Hiding Virtualization from Attackers and Malware

Matthew Carpenter, Tom Liston, and Ed Skoudis
Intelguardians

Virtual machine environments (VMEs) let a user or administrator run one or more guest operating systems on top of a host operating system—for example, three or four instances of the Microsoft Windows operating system could run as guest systems on a Linux host operating system on a single PC or server. Such environments are widely used as clients or servers in a variety of commercial, government, and military organizations. Beyond normal business operations, security researchers and honeypot technologies often leverage VMEs to analyze malicious code discovered in the wild to determine its functionality, business model, origin, and author. Because VMEs offer useful monitoring and isolation capabilities, malware researchers are increasingly reliant on these products to conduct their trade. Furthermore, a malware researcher can use the "snapshot" capabilities of some VMEs to create a pristine uninfected picture, infect that machine, observe the infection's impact, and restore the system to a pristine state quickly and easily so that the researcher can move on to review another specimen. Indeed, various malware researchers in antivirus and antispyware companies are automating malicious code analysis with large numbers of VMEs that have such snapshot capabilities.

With security researchers relying on VMEs in their analysis work, attackers and their malicious code have a significant stake in detecting the presence of a virtual machine. Virtualization, by its very nature, creates systems that have different characteristics from real machines. From a theoretical perspective, any difference between the virtual and the real could lead to a fingerprinting opportunity for attackers. This article focuses on detection techniques and mitigation options for the most widely deployed VME product today, VMware.

Threats

Attackers often use VME detection to confound security researchers. Because malicious code analysis experts frequently use VMEs when dissecting malicious programs, some of the most cutting-edge malware specimens can detect virtual machine containment and modify their behavior to hide the code's full functionality. VME-detecting malware might even behave in an entirely benign fashion inside a VME, to the point that a malware researcher might not realize its true destructive nature. When this detection is coupled with existing code-obfuscation techniques, it can be very difficult for researchers to identify the malicious code's full behavior, thus causing costly delays for antivirus vendors and leaving millions of computer systems vulnerable.

VME detection could evolve into a dangerous game of cat and mouse if attackers can discover flaws in the underlying VME code. Essentially, VMEs are a complex layer of software that usually tries to isolate the host and guest operating systems. Software developers know that any major, complex software package often has security flaws. If an attacker can find a flaw in the VME-provided host/guest isolation, virtual machine detection could become a significant security risk as a precursor to VME escape—a procedure in which malicious code running inside a guest machine can escape and begin running on the host. Although no public VME escape tools are available today, such attacks are theoretically possible and are an active area of research. In a production server environment, attackers who discover a VME can look for exploits to escape the guest and attempt to break into other guest or host server systems. Likewise, malicious code on a guest machine in a production client environment could try to infect other guest systems.

VME detection techniques

The most popular VMEs today implement virtualized x86 PC systems as guest machines running on top of x86 host systems. Each guest has a view of a virtualized processor and its own virtualized hardware, which makes the software running inside a guest machine appear to run on a completely separate machine from the host. To detect VMware, malware typically relies on one of two different aspects.

VMware communications channel

VMware allows for communication between host and guest operating systems via a custom communications channel hard-coded into all major VMware products. This channel lets the guest and host operating systems interact for a variety of functions, including improved GUI performance, support for moving data in and out of the host clipboard, and dragging and dropping files between guest and host.

As part of our research at Intelguardians, we've extensively studied the protocol this communications channel uses to understand how malicious code interacts with the channel. Two years ago, in an incident response engagement for a client infected with malware, we discovered a specimen with a small snippet of code that checked for this communications channel's presence. We discovered the code only because the executable exhibited noticeably different behavior than we expected when it ran in the VME. The executable carrying this code falls into a class of malicious software known as *cascading file droppers*. A file dropper is simply an executable that carries another program as encoded data and, when executed, decodes that data, writes it to another file, and (usually) executes it. In the case of a cascading file dropper, the dropped file itself is also a file dropper, a nice recursive twist often found in modern malware.

When we executed the program in a virtual machine in our investigation, nothing appeared to happen. But upon closer examination, we found that when the second stage of the cascade executed, the following code snippet attempted to detect the presence of VMware by invoking the VMware communications channel:

```
MOV EAX, 564D5868 <—
"VMXh"
MOV EBX, 0
MOV ECX, 0A
MOV EDX, 5658 <— "VX"
IN EAX, DX <— Check for
VMWare
CMP EBX, 564D5868
```

In this machine language snippet, the program first loads the hexadecimal value 564D5868 into register EAX. This value, which is the equivalent of ASCII "VMXh", is hard-coded into VMware and represents the magical incantation required to invoke the communications channel, acting rather like a fixed password for the channel. Next, the program loads the number zero into register EBX, clearing out the place where our result will be stored later. Then, it loads the value 10 (hexadecimal 0A) into register ECX, which will tell the VMware communications channel what we want to do. (The 0A value indicates that we want to perform a VMware version check.) We then load into register EDX a value of 5658 (which is ASCII "VX"), a specialized hardware port associated with VMware. After initializing our registers in this way, the program is ready to test for the presence of VMware by using the `IN` instruction.

An x86 processor normally uses the `IN` instruction to read data from a hardware device such as a modem, but VMware has extended the `IN` instruction's capabilities for guest machines to implement its communications channel. When a program calls the `IN` instruction to pull data from port "VX" while register EAX holds "VMXh," for example, VMware traps the I/O call. Instead of really reading data from that port, VMware moves the magic value "VMXh" into register EBX. Thus, a simple compare of register EBX with "VMXh" can tell us whether our code is running in a VMware guest. In a VMware guest machine, our comparison will evaluate to positive, but in a machine that isn't a VMware guest, these instructions will trigger exception-handling code, which is the actual payload of the malware itself.

The use of this type of detection code in the wild shows that the computer underground is well aware of VMware's widespread use in malicious code research, and that easily detectable VMEs are becoming a liability for malware researchers. Another publicly released tool called Jerry.c by Tobias Klein also identifies the presence of a VME with the technique we just described (www.trapkit.de/research/vmm/). In our research, we've found that detecting the VMware communications channel is the single most popular method for VME detection today.

With security researchers relying on VMEs in their analysis work, attackers and their malicious code have a significant stake in detecting the presence of a virtual machine.

Taking the Red Pill

Beyond measuring the `IN` command's specialized behavior in VMware, other methods for VME detection exist. Because the guest operating system is virtualized by software running on the host operating system and shares the same physical memory, a VME typically introduces some differences in the location of mapped global items in memory. In particular, as John Robin and Cynthia Irvine originally described (www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf), the locations of the Interrupt Descriptor Table (IDT), the Global Descriptor Table (GDT), and the Local Descriptor Table (LDT) predictably vary between host operating systems and guest machines. By looking at the memory locations of these critical operating system objects, an attacker or malicious code could detect a virtual machine.

The first publicly released tool to use this technique was the Red Pill, which security researcher Joanna Rutkowska released in November 2004 to inspect the contents of the Interrupt Descriptor Table Register (IDTR) via the SIDT (Store the Interrupt Descriptor Table) instruction. Rutkowska observed that on VMware guest machines, the IDT is typically located at `0xffXXXXXX`, whereas for host operating systems, it's far lower in memory. The Red Pill program deduces that it's running in a guest machine if the IDTR is greater than `0xd0000000`. Our team found that the results were highly accurate for VMware running in a variety of Linux and Windows operating systems. Likewise, Scoopy—another program from Klein—looks at the location of the Interrupt Descriptor Table, the Global Descriptor Table, and the Local Descriptor Table using similar techniques to the Red Pill.

It's worth noting that many of these same memory anomalies appear in multiprocessor or multicore environments as well. Therefore, as multicore processors become increasingly prevalent, this VME detection method will become increasingly inaccurate, possibly forcing attackers to rely on the already popular VMware communications channel detection technique.

Mitigation techniques: Proof of concept

To dodge VME-detecting malware, researchers can rely on several different methods to disguise a virtual machine. We've identified two particularly useful approaches to foil the most popular VME detection mechanisms used by malware.

Undocumented VMware options

VMware `VMX` configuration files contain various parameters for a guest machine that a VMware administrator can change. This file is typically located in the host operating system, and it controls various settings for the guest machine.

A long list of `VMX` configuration parameters can be changed or added to the `VMX` file. (See www.vmware.com/community/thread.jspa?threadID=37190&tstart=0, www.vmts.net/vmbkmanual.htm, and www.easyvmx.com/expertform.shtml for some well-documented features and settings.) Through various sources and experiments, we've also identified several undocumented configuration options that can control or eliminate behaviors that allow VMware detection. For example, setting the following parameters in the `VMX` file will stop Jerry.c from detecting VMware by tweaking the behavior of the communications channel version-check functionality:

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"

isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
```

Although the `isolation.tools.setVersion` and `getVersion` configuration options also prevent Jerry.c's detection method from working, they don't stop the IDT-based detection method that the Red Pill and Scoopy use. To prevent both from detecting the presence of VMware, we must change several additional `VMX` configuration properties:

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"

monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"

monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"

monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

These specific settings alter VMware's memory-relocation functionality and also modify its binary translation (BT) functionality. BT is the method by which VMware virtualizes systems—by altering some of the guest's machine language instructions before they have a chance to execute in the host. Although setting these configuration options will stop local detection of VMware via Red Pill and Scoopy, they're neither documented nor officially supported by VMware, so the full impact on the guest system's functionality isn't well known. Furthermore, an organization that uses guests with such configurations won't likely be able to get vendor support for their installations using these options.

These VMX file-configuration changes can block the most popular detection techniques in current use, but the guest machine configuration severely restricts functionality, thus degrading or disabling many of the ease-of-use features VMware provides, such as drag-and-drop, cut-and-paste via the clipboard, and shared file directories. Fortunately, malicious code researchers rarely require such functionality. A stealthy guest is less useful for general-purpose computing, but is adequate for most malware researchers who simply want to infect a machine to inspect malicious code functionality.

Altering the magic value

Because of the undesirable side effects caused when a researcher or administrator uses the VMX file-configuration options to mitigate detection, we searched for an alternative method to thwart VMware detection. Knowing that Jerry.c-style detection of VMware's command channel is the most prevalent attacker method, our research focused on blocking this technique.

One effective method we found was to disable or change (by patching the VMware binary executable file) the magic value of `VMXh` associated with the communications channel. Perhaps the best-known implementation of binary patching for this purpose is Kostya Kortchinsky's Honey-VMware patch (http://honeynet.rstack.org/reports/r2005_2.html). This tool only disables the command channel in the Linux version of VMware Workstation 5.0, but the concepts should apply to all versions of VMware and can be leveraged to disable the command channel as well as change its characteristics. Unfortunately, both the host-side VMware program itself and the guest program's VMware tools need modification to alter the VMXh value. Perhaps future versions of VMware will make this value adjustable in both the guest and the host.

As part of our research, we developed a tool called VMmutate, which alters the VMware binary and searches through a VMware disk image, selectively modifying each instance of the VMXh value to a user-defined alternative. In a file as large as most multigigabyte VMware disk images, a program like VMmutate will likely find VMXh numerous times, simply by chance and often having nothing to do with the VMware communications channel. To avoid false positives that would alter noncommunications channel VMXh instances, VMmutate contains code that looks at the value's context before altering it. Although VMmutate is still beta-level software, modifying the command channel is a feasible method for disguising virtual machines.

Although we've successfully blocked VME detection by using VMware's undocumented features and modifying the VMware binary program, both come with a price: a loss of functionality. Furthermore, VME detection is indeed an arms race. Although the techniques covered in this article stop VME detection by most of today's malware, computer attackers are a clever bunch, and they'll surely raise the stakes by devising other detection mechanisms. Although VME detection is a budding area of research, we wholeheartedly expect malicious software and attackers to continue to leverage this information against their targets. ▢

Acknowledgments

This work was supported by the US Dept. of Homeland Security (DHS) Science and Technology (S&T) Directorate and the Air Force Research Laboratory (AFRL) under contract number FA 8750-05-C. Points of view expressed in this document are those of the authors and do not represent the official position of the DHS S&T Directorate or the AFRL.

***Ed Skoudis** is a senior security analyst and founder of Intelguardians. His research interests include virtual machine security issues, malicious code analysis, and the evolution of computer attack tools. Skoudis has an MS in information networking from Carnegie Mellon University. Contact him at ed@intelguardians.com.*

***Tom Liston** is a senior security consultant for Intelguardians. He serves as an incident handler for the SANS Institute's Internet Storm Center and has written several computer security programs. Contact him at tom@intelguardians.com.*

***Matthew Carpenter** is a senior security analyst at Intelguardians. His research interests include security vulnerability research, binary executable analysis, and forensic analysis. Carpenter has a BA in computer science from Anderson University. Contact him at mcarpenter@intelguardians.com.*