An Automatic Approach to Detect Anti-debugging in Malware Analysis

Peidai Xie, Xicheng Lu, Yongjun Wang, Jinshu Su, and Meijian Li

[1] School of Computer, National University of Defense Technology, Changsha, China
peidaixie@gmail.com, xclluu@163.com, wwyyjj1971@126.com

Abstract. Anti-debugging techniques are broadly used by malware authors to prevent security researchers from reversing engineering their created malware samples. However, the countermeasures to identify anti-debugging code patterns are insufficient, and mainly manual, which is an expensive, time-consuming, and error-prone process. There are no automatic approaches which can be used to detect anti-debugging code patterns in malware samples effectively. In this paper, we present an approach, based on instruction traces derived from dynamic malware analysis and an instruction-based pattern matching method, to detect anti-debugging tricks automatically. We evaluate this approach with a large number of malware samples collected in the wild. The experience shows that our proposed approach is effective and about 40% of malware samples in our experimental data set has been embedded anti-debugging code.

Keywords: malware analysis, anti-debugging, instruction trace, obfuscation, dynamic analysis.

1 Introduction

Malicious software (malware) is a generic term to denote all kinds of unwanted software that fulfills the deliberately harmful intent of attackers. Terms such as viruses, worms, Trojan horses, spywares or bots are used to describe malware samples that exhibit some specific malicious behavior[1]. Attackers create malware in order to infiltrate computer systems, collect users' private-sensitive information and attack internet infrastructures. To gain financial benefits is their ultimate goal, and the financial loss caused by malware can be billions of dollars in a year[2]. Nowadays, the sheer number of unique malware samples grows exponentially every year[2] and poses a major security threat to internet.

To detect and mitigate malware effectively, the first step is the dissection of the target, means to extract malicious behavior patterns of malware accurately, a process of malware analysis[3]. There are two major approaches which have been established: *static analysis* and *dynamic analysis*. Static analysis techniques, when used for malware, have several weaknesses, such as it is time-consuming, reliant on disassembling heavily and vulnerable to code obfuscations, and the code being analyzed is possible not the code executed actually, etc.

Y. Yuan, X. Wu, and Y. Lu (Eds.): ISCTCS 2012, CCIS 320, pp. 436–442, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Dynamic malware analysis techniques are proposed to gain a briefly understanding of malware sample, and at the same time, the syntactic signatures are extracted for malware detection. Several specific debuggers, such as WinDBG[4], OllyDBG[5], etc., are playing a significant role as restricted environments for dynamic malware analysis.

However, malware authors use anti-analysis techniques broadly to impede reversing engineering of their creations in order to evade analysis and detection. If a malware sample is aware of an unreal environment in which it is running, it will quit or suspend running to avert exposure of its malicious behavior. In this paper, we focus on anti-debugging techniques. Lots of tricks can be played by a malware sample to detect if the running is in debug-mode. When a huge number of malware samples use anti-debugging techniques, the effectiveness of impeding malware analysis cannot be undervalued.

The existing countermeasures proposed in literatures are insufficient. For example, some plugins for OllyDBG are developed in [5] to tackle the problem of debugger detection, but they can only detect whether a Windows API `IsDebuggerPresent()` is invoked. A stealthy debugger based on Virtual Machine environment or a hardware-level emulator such as QEMU[6] can avoid debugger detection, but the dynamic analysis tool aims at the effects performed by the sample under analysis on operation system resources (e.g., which files or register hives are created or modified). There is not a general method for detecting the anti-debugging code fragments automatically and efficiently.

In this paper, we present an automatic approach, named as ITPM, to detect anti-debugging in malware samples. This approach is based on *Instruction Trace* derived from a dynamic analysis tool and *Pattern Matching* algorithm between the instruction trace and predefined rules which is configured into a database to describe the anti-debugging code patterns. We implement a prototype to demonstrate its effectiveness and the experiment shows that more than 40% of malware samples use anti-debugging techniques even though the packers are broadly used.

This paper makes the following contributions:

- ITPM, an approach based on instruction trace and pattern matching technique, is proposed for identifying automatically anti-debugging in malware samples. The instruction trace is derived from a dynamic analysis tool implemented by ourselves and the patterns of anti-debugging code are in the form of predefined rules configured into a database. ITPM is scalable for new form of anti-debugging patterns.
- A prototype system is designed and implemented to demonstrate ITPM's effectiveness. The dynamic analysis tool used for generation of instruction trace is built on top of QEMU.

The remainder of this paper is structured as follows. In section 2, we introduce related work of detection methods of anti-debugging in malware analysis. In section 3, we describe detailed ITPM, including rule generation, the instruction tracer, the trace refiner, and an instruction-based matching algorithm. Section 4 evaluates our detection approach. Finally, section 5 concludes this paper.

2 Related Work

In this section, we briefly explain previous studies related to detection methods of anti-debugging in malware analysis. A malware analyst usually removes anti-debugging code manually depending on the reverse engineering experience during analyzing a malware sample in a debugger. It is time-consuming and error-prone, and special skills of malware analysis in a certain level are required[7].

Kawakoya[8] implemented a stealthy debugger for automatically unpacking. A stealthy debugger is a debugger which uses original debugging functionalities embedded in a virtual machine monitor in order to hide from the malware running on a guest OS. This method is effective but cannot known what anti-debugging techniques are used in malware samples.

3 The ITPM Approach

In this section, we describe our proposed approach, ITPM, in detail. The work flow of ITPM approach is shown in Fig. 1. The rules of anti-debugging code patterns are generated from corresponding code fragments identified by experts. Instruction traces are recorded from a dynamic analysis tool and be deobfuscated to match with the predefined rules. One instruction trace should be matched with all rules.

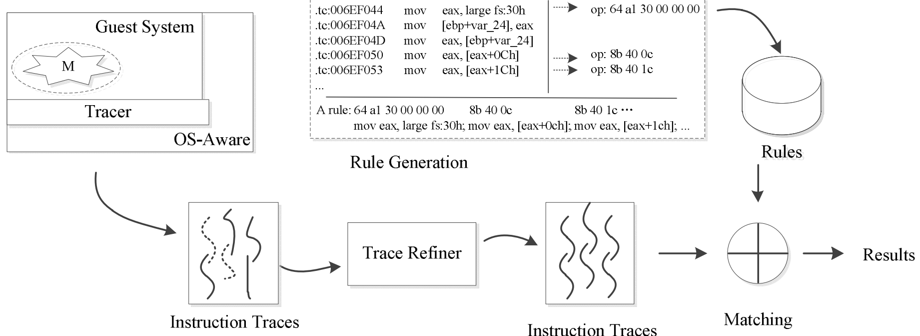

Fig. 1. The work flow of ITPM approach

3.1 Rule Generation

In order to detect the anti-debugging in malware samples, patterns of anti-debugging code fragment should be known as a prior knowledge. All rules are generated from well-known code fragments implemented for anti-debugging by malware authors.

A rule used in ITPM is defined as <*id*, *I*, *D*, *desc*>. The *id* is an identification of a rule; the *I* is an instruction sequence i_1; i_2; i_3…, a set of instructions; the *D* is the binary data corresponding instructions, printed with hex value if needed; the last *desc* is a description of the rule.

When generating a rule with a code pattern of anti-debugging techniques, it is necessary to get rid of uncorrelated instructions imbedded in the code fragment. Uncorrelated instructions include obfuscation code, some redundant branch instructions and so on. It is very difficult to refine a pattern of anti-debugging code fragment as more or less instructions in a rule are all not expected for pattern matching.

As anti-debugging code pattern is varied, the form of rules can be upgrade to deal with corresponding scenarios.

3.2 Instruction Tracer

The tracer implemented in host OS is responsible for recording the instructions executed in the process of a malware sample. It is a dynamic analysis tool for malware analysis built on top of QEMU, an open source CPU emulator, that there are several features due to the hardware-level implementation which are exceeding appropriate for malware analysis. Certainly, there is a semantic gap between guest OS and host OS. In instruction tracer, we bridge the semantic gap by installing a kernel module in the guest OS, a very common solution.

Given the OS semantic information, the tracer reads 15 bytes to a buffer from the EIP in memory when CPU executes an instruction which belongs to a thread whose process is under monitoring. A third party open source library is used to disassemble data in the buffer. And then the EIP, the binary data according to length of current instruction and the instruction denotation are logged as a single line. The instruction trace only includes instructions which belong to the executable under analysis. If the running jumps to a DLL module, all instructions will be omitted.

A timeout interval is set for each analysis. The process which is still running when the interval elapsed will be killed. A clean snapshot of guest OS is loaded for next analysis.

3.3 The Trace Refiner

Code obfuscation techniques are heavily applied in malware samples for evading analysis and detection. If a sample under analysis is obfuscated with syntactic transformation, the trace is too rough to match rules which ought to be matched[9].

On the other side, a trace may include a large number of loops unfolded during actually execution. Those instructions are redundant and should be removed.

The trace refiner is responsible for trimming instruction traces by deobfuscation[10] and loop identification algorithm[11]. Two obfuscations, i.e. code reordering and junk code insertion[12], are detected and replaced by NOP instruction. Loops are pruned to one loop iteration.

3.4 Instruction-Based Matching Algorithm

The last step of ITPM is matching the trimmed instruction traces with rules. We present an instruction-based matching algorithm, as is show in Table 1.

Table 1. Instruction-based Matching Algorithm

Algorithm 1: Instruction-based matching.
Input: The rule set R and a trimmed instruction trace T.
Output: A set O, the element of which is $<r, p>$, r is a matched rule and p is value of EIP.

```
Begin
  while R is not empty do
    Let r is one element of R
    R ← R\r
    j←0, k←0
    Foreach i_j ∈ I_r and t_k ∈ T do
      if t_k is NOP then
        k←k+1
        continue
      if i_j == t_k then
        if i_j is the last instruction of I_r then
          O ← O ∪ <r, EIP_T>
          break
        else j←j+1
      else j←0
      k←k+1
  return O
End
```

In the instruction-based matching algorithm, the instruction in a rule and an instruction trace is compared one by one. We do not use the corresponding bytes. Although the comparison between instructions is more cost than bytes, the number of comparison operation is reduced by a great amount. Instruction-based matching algorithm is effective according to the experiment shown in next section.

4 Experiment

In this section, we present the results of the experiment to demonstrate that the ITPM approach to detect anti-debugging is effectiveness. We conducted the experiment as follows. First, we generate a set of rules according prior studies and our experience, as is shown in Table 2. To demonstrate its effectiveness, we develop a set of experimental tiny programs of which each one has a form of anti-debugging code pattern which is corresponding a rule. After compiled into binaries and obfuscated using well-known packers, we evaluate the ITPM approach. The result shows that all anti-debugging code patterns are detected.

Second, a set of malware samples are collected from Internet, as is shown in Table 3, and marked using Kaspersky, an excellent commercial anti-virus product. We run each sample in the dynamic analysis tool to record an instruction trace with 5 minutes of a timeout interval. And then we trim the set of traces. The length of several traces after trimmed is too short to be discarded.

Table 2. The rule set for anti-debugging detection

Categories	The Mechanism	#Rules	Average of #Instructions
C1	Windows API	12	9
C2	Flags in windows data structure	6	6
C3	Magic strings of debuggers	5	7
C4	Others	2	7

Third, the instruction-based matching algorithm is performed to generate the results. In Table 3, the row #R (Repeated) denotes that some samples have more than one category of anti-debugging code pattern.

Table 3. The malware samples and detection results of ITPM approach

Categories	#	#Trace	Results						
			C1	C2	C3	C4	#Total	#R	%
Bot	331	328	56	41	35	13	131	14	39.6
Worm	296	290	61	39	23	7	118	22	39.9
Trojan	121	117	34	*0*	22	5	43	18	35.5
Unknown	20	20	3	2	2	0	7	0	35.0
sum	768	755	154	82	82	25	299	54	**38.9**

Table 3 shows that about 40% of malware samples have the ability of anti-debugging. We conclude that Trojan is less to use C2 anti-debugging tricks.

5 Conclusion

Anti-debugging techniques are broadly used by malware authors to prevent security researchers from reversing engineering their creations, means malware. In this paper, we present ITPM, an automatic approach to detect anti-analysis tricks in order to make an automated process of malware analysis. The experiment shows that ITPM is effective and about 40% of malware in the wild have anti-debugging function.

Acknowledgment. This work was partially supported by the National Natural Science Foundation of China under Grant No. 61003303 and No. 60873215, Hunan Provincial Natural Science Foundation of China No.s2010J5050 and 11jj7003, the PCSIRT (NO.IRT1012), and the Aid Program for Science and Technology Innovative Research Team in Higher Educational Institutions of Hunan Province “network technology”.

References

1. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A Survey on Automated Dynamic Malware Analysis Techniques and Tools. J. ACM Computing Surveys, 1–49 (2010)
2. Internet Security Threat Report, vol. 16. Symantec Corporation (January 2012), `http://www.symantec.com/business/threatreport/`
3. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: IEEE Symposium on Security and Privacy, Oakland, pp. 231–245 (2007)
4. Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using DJ graphs (1995)
5. Yuschuk, O.: OllyDbg
6. Bellard, F.: Qemu: A Fast and Portable Dynamic Translator. In: The USENIX Annual Technical Conference (2005)

7. Chen, X., Andersen, J., Mao, Z., Bailey, M., Nazario, J.: Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN 2008), pp. 177–186 (2008)
8. Kawakoya, Y., Iwamura, M., Itoh, M.: Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment. In: Proceeding of the 5th International Conference on Malicious and Unwanted Software (2010)
9. Santos, I., Ugarte-Pedrero, X., Sanz, B.: Collective Classification for Packed Executable Identification. In: Proceedings of the 8th Annual Collaboration, Electronic Messaging, AntiAbuse and Spam Conference (CEAS 2011), pp. 231–238 (2011)
10. Yoann Guillot, A.G.: Automatic Binary Deobfuscation (2009)
11. Wei, T., Mao, J., Zou, W., Chen, Y.: A New Algorithm for Identifying Loops in Decompilation. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 170–183. Springer, Heidelberg (2007)
12. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware Normalization. Tech. Report, No.1539, University of Wisconsin, Madison, Wisconsin, USA (2005)