A Shellcode Detection Method Based on Full Native API Sequence and Support Vector Machine

Yixuan Cheng[1, a)], **Wenqing Fan**[1, b)], **Wei Huang**[1, c)] **and Jing An**[1, d)]

[1]A College of Computer, Communication University of China, Beijing 100024, China.

E-mail: [a)] cuc2014is_bcxq@163.com, [b)]fan.winking@gmail.com, [c)]i@huangwei.me, [d)]anjingcuc@gmail.com

Abstract. Dynamic monitoring the behavior of a program is widely used to discriminate between benign program and malware. It is usually based on the dynamic characteristics of a program, such as API call sequence or API call frequency to judge. The key innovation of this paper is to consider the full Native API sequence and use the support vector machine to detect the shellcode. We also use the Markov chain to extract and digitize Native API sequence features. Our experimental results show that the method proposed in this paper has high accuracy and low detection rate.

1. Introduction

In today's era, computer viruses like Trojans and other malicious software, especially shellcode, continue to appear, bringing a huge threat to the development of Internet. Traditional detection method is based on signature. With the development and update of computer virus technology, signature code is virtually impossible to accurately identify malicious software, especially when malware has been confused or encrypted or after other operations.

Within up-to-date analysis methods, a common way is to extract information in the PE file header as its static feature[1], such as the number of dlls and APIs. However, these features are not sufficient to represent the actual behavior of a malicious program, and its PE file header information can be easily tampered with. As the behavior of malware is mostly based on API functions while shellcode is an instruction set with specific functions, its function is limited. Therefore, to complete the more complex functions, it often calls system API.

One way to detect shellcode is to use the Hook key API, such as LoadLibraryA(W), GetProcAddressA(W), CreateFilea(W), etc. The main problem is that a malicious purpose with unknown behavior shellcode does not just use this limited number of API functions, if it uses other non-Hook API function, the detection method will be invalid.

Another common method is based on the API call sequence to determine[3]. A typical approach is to obtain the call sequence, based on a supervised heuristic learning, to determine whether it is malware[4]. It can also improve the direction of the following two aspects:

1. Compared to Win32 API, the native APIs better reflect the core nature of a program because Win32 API functions often call the underlying native API;

2. The analysis process is static. It disassembles the program, and then generate the API call sequence based on the disassembly results, but the static analysis results are not equal to its actual running results. The actual API call sequence is what we really concerned about.

Aiming at the problem of some previous methods, this paper presents a shellcode detection method based on full Native API sequence and support vector machine. Using the WinDbg tool to monitor the whole Native API sequence information of the shellcode, the Markov chain is used to extract the sequence features, and the support vector machine is used to automate the judgment of the shellcode. The main contributions of this paper include:

1. Take a dynamic analysis method to analyze the program. We determine the final result by dynamically analyzing the whole API sequence information that the program actually calls at execution time.
2. Use the Native API full sequence. Compared to the Win32 API sequence, we use the native API sequences that represent the actual core functionality of the program as the data source for the decision. And we monitored the full sequence of Native API calls to ensure that it can monitor its complete behavioral information.
3. Apply the support vector machine to the detection of the shellcode. We use the support vector machine to judge the characteristics of the native API full sequence and apply it to the detection and judgment of the shellcode.

The remainder of this article is organized as follows. In the second part, we introduce some of the related work in the field of malware detection. In the third part, we describe the architecture and implementation of the shellcode detection system. In the fourth part, we present our experimental results, while concluding remarks are addressed in fifth part.

2. Related work

There are two common malware detection methods: static analysis and dynamic analysis.

The static analysis method is to acquire its signature, or to extract information of its PE head or data portion as its characteristics. Hu[5] used the static method to select the PE header information and the middle information as a feature to determine. Alazab[4,6] and Sathyanarayan[7] used the IDA tool to disassemble the program and extract the API call sequence from the disassembly results for analysis. Santos[9] used the opcode sequence to calculate the similarity of the two PE executables. Static method is the current mainstream detection method, but the static methods for current malware which often uses packaging and confusing technology cannot be detected.

The dynamic method performs discriminant detection by obtaining API call information when the program is running. Sami[8] proposed API calls that combine time characteristics with spatial features to monitor samples. Wagener[10] combined with virtual machine technology, first constructed two similar sequences of program API sequences, by calculating the similarity of the classification. Tian[2] and so on recorded the program in the virtual machine to run the log information, such as API call information, to classify it. The current mainstream dynamic analysis methods mainly dynamically run the target program, record the API information for further classification or judgment. But most of the analysis process to determine the information is based on Win32 API, not the Native API which can represent the core function of a program. So in this paper, we use Native API as basic information.

3. Approach

This section describes the architecture of the entire system. As shown in Fig.1, the entire system consists of three parts.

The Native API is a lightweight program programming interface used by Windows NT and user-mode applications that is implemented by the operating system and is provided in Ntdll.dll. Win32 API main function encapsulated in Kernel32.dll. Painting related system services packaged in Gdi32.dll. Window management is mainly encapsulated in User32.dll. Windows related controls encapsulated in comctl32.dll. Win32 API function in the specific implementation of the function, will call the more underlying relevant Native API function in Ntdll.dll. If the Win32 API uses the CreateFile() function, the NtCreateFile() function in the Native API is called inside the CreateFile() function, so the Native API is better to reflect the core functionality of the program. And because the Native API can be invoked directly by the application, the detection method based on the Win32 API will fail if the malware directly calls the Native API to implement the functionality without using the Win32 API, so in the method presented in this article, we use dynamic analysis methods and the full

Native API call sequence is used as the basis for subsequent judgments.

In the first part, there are two steps:

Step 1: Use scripts builder, combined with Native API prototype, to generate WinDbg debug scripts. WinDbg through the bp and other instructions to the function under the breakpoint operation. When hit the breakpoint, the program is in an interrupt state, so we can record the Native API related information, such as the name of the Native API and other information. So according to Native API prototype, we can generate automated debugging script, and the target program will automatically log information and so on to get its dynamic runtime information.

Step 2: On the target host, use the previously generated WinDbg script, using WinDbg to debug the shellcode.

In the second part, call the WinDbg script to debug the target program. Put the shellcode into the target host and run the shellcode. WinDbg records the full Native API information based on the previously generated debug script and generates a log file.

After obtaining the log file, it is necessary to extract its digital features to carry out the upcoming classification work. In this article, we take 4-Gram to characterize. Markov hypothesis think that the appearance of a word depends only on the finite word. And Markov chain is a discrete sequence satisfying the Markov hypothesis. Combining the previous studies with the Markov Chain applications in the API sequence and the experimental results, we believe that the 4-Gram algorithm to express the API sequence can achieve better results. The extraction algorithm is as follows:

Step 1: Extract the API sequence in each log file as input to the next step 4-Gram algorithm;

Step 2: In the 4-Gram algorithm, for each function, the probability of its occurrence is only related to the previous three functions, so we choose a sliding window of size 4. Using the number of occurrences of each Gram, the number of times as the characteristics of each Gram, we get the Gram characteristics group of each program;

Step 3: The highest number of n Grams in shellcode will be used as the key Gram characteristic vector of shellcode. In this paper, we select n as 40 as the number of Gram feature vectors;

Step 4: Select the Gram vector in the Gram feature vector of the shellcode extracted from step 3 from the Gram attribute group of each program to be analyzed as the Gram characteristic vector of the program to complete the feature extraction of each program.

In the third part, we use Support Vector Machines (SVM) to classify. SVM is a kind of supervised learning model with related learning algorithm in machine learning, which can classify and analyze the data. The main idea is to find a super-plane as a training sample segmentation, to ensure the smallest classification error rate. SVM allocates a given training set to one or more categories by constructing a non-probabilistic binary linear classifier. According to the Gram feature vector extracted in the second part, 50% of the training feature is selected as the training set, and the other 50% is used as the test set, and the support vector machine is used to classify. Finally, the results are judged. SVM needs to select the kernel function. After several experiments, we think that the RBF kernel function is better for the final classification. In this paper, RBF kernel function is chosen as the kernel function of SVM.

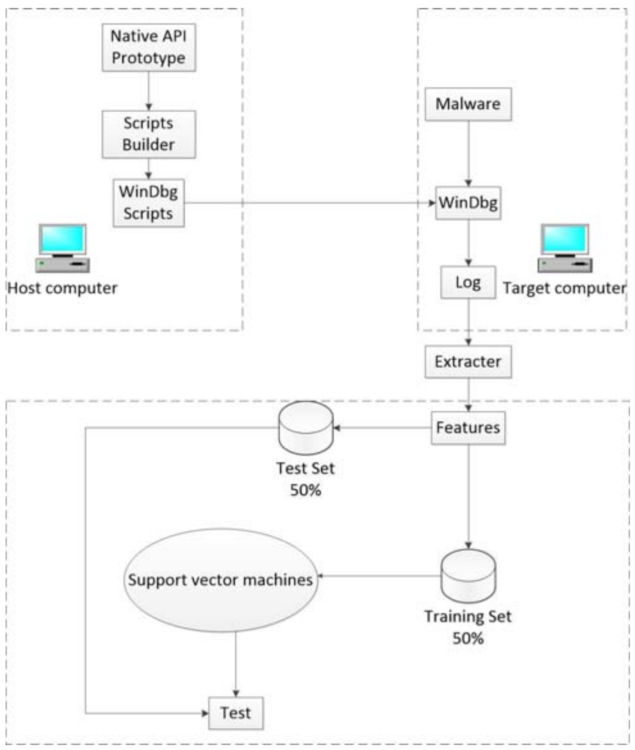
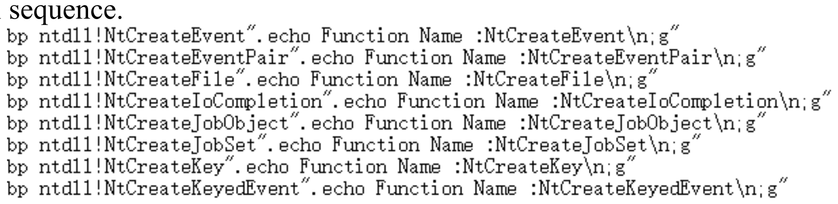


FIGURE 1. Architecture of shellcode detection system.

4. Experimental evaluation

The data sets for this article is made up of 18 shellcode programs on exploit-db and 72 PE files in authentic Windows XP system directories. In order to ensure that there is no deviation from the classifier in the course of training, we select 50% of the shellcode program and the benign program with the same number as the training set and the other part as the test set.

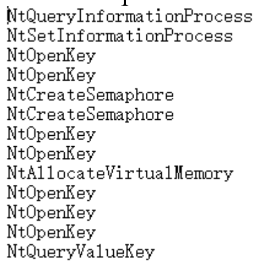
In the Scripts Builder, we generated the WinDbg script shown in Fig.2. We use the bp command to perform breakpoints on all functions in ntdll and output their function names to the log file as the full Native API call sequence.



```
bp ntdll!NtCreateEvent".echo Function Name :NtCreateEvent\n;g"
bp ntdll!NtCreateEventPair".echo Function Name :NtCreateEventPair\n;g"
bp ntdll!NtCreateFile".echo Function Name :NtCreateFile\n;g"
bp ntdll!NtCreateIoCompletion".echo Function Name :NtCreateIoCompletion\n;g"
bp ntdll!NtCreateJobObject".echo Function Name :NtCreateJobObject\n;g"
bp ntdll!NtCreateJobSet".echo Function Name :NtCreateJobSet\n;g"
bp ntdll!NtCreateKey".echo Function Name :NtCreateKey\n;g"
bp ntdll!NtCreateKeyedEvent".echo Function Name :NtCreateKeyedEvent\n;g"
```

FIGURE 2. Part of the script command.

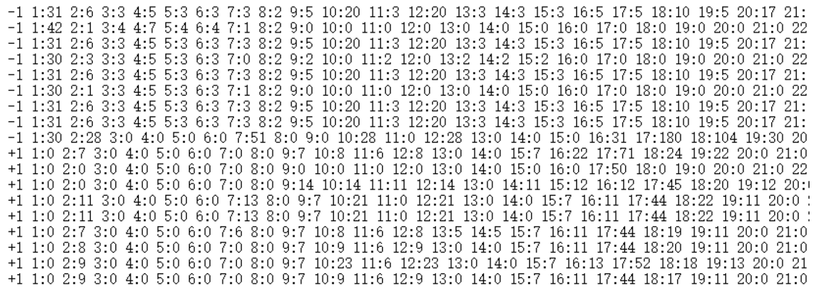
In Extracter, we first process the log file into a pure API sequence, as shown in Fig.3.



```
NtQueryInformationProcess
NtSetInformationProcess
NtOpenKey
NtOpenKey
NtCreateSemaphore
NtCreateSemaphore
NtOpenKey
NtOpenKey
NtAllocateVirtualMemory
NtOpenKey
NtOpenKey
NtOpenKey
NtQueryValueKey
```

FIGURE 3. Part of the API sequence.

According to the extraction algorithm in Section 3, the API sequence is processed into a Gram feature vector and the number 1-40 is used to represent the selected 40 feature vectors, as shown in Fig.4. We also count the number of feature vectors in the program as the value of the feature vector.



```
-1 1:31 2:6 3:3 4:5 5:3 6:3 7:3 8:2 9:5 10:20 11:3 12:20 13:3 14:3 15:3 16:5 17:5 18:10 19:5 20:17 21:
-1 1:42 2:1 3:4 4:7 5:4 6:4 7:1 8:2 9:0 10:0 11:0 12:0 13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22
-1 1:31 2:6 3:3 4:5 5:3 6:3 7:3 8:2 9:5 10:20 11:3 12:20 13:3 14:3 15:3 16:5 17:5 18:10 19:5 20:17 21:
-1 1:30 2:3 3:3 4:5 5:3 6:3 7:0 8:2 9:2 10:0 11:2 12:0 13:2 14:2 15:2 16:0 17:0 18:0 19:0 20:0 21:0 22
-1 1:31 2:6 3:3 4:5 5:3 6:3 7:3 8:2 9:5 10:20 11:3 12:20 13:3 14:3 15:3 16:5 17:5 18:10 19:5 20:17 21:
-1 1:30 2:1 3:3 4:5 5:3 6:3 7:1 8:2 9:0 10:0 11:0 12:0 13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22
-1 1:31 2:6 3:3 4:5 5:3 6:3 7:3 8:2 9:5 10:20 11:3 12:20 13:3 14:3 15:3 16:5 17:5 18:10 19:5 20:17 21:
-1 1:31 2:6 3:3 4:5 5:3 6:3 7:3 8:2 9:5 10:20 11:3 12:20 13:3 14:3 15:3 16:5 17:5 18:10 19:5 20:17 21:
-1 1:30 2:28 3:0 4:0 5:0 6:0 7:51 8:0 9:0 10:28 11:0 12:28 13:0 14:0 15:0 16:31 17:180 18:104 19:30 20
+1 1:0 2:7 3:0 4:0 5:0 6:0 7:0 8:0 9:7 10:8 11:6 12:8 13:0 14:0 15:7 16:22 17:71 18:24 19:22 20:0 21:0
+1 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0 12:0 13:0 14:0 15:0 16:0 17:50 18:0 19:0 20:0 21:0 22
+1 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:14 10:14 11:11 12:14 13:0 14:11 15:12 16:12 17:45 18:20 19:12 20:
+1 1:0 2:11 3:0 4:0 5:0 6:0 7:13 8:0 9:7 10:21 11:0 12:21 13:0 14:0 15:7 16:11 17:44 18:22 19:11 20:0
+1 1:0 2:11 3:0 4:0 5:0 6:0 7:13 8:0 9:7 10:21 11:0 12:21 13:0 14:0 15:7 16:11 17:44 18:22 19:11 20:0
+1 1:0 2:7 3:0 4:0 5:0 6:0 7:6 8:0 9:7 10:8 11:6 12:8 13:5 14:5 15:7 16:11 17:44 18:19 19:11 20:0 21:0
+1 1:0 2:8 3:0 4:0 5:0 6:0 7:0 8:0 9:7 10:9 11:6 12:9 13:0 14:0 15:7 16:11 17:44 18:20 19:11 20:0 21:0
+1 1:0 2:9 3:0 4:0 5:0 6:0 7:0 8:0 9:7 10:23 11:6 12:23 13:0 14:0 15:7 16:13 17:52 18:18 19:13 20:0 21
+1 1:0 2:9 3:0 4:0 5:0 6:0 7:0 8:0 9:7 10:9 11:6 12:9 13:0 14:0 15:7 16:11 17:44 18:17 19:11 20:0 21:0
```

FIGURE 4. Part of the training set.

We choose to use LIBSVM to achieve SVM classification. We choose RBF as the kernel function. The final judgment accuracy is 94.37%, with the false positive rate of 1.41%, and the false negative rate accounting for 44.44%.

5. Conclusion

The experimental results show that the shellcode detection method based on the full Native API sequences and the support vector machine has a high detection accuracy as well as a low false positive rate. Good results are achieved from benign programs. We still have defects: few experimental data. The reason for the high false negative rate is that the size of training sets is too small and the extracted Gram characteristic vector is not big enough. In the future improvement plan, we can increase the data sets size, especially the number of shellcode samples. We have carried out a preliminary verification of the proposed scheme. A high detection accuracy and a low false positive rate indicates that our program effect is effective.

Acknowledgments

We would like to thank Xinyuan Chen, Yue Pan and Yawen Song for their support, offering good ideas to help us finish this paper. Yixuan Cheng is with the Communication University of China.

References

[1] Schultz, M. G., Eskin, E., Zadok, F., & Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on (pp. 38-49). IEEE.

[2] Tian, R., Islam, R., Batten, L., & Versteeg, S. (2010, October). Differentiating malware from cleanware using behavioural analysis. In Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on (pp. 23-30). IEEE.

[3] Xu, J. Y., Sung, A. H., Chavez, P., & Mukkamala, S. (2004, December). Polymorphic malicious executable scanner by API sequence analysis. In Hybrid Intelligent Systems, 2004. HIS'04. Fourth International Conference on (pp. 378-383). IEEE.

[4] Alazab, M., Venkatraman, S., Watters, P., & Alazab, M. (2011, December). Zero-day malware detection based on supervised learning algorithms of API call signatures. In Proceedings of the Ninth Australasian Data Mining Conference-Volume 121 (pp. 171-182). Australian Computer Society, Inc..

[5] Hu, X. (2011). Large-scale malware analysis, detection, and signature generation (Doctoral dissertation, University of Michigan).

[6] Alazab, M., Venkataraman, S., & Watters, P. (2010, July). Towards understanding malware behaviour by the extraction of API calls. In Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second (pp. 52-59). IEEE.

[7] Sathyanarayan, V. S., Kohli, P., & Bruhadeshwar, B. (2008, July). Signature generation and detection of malware families. In Australasian Conference on Information Security and Privacy (pp. 336-349). Springer, Berlin, Heidelberg.

[8] Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., & Hamze, A. (2010, March). Malware detection based on mining API calls. In Proceedings of the 2010 ACM symposium on

applied computing (pp. 1020-1025). ACM.
[9] Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., & Bringas, P. G. (2010, February). Idea: Opcode-sequence-based malware detection. In International Symposium on Engineering Secure Software and Systems (pp. 35-43). Springer, Berlin, Heidelberg.
[10] Wagener, G., & Dulaunoy, A. (2008). Malware behaviour analysis. Journal in computer virology, 4(4), 279-287.