TELKOMNIKA, Vol.10, No.7, November 2012, pp. 1765~1770
e-ISSN: 2087-278X
accredited by DGHE (DIKTI), Decree No: 51/Dikti/Kep/2010

An Approach to Detect Malicious Behaviors by Evading Stalling Code

Chao You*, Jianmin Pang, Yichi Zhang, Chao Dai,Xiaonan Liu
National Digital Switching System Engineering & Technology Research Center, Zhengzhou, China
*corresponding author, email: ycxxgc@163.com

Abstract

Since malwares contain stalling codes, malicious behaviors can't be detected in emulated analysis environment. This paper proposes an approach to detect malicious behaviors by evade stalling codes. First, we executed a malware in the emulated analysis environment, and saved every executed instruction in a trace file; Second, we began to detect stalling codes with the trace file, and constructed stalling code evasive points; At last, we executed the malware again and evade stalling codes with the evasive points, and then the malicious behaviors detected. It has been proven by experiments that the approach can evade stalling codes to detect the later malware behaviors effectively, and improve the performance of detecting the malicious behaviors in the emulated analysis environment.

Keywords: *stalling code; emulated analysis environment; malware; malicious behaviors; evasive point*

Copyright © 2012 Universitas Ahmad Dahlan. All rights reserved.

1. Introduction

Now, malwares such as viruses, worms and Trojan Horses threaten the computer security. Every malware exhibits malicious behaviors. In order to detect malicious behaviors, analysts use static analysis or dynamic analysis method to analyze malwares. Analyzing a malware without executing it is called static analysis. Analyzing the actions performed by a malware while it is being executed is called dynamic analysis. Malware analyzers often use a virtual or emulated analysis environment (such as ANUBIS [1], TEMU [2]) to analyze the malicious behaviors in a malware.

As analysis techniques and tools become more and more elaborate, malware authors propose evasion techniques to prevent a malware from being analyzed. This leads to malware sample that try to detect analysis environment and then either terminate or exhibit non-malicious behaviors to evade analysis. Some researchers (PALEARI, Ferrie, et al.) have proposed methods to detect execution in a system emulator, such as QEMU [3- 5]. Some others (Lau and Svajcer, Rutkowska, Zeltser et al.) have introduced techniques to detect execution inside a virtualized environment [6- 10]. Also, some researchers (Raffetseder, KRUEGEL et al.) have introduced methods to detect the differences between the execution of a program in a virtual or emulated environment, and in a real environment [11-14]. These approaches rely on model specific registers, CPU bugs, and differences in timing. moreover, Garfinkel et al. [15] illustrates detection possibilities based on timing discrepancies, logical, and resource [16] . Since some analysis techniques facilitate the trap flag in the EFLAGS register to create fine grained (on the machine instruction level) analysis, malware samples can detect such approaches through reading the EFLAGS register and inspecting the specific bit within [16] .

In order to evading detection, malware authors use many kinds of techniques to hide malicious behaviors of a malware sample. Inserting stalling codes [17] is one way. The stalling code is executed before any malicious code. The purpose of such code is to execute long enough, after that emulated analysis environment gives up the sample before detecting some malicious behaviors.

In this paper, we propose an approach to detect malicious behaviors by evading stalling codes within the allocated time for the analysis of a sample in the emulated analysis environment.

Received July 29, 2012; Revised September 29, 2012; Accepted October 11, 2012

2. Stalling code

As we know, a sample takes a limit time inside a dynamic analysis system. When the allocated time is over, the analysis system will terminate the analysis of the sample. So malware authors can make a program execute a long time by constructing special code. Dynamic analysis system such as ANUBIS [1] monitors every system call and records detailed information of the system call.

Base on the properties of a dynamic analysis system, malware authors can insert stalling codes into a program. A stalling code consists of a sequence of instructions which fulfills two properties. One is that the sequence of instructions executes much slower inside the analysis environment than inside the real environment. The other one is that the time it takes can’t be neglected.

Malware can call system functions many times to slow down inside the analysis environment. Malware authors can also make use of instructions which are particularly slow to emulate, such as MMX, FPU. All of these are called “slow” operations. These “slow” operations are repeated many times to fulfill the two properties. Figure 1 shows an example of a stalling code found in a real malware.

3. An Approach to Detect Malicious Behaviors by Evading Stalling Code

In this section, we introduce our approach to detect malicious behaviors by evading stalling codes in a malware in detail. Figure 2 shows the three steps of our approach: first execution, detect stalling code and construct evasive points, detect malicious behaviors.

```
unsigned int uCount, uTick;
void  Function()
{
        uTick = GetTickCount();
        uTick++;
        uTick = GetTickCount();
        uTick ++;

}

void StallTime()
{
    uCount=0;
    While(uCount<0x7A120)
    {
        Function();
        uCount++;
    }
}
```

Figure 1. A stalling code in a real malware sample

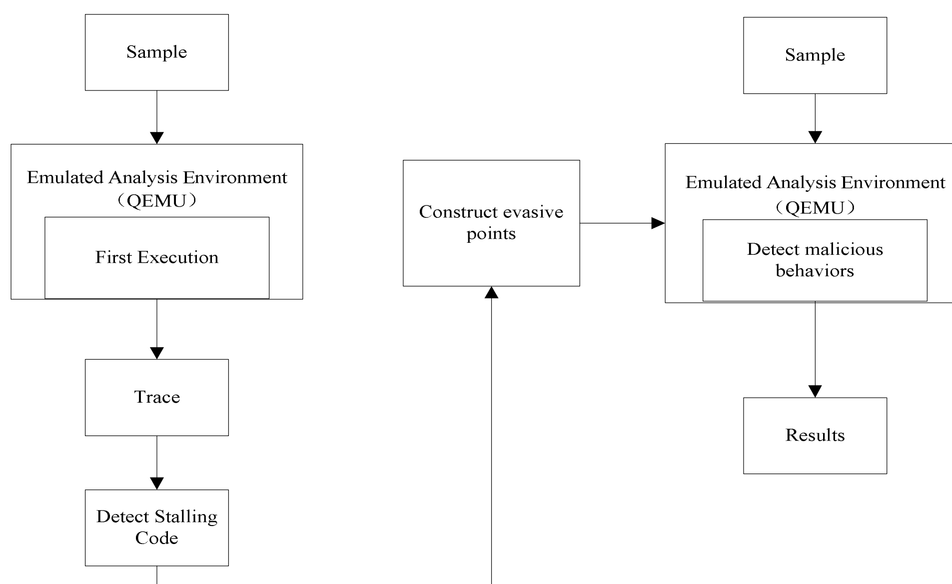

Figure 2. An approach to detect malicious behaviors

3.1 First Execution

As we know, the time a emulated analysis system can spend to execute a sample is limited. How much time is proper? If the allocated time is too long, the number of samples which contain malicious code and evade detection will increase. If the allocated time is too short, the number of the samples analyzed in the system will increase. It can impact the performance of the analysis system.

In order to allocate proper time, We have collected a training set and have tested them in our analysis system. Then, we allow a sample to execute for five minutes. Once the time is over, our system will terminates the sample by force.

We add support for instruction-level traces to our emulated analysis system. When a sample is executing inside the emulated analysis environment, we record every executed instruction. For each instruction, disassembles it, the trace records its address, name and operands. We save all the instructions into a file named “trace”.

3.2 Detect Stalling Code and Construct Evasive Points

The second step of our approach is to detect stalling code and construct evasive points.

1) Detect stalling code

We use the trace file generated in the first step to detect stalling code. A stalling code often consist of a loop and “slow” operations. Figure 3 shows the structure of a stalling code. The first instruction is a comparison instruction. The second one is a branch instruction. The last instruction is a jump instruction which jumps to the address of the first instruction.

```
00401105 cmp    dword ptr [count], 7A120
0040110F jnb    short 00401127
00401111 call   0040100A
00401116 mov    ecx, dword ptr [count]
0040111C add    ecx, 1
0040111F mov    dword ptr [count], ecx
00401125 jmp    short 00401105
00401127 mov    esi, esp
```

Figure 3. The structure of a stalling code

To detect such stalling code in a program, we propose the following work flow, as shown in Figure 4:

a) Is allocated time over? If a sample terminate before allocated time, we consider that this program don’t contain a stalling code, goto d) ; Otherwise，it is likely to contain a stalling code, goto b) .
b) Is trace file searched over? If yes, goto d); otherwise, goto c) .
c) Search trace file. Is there a loop which fulfill the next two conditions? First, the number of the loop is big enough. Second, there are some “slow” operations in the loop. If yes, we consider it is a stalling code, we record the second instruction of the loop, goto b) ; If no, we continue to search the next loop in the trace file, goto b) .
d) Stop detecting. Once the detection is finished, we generate a file named“stalling code list”,which consists of branch instructions of every stalling code.

2) Donstruct evasive points

An evasive point is a new instruction. After executing this instruction, a sample no longer execute the stalling code which contain the original instruction, it jumps out of the stalling code, continues to execute the following code. An original instruction is an instruction in the file “stalling code list” . A new instruction is created by inverting the original instruction. For example, if the original instruction is “jnz” , through inverting it, the new instruction is “jz” .

We take the code in Figure 3 as a example. The instruction at 0040110F is the original instruction. We construct an evasive point for this instruction. By inverting it, the constructed

evasive point is <0040110F jb short 00401127> . After executing the evasive point, the sample directly jump to address 00401127 without repeatedly executing the stalling code.

A sample maybe contains several stalling codes. For each original instruction in file “stalling code list”, we must construct its evasive point.

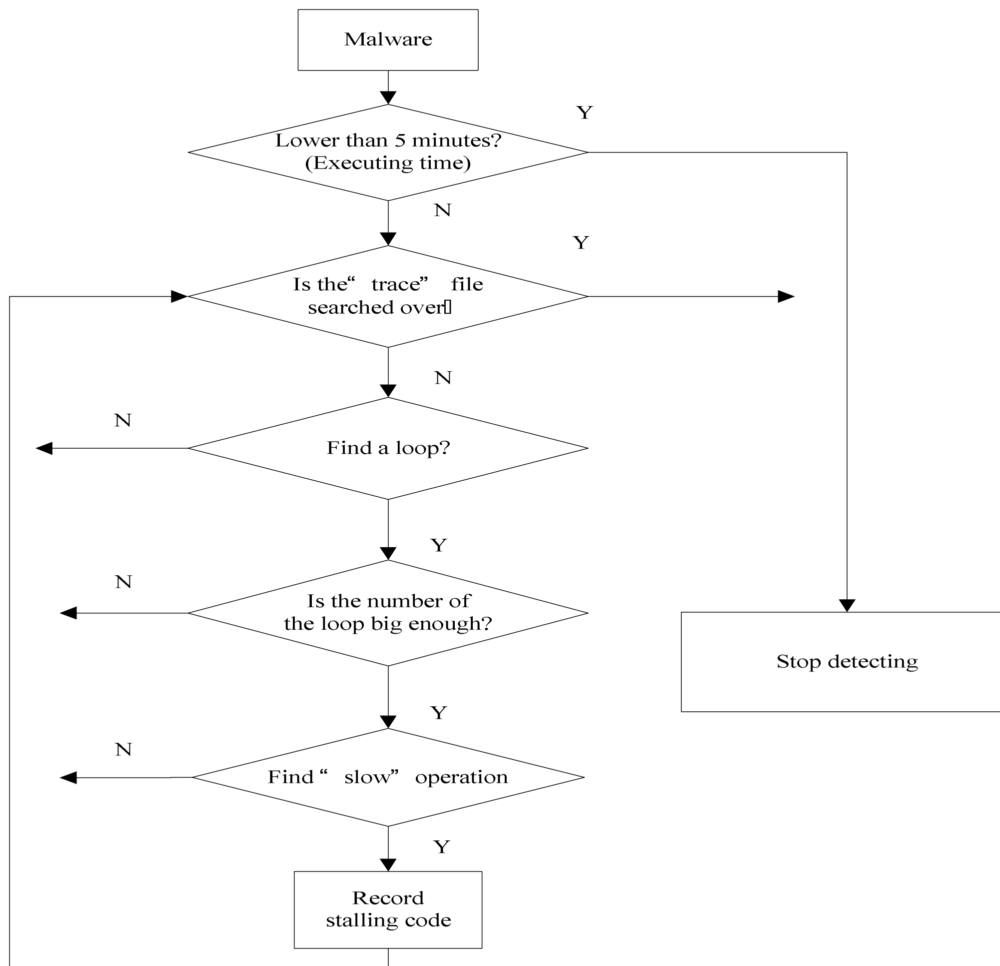

Figure 4. Detect stalling code

3.3 Detect Malicious Behaviors

In this step we analyze and detect malicious behaviors in a sample. First, we must check the “stalling code list”. If it is empty, we give up executing this sample again; otherwise, we need to execute it again.

After loading the sample into the emulated analysis system, we monitor the execution, especially the address of every instruction. While reaching the address of the first evasive point in the “stalling code list” file, we use the evasive point instead of the current instruction, then continue to execute it. We will replace different instruction at different place for several times if there are several evasive points in “stalling code list”.

After this, the emulated analysis system can evade stalling codes, then continue to execute the sample. We can detect the malicious behaviors behind the stalling code in the sample if it has.

4. Results and Analysis

In this section, we evaluate the performance of our approach to detect malicious behaviors by evading stalling codes.

First, we improved QEMU [18] platform, took windows-xp-sp2 as the host operating system. We monitored the execution of a sample, such as every executed instruction, every system call. We randomly selected 400 PE samples from the collected malware program set.

We detected each sample twice between the allocated time(five minutes);once without our approach, and once with our approach. We recorded all detected malicious behaviors. Table 1 and Table 2 show the detailed information.

Table 1. Detected results

Detected Activity	Without Our Approach	With Our Approach	Added Number	Added Percent
File activity	313	377	64	20.45%
Network activity	197	224	27	13.71%
Process activity	326	383	57	17.48 %
Registry activity	308	341	33	10.71%
GUI activity	255	286	31	12.16%

Table 2. Results of executed samples

Description	Total Samples	Finished Samples	Non-finished Samples	α
Without Our Approach	400	313	87	78.25%
With Our Approach	400	348	52	87.00%
Added Number	0	35	-35	8.75%

From Table 1 we can see that the number of every kind of activity increases after using our approach in the emulated analysis system, the added percent of each activity is increases by 10.71% at least. The experiment proves that the approach to detect malicious behaviors by evading stalling codes in the emulated analysis environment is effective.

We define α as analysis efficiency. It is the percent of the finished samples within allocated time, α=(finished samples/total samples)*100%. Finished sample points to a sample which terminates normally within allocate time. Non-finished sample points to a sample which terminates non-normally. The reasons why many samples terminate non-normally are that :1) analysis system terminates them by force because of time out; 2) a sample uses some anti-emulation techniques; 3) when used an evasive point, a sample may terminates non-normally.

From Table 2 we can see that analysis efficiency increases. It increases by 8.75% after using our approach. The experiment proves that our approach can improves the analysis efficiency while analyzing programs inside the emulated analysis system.

5. Conclusion

Anti-emulation technique in malware is widely used to resist dynamic analysis. Instering stalling codes is another way. It delays the execution of the later malicious code long enough, then we can't detect any behavior in the emulate analysis environment. We have proposed an approach to detect malicious behaviors in a program by evading stalling codes. Our experimental result shows that our approach is effective. While analyzing a sample in dynamic emulate environment, we can detect more additional malicious behaviors. These behaviors are useful for a analyst to know the actions of a malware. But the approach is not perfect enough. In the future, we plan to improve our approach to detect and evade more kinds of stalling codes in malwares while analyzing a sample in a virtual or emulated environment [19] .

References

[1] Anubis. Analysis of unknown binaries. http://anubis.iseclab.org. Last accessed, June 2012.
[2] BitBlaze. http://bitblaze.cs.berkeley.edu/release/index.html. Last accessed, June 2012.
[3] PALEARI, R., MARTIGNONI, L., ROGLIA, G. F., et al. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In usenix-woot, 2009.

[4] Martignoni, L., Paleari, R., Roglia, G. F., et al. *Testing CPU Emulators*. In International Symposium on Software Testing and Analysis (ISSTA) , 2009.
[5] Raffetseder, T., Kruegel, C., And Kirda, E. *Detecting System Emulators*. In Proceedings of the Information Security Conference, 2007.
[6] Lau, B. and Svajcer,V. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology*, 2008.
[7] Skoudis, E. and Zeltser, L. Malware: Fighting Malicious Code. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
[8] Garfinkel,T. and Rosenblum, M. *A virtual machine introspection based architecture for intrusion detection*. In 10th Annual Network and Distributed System Security Symposium (NDSS) , 2003.
[9] Ferrie, P. *Attacks on Virtual Machines*. In Proceedings of the Association of Anti-Virus Asia Researchers Conference, 2007.
[10] Rutkowska, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.invisible-things.org/papers/redpill.html, 2004.
[11] Raffetseder,T., Kr¨ugel, C., and Kirda, E. *Detecting system emulators*. In 10th International Conference on Information Security (ISC). 2007: 1~18.
[12] Kang, M., Yin, H., Hanna, S., et al. *Emulating Emulation-Resistant Malware*. In Workshop on Virtual Machine Security (VMSec) , 2010.
[13] Balzarotti, D., Cova, M., Karlberger, et al. *Efficient Detection of Split Personalities in Malware*. In Network and Distributed System Security Symposium (NDSS) , 2010.
[14] Jochen Hirth, Syed Atif Mehdi. Development of a Simulated Environment for Human-Robot Interaction. *TELKOMNIKA Indonesian Journal of Electrical Engineering*. 2011; 9(3): 465-472.
[15] Garfinkel,T., Adams, K., Warfield, A., and Franklin, J. *Compatibility is Not Transparency: VMM Detection Myths and Realities*. In Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI), 2007.
[16] Manuel Egele, Theodoor Scholte, et al. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys*. 2011; 5(1): 1-49.
[17] Clemens Kolbitsch, Engin Kirda, Christopher Kruegel. *The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code*. CCS'11, Chicago, Illinois, USA, October 17–21, 2011.
[18] F. Bellard. Qemu:A Fast and Portable Dynamic Translator. In Usenix annual technical conference, 2005.
[19] Sandip Chanda, Abhinandan De. Congestion Relief of Contingent Power Network with Evolutionary Optimization Algorithm. *TELKOMNIKA Indonesian Journal of Electrical Engineering*. 2012; 10(1): 1~8.