Detection of VM-Aware Malware

David Yu Zhu and Erika Chin, University of California - Berkeley

December 11, 2007

Abstract

VM-aware malware is a class of malicious software that can detect whether it is running in a virtualized environment and alter it is behavior based on this result. As much of today's malware analysis is done on virtual machines, this type of malware can successfully evade observation and inspection. Currently, analysts often compare the behavior of malware in a physical host to its behavior in a virtual machine. Unfortunately, this painstakingly manual procedure does not lead to the identification or subversion of detection mechanisms used in malware.

In this paper, we present two complementary techniques that promote the identification of VM detection behavior in VM-aware malware. The first method uses dynamic analysis to identify known virtual machine detection techniques with high accuracy and low overhead. The second method uses dynamic taint tracking to detect any impact caused by input that changes the execution path of the malware. This offers the additional benefit of revealing novel VM detection methods. We have developed a prototype based on the first approach and conducted experiments using both proof-of-concept VM-aware program samples and unidentified malware collected in the wild. Our results show that our system can identify the VM-detection method and provides insight to the VM detection mechanisms commonly employed by malware.

1 Introduction and Problem motivation

Malware on the Internet poses a serious threat to the economy and user privacy. Malware, including various types of viruses, worms, and botnet binaries, costs the economy over 62 billion dollars annually [12]. Code Red alone cost 2.6 billion dollars worldwide [8]. Moreover, over 17% of computers are infected with malware [7]. Furthermore, exploited vulnerabilities have given malware writers unlimited access to highly sensitive user information. With the increase of highly confidential Internet traffic, it is important to guard against any possible system exploit.

Virtual machines (VMs) have become increasingly important to the security community. Not only do they allow multiple operating systems to run on the same physical host in an isolated and transparent fashion, but they also allow snapshots to be taken of system so that a user can roll back to a specific state. Thus, they are well-designed to analyze malware without risking permanent infection. This design has found a wide range of applications in modern computing infrastructures. Most notably, virtual machines have been used to construct honeypots or honeynets in order to

capture malware samples. Adversarial malware writers have noticed this trend, and in an effort to evade detection and analysis, they have developed several techniques to detect whether their malware is running in a virtualized environment. In most cases, the malware will stop propagation and/or its malicious behavior when it detects the presence of a virtual machine. This renders malware detection and analysis much more difficult.

There is evidence that an increasing percentage of malware is using VM detection logic. "Red Pill" demonstrates the feasibility and ease of such detection strategies and has led to more malware exploiting the SIDT instruction [15, 16]. Others exploit the call of illegal opcodes [6]. Despite this large increase in VM-detecting malware, there has not been extensive academic research in this area. Analysts are forced to identify such VM-aware malware by executing them on physical host, which introduces significant risk. For this reason, it is important to have the capability and tools needed to not only discover these VM-detecting malware, but to identify the techniques they use. We propose a solution to counteract such detection mechanisms by using a two-fold approach. One is designed specifically to identify known detection mechanisms with high accuracy and a low false positive rate, while the other one can be used to uncover novel VM detection methods based on dynamic taint tracking.

In the next section, we discuss some of the known VM detection techniques. In Section 3, we give an overview of our approach to discovering VM-aware malware. Section 4 details the our design and implementation, and Section 5 presents our evaluation. Section 6 discusses related work and Section 7 discusses some limitations and ideas for future work. We conclude in Section 8.

2 Background

Very little is known about the VM detection methods used in the wild. As most malware analysis is carried out in virtual machines, VM-aware malware can easily avoid analysis. We can, however, comment on where VM artifacts may be visible to the process and where the behavior of a VM is different than the behavior of a physical machine.

There are many virtual machine artifacts in the processes, file system, and registry [16]. Often, the registry will contain entries such as "VMware Virtual IDE Hard Drive" and other similar self-named items. Given access to the registry, a process can do a string comparison to "VMware" or other virtualization brands to confirm that it is running in a VM. There are over 300 references to "VMware" in the registry alone [16].

The guest's memory may also contain evidence that it is not in native hardware. One of the simplest and most widely-used method is to check for the address of the interrupt descriptor table (IDT), local descriptor table (LDT), or global descriptor table (GDT). These tables are generally stored in memory at lower addresses on physical machines and in memory at higher addresses on virtual machines. In fact, the exact locations of these tables have been well-documented for multiple Windows, Linux, and VMware versions. This method of identification is so simple that it can been done using one instruction. The most widely-known implementation of this technique is

detailed in Red Pill [15]. Scoopy [11] also checks for the IDT, LDT, and GDT.

Attacks can also exploit hardware differences. Doo [11] checks the hardware for VM markings. Another attack is to check the synchronization between guest and host clocks. Virtualized machines are frequently slower than physical machines, and this makes it easier for programs to detect that it is in a VM.

It is also possible to detect the VM by its unique actions. Jerry [10] does this by looking for unusual processor behavior due to special features of the VM. Also, by looking for VMware uses a special communication channel to pass messages between the guest and host processes. This is another point of attack. This however can be easily avoided by turning communication off. VMware also handles illegal opcodes differently from physical machines. Based on the machine's response, a malicious process can determine whether it is running in a VM. We will ignore these last points of attack, as they can be easily fixed by changing VM options and fixing software bugs.

3 Approach

Traditionally, the anti-virus community has been divided into two camps. There are those in industry who value few false positives and rely on signature-based schemes to accurately identify malicious software. Others have advocated the use of behavior-based methods such as anomaly detection because of its potential to discover previously unknown attacks and vulnerabilities.

We find value in both of these paradigms and apply them to detect attempts to recognize VMs. Our first approach is to look for specific behaviors or indicators that are key factors in the known detection techniques. For instance, SIDT/SLDT/SGDT instructions should only be used by operating systems software [9]. Any use of these instructions in user-level programs should immediately be flagged. Because malware often obfuscate their binaries to increase the difficulty of reverse engineering, the naive solution of scanning the malicious binary for these instructions would not work well in practice. Instead, we run the suspected VM-aware malware in an emulator and monitor its actions at the instruction level to catch instructions generated on the fly. Using dynamic taint analysis, we are able to identify any SIDT/SGDT/SLDT in both the original and generated code. In addition, we consider the context of the instruction so that any SIDT instruction executed by OS code will not lead to false positives.

The previous approach is quite similar to signature-based methods and it has similar benefits and drawbacks. While it is very effective against well-understood detection methods, it cannot detect new methods. We tried to apply similar techniques to counteract VM identification by using registry values. In this case, calls to registry-related API are monitored. The challenge is to correctly differentiate the legitimate usage of the Windows registry from malicious usage that can potentially identify the VM. The list of potentially VM-identifiable information in the registry is quite long and can vary from VM to VM. Therefore, a pure signature-based approach can no longer accurately identify the detection technique. We solve this by incorporating ideas from a behavior-based approach. We taint any return value from the registry and track their propagation

and see if they affect a conditional jump in the execution. This can help us identify potentially VM-specific registry entries that are used to detect virtual machines.

This leads us to a more general solution that takes advantage of the invariant that malware will make a conditional jump based on some external input or a value derived from external input. These input values can appear in registry entries, system calls, network input, file, etc. By tainting all input and propagating these tainted values throughout the execution of the malware, we can effectively identify the key decision points. This behavior-based approach may have many false positives, but it may also catch previously unknown detection mechanisms. We are currently actively pursuing this approach.

4 Design and Implementation

To demonstrate the feasibility of our approach and gain insight into the prevalent mechanisms used in live malware samples, we have developed Malaware, a plugin to the TEMU platform [1]. At the time of this writing, we have completed the SIDT family detection plugin and conducted experiments on the malware samples. We have completed the development of the registry information identification detection portion but have yet to test them on live malware samples. We will continue to develop a more general behavior-based approach and specifically focus on tracing decision points back to original input and using this information to reduce the number of false positives.

It should be noted that TEMU is based on QEMU [4], which is a whole system emulator itself. QEMU can be used both as a system emulator or a virtualizer [4]. When it is used as an emulator, which is what Malaware uses exclusively, VM code never runs directly on hardware like commercial virtual machines such as VMware [2]. Thus, Malaware has control of the virtual machine state at all times and therefore can manipulate the program's view of the system constantly.

4.1 Descriptor Table Instruction Detection Methods

The first feature of the Malaware plugin is its ability to detect the presence of Descriptor Table Instructions while running malware. This is achieved by interposing the QEMU emulation code that handles the emulation for the specific instructions. However, it is important not to log for all uses of the sensitive instructions, but only when these instructions are found in the suspicious binary or in any other code it generates. One of the fundamental challenges of analysis in a virtualized environment is the semantic gap between the hardware level view at the emulator level and the operating system level view. Using techniques mentioned in [5], the plugin learns the current process and its page table base address. From those, it is able to verify if the current instruction register belongs to the code pages of the monitored process. This indicates that the current instruction belongs to the original binary.

Furthermore, dynamic taint tracking is used to find any code that has been generated by the monitored process. Any memory address written by instruction from the monitored process is tainted. Because we conduct our experiment by loading each module to be tested on a pristine

Windows QEMU image, we do not consider the possibility of the malicious process loading additional code from the file system. This effectively defeats code obfuscation techniques, which is used by many of the malware samples according to our analysis.

4.2 Registry-based VM Detection Methods

The second feature of the Malaware plugin is the use of dynamic taint tracking extensively to track the input from the registry. Windows symbols table provides the addresses of WINAPI functions. By tracking the current instruction register, a hooking function is executed when a call to a registry-related WINAPI function is made. Thus, we are able to taint all return values from registry functions. These tainted inputs are propagated according to a set of rules similar to the one mentioned in [5]. To evaluate if a branching decision is made based on tainted input, it is essential to determine whether the condition code used is a result of operations on tainted values. QEMU uses a lazy evaluation technique to compute its condition code [3], and keeps a copy of the last instruction, one of the operand, and the result that could affect the content of the condition register. Taint propagation rules are designed to preserve taint information in these copies, thus making it easy to check if the branching decision is made based on tainted content.

5 Evaluation

5.1 Experiment Setup

To setup our experiment, we first acquired a Dell Dimension 3100 Desktop computer, wiped it clean, and installed Ubuntu 7.04 (April 2007). On this system, we installed two different types of virtual machines: VMware [2] and QEMU [4]. Our choice in these two emulators is motivated by their drastically different emulation methods. VMware is a commercial, (hardware-bound) reduced privilege guest virtual machine emulator. It utilizes the hardware to execute instructions. As commercial software, it is commonly used by security companies and, therefore, is a good choice for recreating the environment of malware analyzers. In contrast, QEMU is an open source, pure software virtual machine emulator. This pure software approach provides more control to the emulation layer and is generally more difficult to detect.

We also created a clean image of Windows to run on our emulators. For malware samples, we collected binaries of known methods of VM detection including Red Pill [15] and Scoopy [11]. These methods were useful for testing our system on benign, confirmed binaries that detect the VM. We also received binaries of unconfirmed malware samples. These binaries are those that Fabian Monrose collected for his work in [14] but was unable to observe malicious behavior in. It was suspected that these binaries were detecting the VM and not executing its malicious code.

We then ran these samples in the VMware and observed the obvious artifacts of execution. These were then classified (See Figure 1). We also ran these malware in QEMU and noted the

malware that had noticeable differences. Not only did we see many cases of the samples behaving differently, but we also saw known malicious behavior. These samples were then run in Malaware.

5.2 Experiment Result

As part of our preliminary tests, we ran the Red Pill and Scoopy samples and our system detected the SIDT in both of them. These samples were able to detect the VM in VMware but not in QEMU. This is because VMware executes guest OS code directly on the CPU and therefore needs to create its own IDT, while QEMU can present an address that would be reasonable on a real machine.

Running the 71 malware samples in a clean image of Windows XP in VMware, we have observed the following:

- 19 samples take no observable action
- 12 samples deletes itself upon completion of execution
- 10 samples have a pop-up box that states that the program is "unable to run under a virtual machine"
- 9 samples immediately log the user out of Windows
- 7 samples have a pop-up box that states that it is "not a valid Win32 application"
- 7 samples crash/run into an error and launch the Visual Studio debugger
- 3 samples have a pop-up box that states that it is "not a valid Windows image"
- 2 samples have a pop-up box that states that it "failed to initialize properly"
- 1 sample has a pop-up box that states that it has an "Internal error"
- 1 sample has a pop-up box that says "UNLICENSED DEMO! This application is protected by the demo version of Obsidium. Do no [sic] distribute this application. The full version will not display any messages." and later "File integrity violated"

Running the samples in QEMU, we found the samples that logged out of Windows in VMware did not log out in QEMU. The samples that were "unable to run under a virtual machine" did execute in QEMU. All "log out" and "unable to run under a VM" samples were caught by our system. Their actions in the VM appeared to be an indication of VM awareness. Most of these VM samples showed more malicious behavior in the QEMU environment, where the SIDT/SGDT/SLDT instruction does not alert the malware to the presence of the emulator. These behaviors include spawning another process with system process-like names, deleting itself upon execution, trying to connect to remote hosts, etc. Additionally, we were able to observe that a popular software

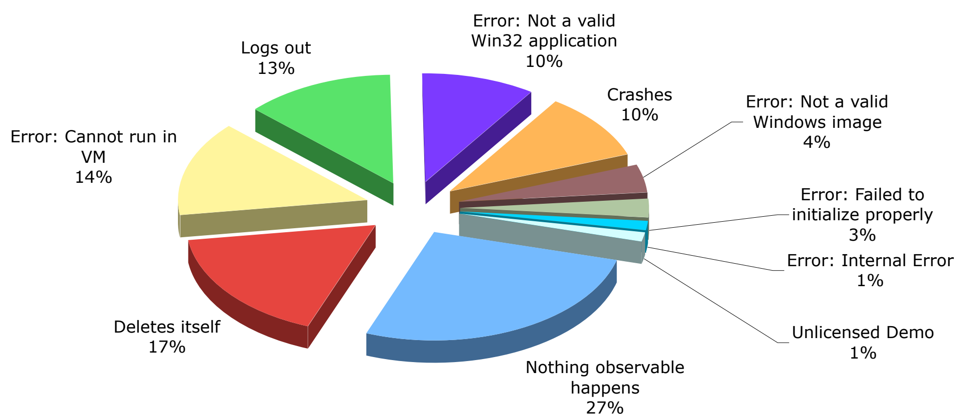

Figure 1: This pie chart shows the malware categorized by their observable actions when run in VMware.

protection system, Themida, has been used to obfuscate many of the malware samples. These were the ones that claimed that they could not run in a virtual machine.

The samples that had image or Win32 errors still did not run in QEMU. They may use more advanced VM-detection mechanisms but this still needs to be confirmed on a physical host.

We observed the malicious behavior in QEMU in 52 samples. Of these 52 samples, 37 of them exhibited different behavior when run in VMware, confirming that these samples are "VMware"-aware. Of these 37 confirmed samples, our system identified 28 of them as using the SIDT instruction, giving us a 76% detection rate with this method. This indicates that a majority of malware writers have taken the basic SIDT approach to VM-detection.

The remaining 19 samples that did not exhibit malicious behavior in QEMU require further

investigation and analysis. It is possible that some of these binaries are faulty or run in a different OS. On the other hand, it is possible that these samples are triggered by some other action or environment.

We also tested our system for false positives. We ran three programs, Notepad, Calculator, and Windows Media Player, and none of these programs were found to flag our system. These results were expected as the SIDT/SLDT/SGDT should only be legitimately called by the OS and not a user level process.

5.3 Challenges

This project has been an interesting and rewarding experience. However, it is not without its challenges. Initially, we had some difficulty setting up the TEMU infrastructure. This is partly due to the large amount of the package dependencies it has. It also had specific Linux distribution and version requirements that were not clearly stated in its documentation. Some documentation was distributed in different locations rather than centrally located. This made it more difficult for us to grasp the overall code structure. On the technical side, it is quite a large code base with a lot of of low-level intricacies, so the initial learning curve of the system was relatively high.

Malware analysis is inherently risky. There was limited network protection and physical resources to allow us to freely experiment with the malware samples on physical hosts, which can be used to identify VM-awareness. We also suspect some malware will only exhibit malicious behavior when a network connection is present.

Although QEMU is a great emulation platform, it is extremely slow, particularly because we had to run it in emulation mode to get the benefits of having additional control. Analysis of a large malware sample took a considerable amount of time.

The above mentioned challenges negatively impacted our progress and caused our mid-course correction to limit our scope to a more restricted set of detection methods than we had envisioned.

6 Related work

Most of the known mitigation techniques take smaller, more localized approaches. The most basic approach is to statically analyze binaries and check for matching signatures of confirmed VM-aware malware.

VMmutate takes a slightly more brute force approach [16]. Knowing that a large weakness of the VMware is its static "magic" number for command channel authentication, it targets the VMware code and changes all references of the "magic" number to a user chosen value. This approach, however, has been known to have numerous false positives and it has also be known to break keyboard and mouse functionality.

Other systems take an arms race approach and modify the platform or hardware to resist malware. By altering Linux, one can prevent any memory accesses from Ring 3 [16]. Furthermore, RedPill can also be defeated by using multicore processors [13].

VMware has also added some additional (undocumented) features to their software in efforts to mitigate VM detection [16]. In it, a user can alter a text file (with extension .vmx) to customize the guest host. Through this file, the user can turn off the guest-to-host communication backdoor that VM-aware malware commonly targets. The problem of this technique is that these options often break other features of the VMware and as they are undocumented features, they may have side effects and are not guaranteed to work.

7 Limitations and Future Work

There are a few limitations to our approach. Our signature-based approach is a reactive solution and is not effective against unknown VM detection mechanisms. Our generalized behavior-based approach can effectively address this problem, but it is vulnerable to a high false positive rate.

We plan to continue this work, focusing on implementing and experimenting with the second approach which uses dynamic taint tracking to detect any impact caused by input that changes the execution path of the malware. The key issues to be addressed include the interpretation of tainted branching at OS semantics, visualization of the tainted trail, and the potential use of instructions near the branching point and target to reduce false positives.

As with most approaches, adversaries will always try to evade detection or defeat our scheme. We examine some ways adversaries will do this. QEMU is a virtual machine itself and can be detected in various ways. However, such detection will still be subjected to inspection by Malaware. If Malaware can successfully detect attacks against a normal VM, it should be able to detect similar attacks against QEMU. Even if malware examines QEMU-specific information and then uses this information in a branching point, Malaware will be able to detect this.

One of our concerns with Malaware is that the adversary can insert unnecessary conditional jumps based on sensitive input and thus launch a denial of service attack against our behavior-based method. This can be mitigated if we find the exact point of deviation by comparing the differences in execution logs on a real machine and virtual machine. However, this is a fundamentally difficult problem because the complexity explodes exponentially as these malicious branching instructions are inserted. This will involve further investigation.

8 Conclusion

In this paper, we present a two-fold approach to identifying VM-aware malware. The first approach uses dynamic analysis to identify known virtual machine detection techniques. This approach has the benefits of high accuracy with low overhead. The second approach uses dynamic taint tracking to detect any impact caused by input that changes the execution path of the malware. This approach has the benefit of discovering new virtual machine detection techniques. These approaches together make our system, Malaware. As of this writing, Malaware has successfully detected the SIDT technique in known malware and malware samples from the wild.

9 Acknowledgments

We would like to thank Dawn Song and David Wagner for their guidance and Heng Yin for his help on TEMU. We would also like to thank Fabian Monrose and Moheeb Rajab for their malware binaries.

References

[1] Temu. http://bitblaze.cs.berkeley.edu/temu.html, 2007.

[2] Vmware. http://www.vmware.com/, November 2007.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference*. USENIX, 2005.

[4] F. Bellard. Qemu, February 2007.

[5] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.

[6] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[7] C. Garretson. One in six pcs could be infected with malware. *PC World*, 2007.

[8] Anup K. Ghosh. Code-driven attacks: The evolving internet threat.

[9] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*.

[10] T. Klein. Jerry - a(nother) VMware fingerprinter. http://www.trapkit.de/research/vmm/jerry/index.html, 2003.

[11] T. Klein. Scooby Doo - VMware fingerprint suite. http://www.trapkit.de/research/vmm/scoopydoo/index.html, 2003.

[12] J. Leyden. Malware wars: Are hackers on top? *The Register*, 2006.

[13] D. Quist and V. Smith. Detecting the presence of virtual machines using the local data table. http://www.offensivecomputing.net, 2006.

[14] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. *Proc. of ACM SIGCOMM/USENIX Internet Measurement Conference*, October 2006.

[15] J. Rutkowska. Red pill... or how to detect VMMusing (almost) one CPU instruction. http://invisiblethings.org, 2004.

[16] E. Skoudis and T. Liston. On the cutting edge: Thwarting virtual machine detection. Presentation at SANS@Night, 2006.