A Slicing-based Approach to Anti-anti-emulation in Malware Analysis

Peidai Xie[1], Xicheng Lu[1], Yongjun Wang[1], Jinshu Su[1]

[1] School of Computer, National University of Defense Technology, Changsha Hunan, China

peidaixie@gmail.com, xclluu@163.com, wwyyjj1971@126.com

Abstract. Anti-emulation check is nearly essential component in modern malware for evading dynamic analysis by malicious behavior hidden in order to be a long time alive. In this paper we propose a slicing-based approach to deal with such a scenario. With a difference from trace matching solutions presented in references, our approach is performed on one instruction trace without a reference platform. We evaluate our approach with 189 malware samples collected in the wild. The experience shows that our proposed approach can spot efAPI used for anti-emulation check in an efficient way.

Keywords: Anti-emulation, Malware Analysis, Slicing, Instruction Trace.

1 Introduction

Dynamic malware analysis techniques[1] are main approaches to gain malicious behavior, while it suffers from limitations, such as incomplete analysis results due to anti-emulation check, that is code embedded intentionally in malware for detection of runtime environment means anti-emulation. If a malware judges the running environment to be a virtual environment, it will quit simply.

The existing references on anti-anti-emulation focus on how to detect anti-emulation actions in malware. The [2] and [3] present an instruction trace matching approach which aligns two traces collected from a virtual platform and a *reference platform* to find a divergence point. It is effective for instruction-level anti-emulation check, but the reference platform is impractical.

In **Fig. 1**, we present an actual sample of such efAPI based anti-emulation.

```
bool isInVirEnv(){
    char *blacklist[] = {"sandbox", "honey",
     "vmware", "nepenthes"};
    char user[128];
    DWORD size = 128;
    int i;
    GetUserName(user, &size);
    for(i = 0; i < 4; i++) {
        if(strstr(user,blacklist[i]))
            return true;
    }
    return false;
}
int main(int argc, char *argv[]){
    …
    if(isInVirEnv())
        return 0;
    …
}
```

Fig. 1. The C-style pseudo code of a dbot sample instance

If the *user* is in *blacklist* dbot predefined, the running will quit immediately.

After analyzed a large number of malware samples in our iPanda[4] system, a lightweight approach based on slicing and execution pattern inspection is proposed to detect and contain anti-emulation actions through **environment fingerprint API** (efAPI) in malware. This approach allows us to understand the efAPI based evading dynamic analysis techniques and also build a DSM for bypass such program branches in order to gain more malicious behaviors[5].

2 Slicing-based Anti-anti-emulation Approach

Inspired by program slicing which find all statements that really affected a variable occurrence[6], we present the slicing-based approach, shown in **Fig. 2**.

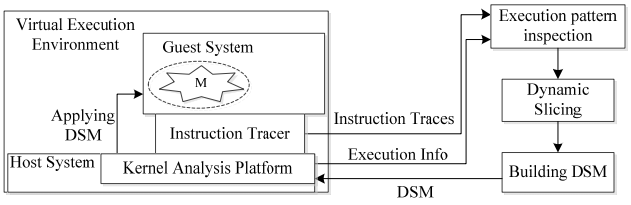

Fig. 2. The workflow of the slicing-based anti-anti-emulation approach

2.1 The Instruction Tracer and Execution Pattern Inspection

The instruction tracer is responsible for recording the instructions executed in the process of an executable for fine-grained malware analysis.

The instruction trace is too rough to be used for slicing. Execution pattern inspection is responsible for trimming the trace, identifying windows API, building dynamic control flow graph (d-CFG) and deciding if a slicing will be performed or not. The algorithm is shown in **Table 1**.

Table 1. Execution Pattern Inspection Algorithm

Stage 1: trimming the trace	**Stage 3:** if it will be slicing
$Sub_T \leftarrow \Phi$	$num_F = 0$, $size = 0$
foreach *I* in *T* **do**	**foreach** *I* in *d-CFG* **do**
if *I* is CALL **and** $Target_I$ is $F \in M_{dll}$ **then**	$size$ += **Size**(*I*)
$Sub_T = Sub_T \cup \{ I \}$	**if** $I \in \{F \mid F$ is in *d-CFG*$\}$ **then** num_F++
until *I* is RET and $NEXT_I \in M_{exe}$	**endif enddo**
endif enddo	**if** $num_F <= \gamma$ **and** $size/$**Size**$(M_{exe}) <= \delta$ **then**
substitute *F* for Sub_T	*isSlicing*=**true**
repeat the procedure on *T*	**else**
Stage 2: building d-CFG	*isSlicing*=**false**
building a *d-CFG* on *T'* using common CFG algorithm	**end**

2.2 Dynamic Backward Slicing Algorithm

The slice criterion is $<I_c, var>$ that the I_c is memory address of a conditional branch instruction and *var* is its conditional variable. We present algorithm in **Table 2**.

Table 2. Dynamic Backward Slicing Algorithm

$S \leftarrow \Phi$	**if** *I Reference var* **then**	$S \leftarrow I$
Define(*var*) $\leftarrow \Phi$, **Reference**(*var*) $\leftarrow I_c$	**Reference**(var) $\leftarrow I$	**endif**
foreach *I* in **Define**(*var*) **do**	$S \leftarrow I$	**end end end**
foreach *r* when $I \in$ **Reference**(*r*) **do**	**elif** *I Define var* **then**	**return** *S*
foreach *I* from I_c to $I_{call\text{-}self}$ backward **do**	**Define**(var) $\leftarrow I$	

3 Experiment

We conducted the experiment as follows. Firstly, a set of malware samples are executed in iPanda for recording traces, and then execution pattern inspection algorithm is performed for picking proper traces for slicing.

The execution pattern inspection hits 8 in 73 samples for performing slicing (#FSlicing) when γ is 20, δ is %5, and 7 samples are successful sliced, as is shown in **Table 3**. The one slicing failed uses SEH-based anti-emulation check.

Table 3. The Slicing

ID	#FSlicing	#SSlicing	Average Time
S1	5	4	13s
S2	3	3	61s
S3	0	0	-

The efAPI list found by our approach is shown in **Table 4**. The #Q stands for the number of samples quitting at the API, and the #A stands for the number of samples invoked the API.

Table 4. The efAPIs used by malware samples in the experiment

API	#Q	#A	API	#Q	#A
IsDebuggerPresent()	2	7	CreateMutex()	1	1
CheckRemoteDebuggerPresent()	0	1	RegQuery()	2	1
GetComputerName()	0	1	CreateFile()	1	3
GetUserName()	1	2			

Reference

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. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating Emulation-Resistant Malware. In: VMSec, pp. 11-23. Chicago, Illinois, USA (2009)
3. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10). San Diego, USA (2010)
4. Xie, P.D., Lu, X.C., Su, J.S., Wang, Y.J., Li, M.J.: iPanda: A Comprehensive Malware Analysis Tool. In: 27th International Conference on Information Networking (ICOIN'13). Bangkok, Thailand (2013)
5. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting Environment-Sensitive Malware. In: Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11), 2011, pp. 338--357. Menlo Park, California, USA (2011)
6. Agrawal, H., Horgan, J.R.: Dynamic Program Slicing. In: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp. 246-256. (1990)