, the simplest original form should be (fstenv [esp-0xc]; pop ecx).

To do this, we use a list to maintain a virtual stack, correlate register information, and merge the corresponding register dictionary during analyse. In addition to capturing information exchanges between registers, it can also be used to monitor parameter save operations and so on. When GetPC code is detected, we push the string “GetPC” into the virtual stack. And we have added a new illegal rule: The current instruction is POP while there is nothing on the virtual stack.

5. Experiment

In order to verify the effectiveness of the proposed method, we use a variety of types of samples including CLET and ADMmutate, as well as original shellcode of the Linux platform contained in the MSF [11]. Various range of shellcode are included, such as connect back shellcode, add-user shellcode, and egg-hunt shellcode which are produced by the encryption engines of MSF. In order to supplement the data set, we also download some samples from the network database Shell-storm [12] and Exploit-DB [13].

The composition of the experimental data set and the result are recorded in table 4. From the table we can know that the proposed method can identify most shellcode, reaching an average accuracy of 94.75%, while the vast majority of benign data can be filtered. And what's more, our target function is to locate the start position of the shellcode.

However, due to the range of ASCII, the performance on ASCII data is less effective. ASCII data is easier to be disassembled into a reasonable instruction sequence and bypass the detector. In addition, there are also some shellcode encrypted by special algorithms that can’t be detected.

| Type       | Source        | Number | Detect | Accuracy (%) | Total (%) |
|------------|---------------|--------|--------|--------------|-----------|
| Shellcode  | Web and MSF   | 400    | 358/400| 89.5         | 94.75     |
|            | CLET          | 200    | 200/200| 100          |           |
|            | ADMmutate     | 200    | 200/200| 100          |           |
| Benign Data| ASCII data    | 200    | 182/200| 91           |           |
|            | Printable data| 200    | 200/200| 100          | 96.7      |
|            | Real traffic  | 200    | 198/200| 99           |           |
6. Conclusion
In this paper, we propose a new static shellcode detection method based on register information flow. Unlike previous methods based on the relationship between bytes or instructions, we capture the relationship between registers and check for their malicious mode. The Experiment result shows that our method can identify most shellcode and have the ability to locate the starting position of the shellcode.

However, it also has some shortcomings. Due to the imperfections in detection criteria, it can’t identify the shellcode accurately. But it may be improved by combining with other methods. In future work, we plan to improve the heuristic rules for detection and combine shellcode detection methods with our positioning approach, which is expected to achieve better detection performance.

References
[1] Polychronakis, M., Anagnostakis, K. G., Markatos, E. P. (2007). Emulation-Based Detection of Non-self-contained Polymorphic Shellcode. In: International Conference on Recent Advances in Intrusion Detection.
[2] Roesch, M. (1999). Snort - lightweight intrusion detection for networks. Proc.usenix System Administration Conf, 229--238.
[3] Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. Computer Networks, 31.
[4] Pasupulati, A., Coit, J., Levitt, K., Wu, S. F., Li, S. H., Kuo, J. C., et al. (2004). Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In: Network Operations & Management Symposium. IEEE.
[5] Payer, U., Kraxberger, S. (2005) Polymorphic Code Detection with GA Optimized Markov Models. In: Communications & Multimedia Security, Ifip Tc-6 Tc-11 International Conference, Cms, Salzburg, Austria.
[6] D. Bilar. (2007) Fingerprinting malicious code through statistical opcode analysis. In: ICGeS 07: Proceedings of the 3rd International Conference on Global E-Security, London (UK).
[7] Zhao, Z., Ahn, G. J. (2013). Using instruction sequence abstraction for shellcode detection and attribution. In: Communications & Network Security. IEEE.
[8] Onotu, P., Day, D., Rodrigues, M. A. (2015) accurate shellcode recognition from network traffic data using artificial neural nets. In: 28th canadian conference on electrical and computer engineering. Canada. 355-360.
[9] Cheng, Y., Fan, W., Huang, W., An, J. (2017) A shellcode detection method based on full native api sequence and support vector machine.
[10] Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., Stolfo, S. J. (2010) On the infeasibility of modeling polymorphic shellcode. Machine Learning, 81(2):179-205.
[11] Metasploit project. (2006) http://www.metasploit.com/.
[12] Salwan, J. Shell-storm, http://www.shell-storm.org/.
[13] Offensive Security. Exploit DB, http://www.exploit-db.com/.