Reducing the Attack Surface of Dynamic Binary Instrumentation Frameworks

Ailton Santos Filho[1], Ricardo J. Rodríguez[2] and Eduardo L. Feitosa[1]

[1] Instituto de Computação, Universidade Federal do Amazonas (UFAM), Manaus, Brazil
`assf@ufam.edu.br, efeitosa@ufam.edu.br`

[2] Centro Universitario de la Defensa, Academia General Militar, Zaragoza, Spain
`rjrodriguez@unizar.es`

Abstract. Malicious applications pose as one of the most relevant issues in today's technology scenario, being considered the root of many Internet security threats. In part, this owes the ability of malware developers to promptly respond to the emergence of new security solutions by developing artifacts to detect and avoid them. In this work, we present three countermeasures to mitigate recent mechanisms used by malware to detect analysis environments. Among these techniques, this work focuses on those that enable a malware to detect dynamic binary instrumentation frameworks, thus increasing their attack surface. To ensure the effectiveness of the proposed countermeasures, proofs of concept were developed and tested in a controlled environment with a set of anti-instrumentation techniques. Finally, we evaluated the performance impact of using such countermeasures.

Keywords: Anti-instrumentation, Analysis-aware, Malware, Dynamic Binary Instrumentation, Anti-analysis

1 Introduction

The number of software specially crafted with malicious intentions (commonly referred as *malware*) has increased in quantity and complexity during the last years [2]. This fact supposes an issue for the anti-virus companies that need to dispose of an up-to-date database of known malware to protect their customers. Thus, malware analysts face every day to an increasing number of malware samples. For instance, Kaspersky (a well-known anti-virus company) stated that they analyzed more than 360.000 malware samples per day in 2017 [11].

The process of analyzing a piece of software and determining if it is malicious can be done manually or automatically. A manual analysis requires to first analyze the program assembly code in a static way (without executing it) and then to execute it to analyze the program behavior when interacting with the Operating System (internal behavior) and with the network (external behavior). However, this manual approach is a very intensive and time-consuming process (in fact, it is usually referred as an art instead of a science). To cope with the increasing trend of malware samples, automation of malware analysis tasks in isolated analysis environments as sandboxes or hypervisors has emerged in recent years [9]. This automatic approach allows the anti-virus companies to keep updated their databases in a timely manner.

Unfortunately, malware writers have also started to incorporate small pieces of code into their software to detect analysis environments. This is mainly motivated because the longer the malware is undetected, the more revenue the cybercriminals achieve. The kind of malware that behave differently depending on where they are executed is referred as *evasive malware*, *analysis-aware malware*, or *split personality malware* [3, 13, 19, 21, 23]. The incorporation of those evasion techniques allows a malware to check where it is being launched and thus behave benignly when an analysis environment is detected. As a consequence, the malware that is wrongly identified as benign software can easily disseminate and penetrate a target system [5].

Special care has to be taken when analyzing an evasive malware to prevent detection of analysis environment. Some authors proposed to execute them first on a bare-metal system and then compare their behavior when executed on other emulation and virtualization-based analysis systems [12]. Other authors proposed tools such as `PinVMShield` [21] or `Arancino` [19] detect or circumvent the evasion techniques used by malware to hinder their analysis. Both tools are based on dynamic binary instrumentation, which allow the malware analyst to insert arbitrary code during malware execution. Although this kind of analysis is feasible, it can also be detected through different techniques, as widely reported in the literature (we reviewed all these works in Section 2).

In this paper, we evaluate three techniques to demonstrate how a framework for dynamic binary analysis (in particular, Intel Pin) can be detected. Those techniques were previously proposed in the literature, but only from a theoretical perspective. Here, we have implemented them to test their detection efficiency. Those evasion techniques increase the attack surface of a dynamic binary instrumentation framework, since they can be used by malware to detect that analysis environment. Moreover, we also provide the corresponding countermeasures to mitigate those evasion techniques. Those countermeasures have been integrated in `PinVMShield` [21], a plugin-based tool specially designed to circumvent evasive techniques performed by malware. Furthermore, they have also been integrated in the benchmark-like tool *eXait* [7], which serves to verify if a DBI framework is recognizable. Finally, to prove the effectiveness of our countermeasures, we evaluate their efficacy and performance using SPEC CPU 2006 [6]. The results showed that although the evasive behavior can be circumvented, the use of current frameworks of dynamic binary analysis has a great overhead, as already claimed by other authors [20].

This paper is organized as follows. Section 2 reviews the literature. Section 3 is devoted to previous concepts needed to follow the rest of this work, as the DBI framework developed by Intel or the architecture of the `PinVMShield` tool. Section 4 introduces the new techniques for detecting those analysis environments, as well as the proper countermeasures. Finally, Section 5 concludes the work and states future work.

2 Related Work

This section presents in chronological order the works related to techniques for detecting a DBI framework, detailing also the countermeasures proposed in each work. We first review the related works in the industry and then in the academy.

Most of the existing studies come from industry-related security conferences. Falcón & Riva introduced in [7] the first techniques to detect the presence of the Intel Pin DBI framework in a Windows OS. In particular, thirteen techniques with the corresponding proofs of concepts (PoCs) were presented. Those PoCs were distributed as a sort of a benchmark-like tool called *eXait*. This tool is useful to verify if a DBI framework is recognizable.

Two years later, Li & Li [14] showed that the DynamoRIO DBI framework can be detected in both Windows and Linux environments through ten new evasion techniques. Furthermore, they remarked the transparency problem of DBI frameworks in some cases. Similarly, Hron & Jermář proposed in [10] six new evasive techniques against the Pin and DynamoRIO, providing also PoCs of those techniques. Sun et al. also presented in [22] other six new evasive techniques aimed at the DBI Pin and DynamoRIO frameworks. In addition, they introduced the idea of escaping of the sandbox-like environment provided by a DBI framework, thus compromising the analysis environment.

Regarding the academic literature, Rodríguez et al. [21] reviewed and provided a taxonomy of the existing anti-DBI and countermeasures techniques up to that time. Moreover, they released an extensible Pin-based tool called `PinVMShield` that enabled to analyze applications with those evasive techniques by means of circumventing them. Later, Polino et al. [19] introduced a review and classification of some anti-DBI techniques manly focused on the Pin DBI framework. Furthermore, they developed a set of approaches to mitigate evasion techniques and used the *eXait* tool [7] to validate their effectiveness. They also introduced a Pin-based tool named `Arancino`, which incorporates the proposed countermeasures and allows to retrieve the original binary form of malware programs protected by software packers.

Recently, Zhechev published a Master's thesis [24] raising the question whether DBI frameworks are appropriate tools for analyzing malware and other potentially evasive artifacts. Moreover, he introduced thirteen new techniques to detect the presence of Intel Pin DBI framework in a Linux OS, and also demonstrated that escaping of such a DBI framework is feasible due to the shared memory model used by DBI frameworks. In this regard, Zhechev stated that the isolation and the stealthiness of the analysis code under DBI frameworks are not guaranteed and thus, DBI frameworks are unsuitable for building any security-related application.

As shown, DBI frameworks are gaining popularity among security researchers as a way to insert arbitrary code during program execution. Following this trend, in this paper we provide PoCs of three evasion techniques already documented in the literature (but without PoCs) and we give the proper countermeasures.

3 Previous Concepts

3.1 Intel Pin DBI Framework

The Pin DBI framework (or Pin for short) enables to build easy-to-use, portable, transparent, and efficient dynamic instrumentation tools. Pin was designed by Intel in 2005 and gives support for the three major desktop Operating Systems (i.e., Windows, Linux, MacOS X).

Pin is composed of the three typical DBI components (depicted in Figure 1): (1) the application to be instrumented; (2) a dynamic binary analysis tool developed with Pin, normally termed as *Pintool*; and (3) the DBI engine. The DBI engine consists of a Virtual Machine (VM), a code cache, and an instrumentation application programming interface (API) invoked by the Pintool. The VM takes as input the native executable code of the application to be instrumented and uses a just-in-time (JIT) compiler to insert the instrumentation code, prior to execution. Then, the resulting instrumented code is saved in the code cache and the execution is transferred to it. After execution, the JIT compiler fetches the next sequence of instructions to be executed and generates more code. The emulator unit is in charge of instructions that cannot be directly executed, such as system calls that require special handling from the VM [16]. As shown in Figure 1, the instrumented application, the Pintool, and the DBI engine are executed in the same memory address space.

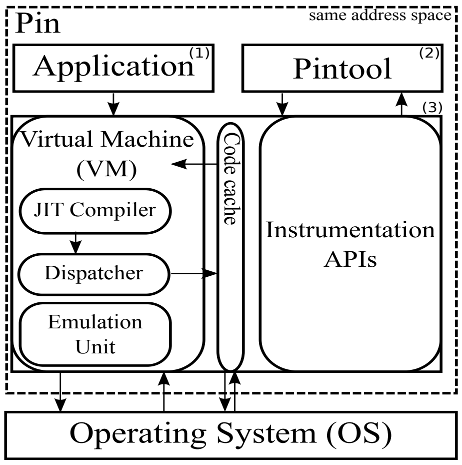

Fig. 1. Pin's software architecture (adapted from [16]).

Pin has been widely used in the scientific community. As a result, a lot of interesting Pintools have been released. In this regard, it is worth mentioning Pintools that enable debugging for instrumented applications, such as PinADX [15] or the tool introduced in [18].

3.2 PinVMShield

`PinVMShield` [21] is a tool to detect and circumvent evasion techniques used by analysis-aware malware. The tool follows a plug-in architecture, enabling an easy extension for covering other evasion techniques than the currently supported ones. It was released under GNU GPL version 3 license and its source code is publicly available at `https://bitbucket.org/rjrodriguez/pinvmshield/`.

`PinVMShield` is mainly focused on Windows OS, and uses two granularity instrumentation levels, at the routine level and at the instruction level. This allows the tool to detect evasive behavior based on Windows APIs, such as checking the presence of a software debugger though `IsDebuggerPresent` or `CheckRemoteDebugger`, or based on specific assembly code instructions (e.g., sidt, sgdt, or sldt [8]).

`PinVMShield` currently addresses the evasive behavior performed by software binaries to detect virtual environments (in particular, VirtualPC, VMWare, and Virtualbox), debuggers (WinDBG, OllyDBG, and ImmunityDebugger), and sandboxes (WinJail, Cuckoo Sandbox, Norman, Sandboxie, CWSandbox, Joe-Sandbox, and Anubis).

4 Evasive Techniques and Countermeasures

In this section, we address some of the evasion techniques that a malware can incorporate into its code to detect when it is being executed inside a DBI framework. Furthermore, to the best of our knowledge we are the first to provide countermeasures for those evasion techniques, which are introduced next.

4.1 Evasive Techniques

An exhaustive list of techniques to detect DBI frameworks is discussed in [22,24]. There, small pieces of code are given as PoCs, as well as proper countermeasures, specially designed for the Intel Pin DBI framework and Linux OS running on top of Intel x86-64 architectures.

Among those evasive techniques for which no countermeasures are provided, there are few of them particularly relevant, since they may be used as an attack vector by malware against the analysis environment. In particular, those techniques are the neglecting of the No-eXecute bit, TLS detection, and code cache signatures detection. In the rest of this section, we describe each of those evasion techniques in detail. Furthermore, we also provide PoCs and countermeasures. Unlike [24], we focus on the Windows OS, as it is most prevalent system attacked by evasive malware [21].

To demonstrate the effectiveness of these evasion techniques against Pin, our PoCs have been integrated as plugins for *eXait* [7]. Those plugins are publicly released under GNU GPLv3 and freely available online[3] Moreover, our proposed countermeasures have been integrated in `PinVMShield` [21].

Neglecting No-eXecute Bit. The No-eXecute (NX) bit is a defense mechanism added at hardware level to prevent the execution of data memory pages by the processor. Roughly speaking, the main idea is that no memory zone is simultaneously writable and executable. This feature is incorporated by almost all the processor's manufacturers, although referred to under different nomenclature (e.g., *execute disable* bit in Intel, *enhanced virus protection* in AMD, or *execute never* bit in ARM). Recall that a DBI framework uses a JIT compiler that "recompiles" the application code in conjunction with the instrumented code (see Section 3.1). That is, it needs to *write-then-execute* certain memory zones. Therefore, as suggested by Zhechev [24], any program instrumented by a DBI framework in JIT mode has this protection disabled.

Therefore, an application can detect a DBI framework by allocating a new heap space, placing valid code on it, and then executing it. When the application is not being instrumented, then the execution crashes since the heap memory page has no permission to execute. However, the execution under a DBI framework will continue normally.

TLS Detection. *Thread Local Storage* (TLS) is a feature that allows a developer to provide unique data for each thread, in vector format, accessed by the process using a global index. Roughly speaking, TLS variables can be seen as global variables only visible to a particular thread and not the whole program. In the case of Windows, those per-thread global variables are maintained in the TLS directory, which is a part of the Portable Executable (PE) header of an executable image. The PE header is the header of any Windows executable file. The minimum number of positions in the directory is guaranteed to be at least 64 for any system, while the maximum number is 1088 [17]. According to [22], DBI frameworks as Pin allocate and use positions of that data structure for internal purposes.

Since allocated indexes are shared by any thread in the process, an application can inspect the number of positions allocated in the TLS directory, revealing the presence of a DBI framework.

Code Cache Signatures Detection. As reported in [19], there are several artifacts that a DBI tool unavoidably leaves in memory. In this regard, the authors in [22] shown that those memory artifacts can be easily detected by any application under analysis. In the case of Pin, one of those memory artifacts is

[3] See `https://github.com/ailton07/eXait_Plugin_PinDetectionByDEPNeglect`, `https://github.com/ailton07/eXait_Plugin_CodeCacheDetectionByFEEDBEAF`, `https://github.com/ailton07/eXait_Plugin_PinDetectionByTLS`

the hexadecimal pattern `0xFEEDBEAF`. This byte sequence is repeated through the memory zones used as code cache by Pin, as an identification fingerprint.

Therefore, an application can detect those memory patterns by means of memory scans. The presence of those byte patterns in the memory allocated by a process can be used as a criterion to indicate the presence of Pin in this particular case.

4.2 Countermeasures

For the development of the proposed countermeasures, we used the tool `PinVMShield` since its plugin-based architecture makes easier for a developer to incorporate new mitigation techniques against an evasion technique. To foster research in this area, the developed countermeasures are also publicly released an available online[4]. In the following, we briefly describe each countermeasure.

Neglecting No-Execute Bit. As countermeasure, we can rely on the set of Windows APIs that allow a developer to check for specific memory addresses and permissions. The developed countermeasure detects when an attempt to execute a piece of code from an invalid region (non-executable memory) occurs, and allows the binary analyst to take action afterwards. A more transparent implementation would raise an exception with status *STATUS_ACCESS_VIOLATION*, imitating the behavior outside the DBI framework.

TLS Detection. Since the TLS access is made exclusively using APIs provided by Windows OS, the proposed countermeasure relies on monitoring those APIs and logically isolating the positions used by Pin. Thus, an attempt to read or modify a position already allocated by the DBI tool is redirected to a position unrelated to it. The `PinVMShield` tool already provides facilities for intercepting and modifying Windows APIs, through the *ReplaceWinAPI* method.

Code Cache Signatures Detection. As suggested in [19], a process can scan its entire memory space performing an inspection of every allocated memory page. In Windows OS, this can be done through APIs as *VirtualQuery* or *NtQueryVirtualMemory*. To mitigate this evasion technique, it is necessary to establish a memory access policy, telling which memory addresses are *protected memory addresses*. This can be done by creating a *blacklist* of memory addresses related to the DBI framework which cannot be accessible by the instrumented application. In this regard, Pin provides a set of APIs, such as *CODECACHE_AddTraceInsertedFunction*, which allows an analyst to monitor the creation of code caches and thus get their addresses, enabling the creation of that blacklist upon execution.

Once the blacklist is initialized, those Windows APIs that allow memory page inspection can be monitored to bypass the queries of any protected memory

[4] See `https://github.com/ailton07/PinVMShield`

regions. Our developed countermeasure intercepts those calls when an attempt to obtain information from a protected memory region is done and returns a zero value, indicating that the function failed.

4.3 Efficacy of the Countermeasures and Performance Impact

These countermeasures were incorporated in `PinVMShield` to evaluate their effectiveness. We developed then small pieces of code implementing evasive techniques and execute them with `PinVMShield` and our countermeasures enabled. As a result, *TLS Detection* and *Code Cache Signature Detection* were mitigated successfully. Similarly, *Neglecting No-Execute Bit* was detected during the execution attempt, allowing the analyst to take appropriate action at run time.

To measure the impact caused by the use of the `PinVMShield` tool in the application performance, we used the benchmark tool SPEC CPU2006 [6] widely used to evaluate performance measurements on DBI tools [1, 4, 16]. While executing SPEC with `PinVMShield`, features not related to the countermeasures presented here were disabled to avoid performance degradation by other means.

As experimentation environment, we used a virtualized environment on top of a KVM processor 8 cores 3.41 GHz, with 16GB of RAM memory. The virtual machine was running a Windows 7 SP1 x64 with Intel Pin 2.14-71313 and Microsoft 32bit C/C++ compiler v16 for SPEC2006 compilation.

Table 1 summarizes the results obtained for each one of benchmark tools of SPEC. The first column lists the names of the benchmarks. The second column tells the execution time (in seconds) of Pin, while the third shows the execution time (in seconds) of Pin with the `PinVMShield` tool. Finally, the fourth column presents the overhead when `PinVMShield` is used. On average, the overhead was 59.73% with a standard deviation of 98.68%.

Benchmark	**Instrumentation Time (s)**	`PinVMShield` **Time (s)**	`PinVMShield` **Overhead (%)**
400.perlbench	484	534	10.3305
401.bzip2	560	562	0.3571
403.gcc	443	1150	159.5936
429.mcf	192	209	8.8541
445.gobmk	521	537	3.0710
456.hmmer	680	718	5.5882
458.sjeng	606	607	0.1650
464.h264ref	1069	4071	280.8231
471.omnetpp	294	819	178.5714
473.astar	339	326	-3.8348
483.xalancbmk	287	326	13.5888

Table 1. Overhead introduced by `PinVMShield` with our countermeasures.

5 Conclusions and Future Directions

Malicious software able to detect an analysis environment is a potential threat, since they can can behave differently to evade the classification as malware when they detect an analysis environment.

In this paper, we have studied three evasion techniques of DBI frameworks. Those techniques are used as attack vectors by malware to detect an analysis environment. We have provided small pieces of code as proof of concepts of those techniques, as well as their countermeasures to reduce the attack surface of DBI frameworks. Those countermeasures were developed on top of the `PinVMShield` tool. Both proof of concepts and countermeasures have been evaluated in a virtualized environment to prove their effectiveness. Our experiments showed a performance overhead close to 60% by the use of developed countermeasures.

Unlike the work in [24], we have shown that DBI frameworks are suitable for security purposes: when equipped with the appropriate tools, the requirements needed in a security analysis context (i.e., stealthiness and isolation) are achieved. However, the performance overhead introduced by the use of DBI frameworks is still an issue. Nonetheless, we argue that it is necessary to keep observing the growth of evasion techniques focused on DBI tools. Despite efforts done to develop DBI evasion techniques countermeasures, there are still evasion techniques that threaten the stealthiness of DBI frameworks and pose a challenge to system security professionals. For instance, the intrinsic problem of overhead detection that basically affects to any dynamic analysis tool.

Acknowledgments

The research of A. Santos Filho and E. L. Feitosa supported in part by the FAPEAM proc. no. 009/2017 and by the Federal University of Amazonas (UFAM). The research of R. J. Rodríguez was supported in part by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (project references T21-17R).

References

1. Arafa, P.: Time-Aware Dynamic Binary Instrumentation. Ph.D. thesis, University of Waterloo (2017)
2. AV-TEST GmbH: The AV-TEST Security Report 2017/2018 (2018)
3. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2010)
4. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. ACM SIGPLAN Notices **47**(7), 133–144 (2012)
5. Carpenter, M., Liston, T., Skoudis, E.: Hiding Virtualization from Attackers and Malware. IEEE Security & Privacy **5**(3), 62–65 (2007)
6. CPU2006, S.: Standard Performance Evaluation Corporation. [Online; `https://www.spec.org/cpu2006/`] (2006)

7. Falcón, F., Riva, N.: Dynamic Binary Instrumentation Frameworks: I know you're there spying on me (2012)
8. Ferrie, P.: Attacks on Virtual Machine Emulators. Symantec Advanced Research Threat Research pp. 1–13 (2007)
9. Greamo, C., Ghosh, A.: Sandboxing and Virtualization: Modern Tools for Combating Malware. IEEE Security & Privacy **9**(2), 79–82 (2011)
10. Hron, M., Jermář, J.: SafeMachine malware needs love, too. [Online; `https://www.virusbulletin.com/uploads/pdf/conference_slides/2014/sponsorAVAST-VB2014.pdf`] (2014)
11. Kaspersky Lab: Kaspersky Lab detects 360,000 new malicious files daily – up 11.5% from 2016. [Online; `https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily`] (2017)
12. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: Bare-metal analysis-based evasive malware detection. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 287–301. USENIX Association, San Diego, CA (2014)
13. Kumar, A.V., Vishnani, K., Kumar, K.V.: Split Personality Malware Detection and Defeating in Popular Virtual Machines. In: Proceedings of the 5th International Conference on Security of Information and Networks (SIN), pp. 20–26. ACM (2012)
14. Li, X., Li, K.: Defeating the transparency features of dynamic binary instrumentation. BlackHat US (2014)
15. Lueck, G., Patil, H., Pereira, C.: PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation. In: Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO), pp. 114–123. ACM, New York, NY, USA (2012)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '05, pp. 190–200. ACM, New York, NY, USA (2005)
17. Microsoft: Thread Local Storage. [Online; `https://msdn.microsoft.com/en-us/library/windows/desktop/ms686749(v=vs.85).aspx`] (2018)
18. Pan, H., Asanović, K., Cohn, R., Luk, C.K.: Controlling Program Execution through Binary Instrumentation. SIGARCH Comput. Archit. News **33**(5), 45–50 (2005)
19. Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontata, L., Gritti, F., Zanero, S.: Measuring and Defeating Anti-Instrumentation-Equipped Malware. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 73–96. Springer International Publishing, Cham (2017)
20. Rodríguez, R.J., Artal, J.A., Merseguer, J.: Performance Evaluation of Dynamic Binary Instrumentation Frameworks. IEEE Latin America Transactions (Revista IEEE America Latina) **12**(8), 1572–1580 (2014)
21. Rodríguez, R.J., Gaston, I.R., Alonso, J.: Towards the Detection of Isolation-Aware Malware. IEEE Latin America Transactions **14**(2), 1024–1036 (2016)
22. Sun, K., Li, X., Ou, Y.: Break Out of The Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. Black Hat Asia (2016)
23. Vishnani, K., Pais, A.R., Mohandas, R.: Detecting & Defeating Split Personality Malware. In: Proocedings of the 5th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), pp. 7–13 (2011)
24. Zhechev, Z.: Security Evaluation of Dynamic Binary Instrumentation Engines. Master's thesis, Department of Informatics Technical University of Munich (2018)