Alternative Medicine: The Malware Analyst's Blue Pill

Paul Royal
Damballa, Inc
paul@damballa.com

ABSTRACT

Malware has become the centerpiece of many security threats on the Internet. Malware analysis is important for information security practitioners because it is the basis for understanding the *intentions* of malicious programs. Current malware analysis approaches reside in the guest OS or emulate part of its underlying hardware, which leaves them vulnerable to detection and attack by modern malware.

This paper presents an alternative, external approach to malware analysis. The resulting malware analysis tool, called *Azure*, operates outside of the guest through the use of hardware virtualization extensions such as Intel VT. Experiments with packers used to create the majority of modern malware show that Azure is an effective, transparent tool for malware analysis.

1. INTRODUCTION

Malware has become an artifact whose use intersects multiple major security threats faced by information security practitioners. Given the financially motivated nature of these threats, methods of recovery now mandate more than just remediation–knowing what occurred after a system became compromised is as valuable as knowing it was compromised. Concisely, independent of simple detection, there exists a pronounced need to *understand* the intentions of modern malware.

Before advances in malware analysis can be used to determine what a malware instance does or might do, the runtime behavior of that instance and/or an unobstructed view of its code must be obtained. However, modern malware contain a myriad of anti-debugging, anti-instrumentation, and anti-VM techniques to stymie attempts at runtime observation [6, 8, 9, 13]. Given that almost all analysis approaches reside in the guest OS or emulate its underlying hardware, little effort is required by a knowledgeable adversary to detect their existence and evade.

This paper presents an alternative, external approach to malware analysis. Its creation required diverging from existing approaches that employ in-guest components, API virtualization or full system emulation, as these implementations are often vulnerable to detection techniques used by malware. Based on novel application of hardware virtualization extensions such as Intel VT [2], the resulting prototype analyzer–called *Azure*–resides completely outside of the target OS environment. The results of testing (presented in Section 3) show that Azure is able to identify, instrument and trace the vast majority of modern malware.

The remainder of this paper is organized as follows. Section 2 provides an in-depth explanation of *how* hardware virtualization extensions are leveraged and describes Azure's implementation. Section 3 details how experimentation was performed and provides an analysis of the results. Section 4 briefly provides some concluding remarks.

2. DESIGN AND IMPLEMENTATION

This section describes how hardware virtualization extensions can serve as means to perform transparent, external malware analysis. It provides a brief overview of Intel VT, then describes how its features and attributes can be used to implement virtual machine introspection, fine- and coarse-granularity tracing. It concludes with a presentation of Azure, a proof-of-concept implementation that uses Intel VT to externally identify, instrument, and perform fine-grained tracing on a malware instance.

2.1 Intel VT

Introduced at the Intel Developer's Forum in 2005, Intel VT is a hardware-assisted means of facilitating virtualization of the x86 instruction set. Its operation is supported through a combination of hardware elements (e.g., the virtual machine control structure), virtualization instructions (e.g., VMXON, VMRESUME), and a supporting administrative software component. These extensions are one way for allowing the execution of unmodified guest operating systems inside a host operating system.

Under Intel VT, both the host and guests can freely execute in rings 0-3. The difference is that the host operates in a special mode, called VMX root mode, while guests operate in VMX non-root mode. VMX root mode allows the host to read from, write to, and receive notification for (then handle, forward or discard) a subset of events that otherwise occur in or would be delivered to the guest. As the guest operates in VMX non-root mode, when an event occurs that will instead be delivered to the host, the guest is frozen, and a transition from VMX non-root mode to VMX root mode (called a VMExit) takes place. The administrative software component on the host uses VMX root mode privileges to provide a number of support services to the guests (e.g., memory management services).

The administrative powers provided to the host by Intel VT (e.g., privilege to read from, write to, and in some cases preempt the guest) suggests the possibility of its use in malware analysis. However, given that Intel VT offers (for the most part) only what is necessary to support guest operation, mechanisms for introspection, instrumentation and tracing must be derived from existing functionality. With this challenge as a pretext, the next several subsections describe the (sometimes) unobvious use of Intel VT to externally identify, instrument, and observe the behavior of programs.

2.2 Virtual Machine Introspection

Under Intel VT, the host must act as the guest's memory management unit (MMU). To support the role of guest MMU, Intel VT provides the host with the ability to receive preemptive notification of certain memory-related events, such as page faults. Notification is provided in the form of a VMExit, where (as de-

scribed previously) the guest is frozen and a transition to VMX root mode occurs. When combined with the operational particulars of virtual memory on the x86 architecture, the MMU job function of the host can be leveraged to facilitate *virtual machine introspection*.

Defined in [10], virtual machine introspection is the approach of inspecting a guest process externally for the purposes of analysis. As an example of Intel VT's ability to support virtual machine introspection, consider the requirement of a malware analysis tool to monitor a target process from the start of its execution. A prerequisite to this requirement is the *identification* of the process during or after loading but before the beginning of execution. In Intel VT the host's MMU responsibilities can be used to perform this identification, as notification is provided to the host each time the guest performs a context switch.

In x86, the CR3 register contains the page directory pointer of the current process. During a context switch, a MOV to CR3 is used to set the upcoming process' page directory pointer. When the guest OS performs a context switch and attempts to change to the value of its CR3 register, a VMExit occurs that the host (as the guest's MMU) resolves accordingly. Since the guest remains frozen until the host issues a VMRESUME, a series of guest reads can be used to identify the upcoming process *before* it begins execution.

2.3 Fine-Grained Tracing

In malware analysis, numerous approaches require monitoring a program's behavior with instruction-level granularity. Techniques that use low-level or *fine-grained* tracing include dynamic taint analysis, multi-path exploration and precision automated unpacking [11,14,17]. While not as straightforward to implement as virtual machine introspection, external fine-grained tracing can also be performed using Intel VT.

In x86, the FLAGS register contains the set of processor status, control, and system flags. One of the system flags, called the *trap flag*, can be used to enable single-stepping mode. When the trap flag is set, a debug exception (which clears the flag) is thrown immediately after execution of the next instruction. Previous malware analysis frameworks such as [15] have used this functionality to perform fine-grained tracing by installing a custom exception handler in the guest's kernel and then repeatedly setting the trap flag during execution of a target process.

By leveraging functionality offered by Intel VT that is not explicitly required to support normal guest operation, a similar approach can be used to perform fine-grained tracing *externally*. First, the host can use its access to the guest's registers to set the trap flag in the guest's FLAGS register. Then, as a substitute to installing an in-guest exception handler, the host can use Intel VT to preempt receipt of (and then discard instead of delivering) the debug exception thrown when the guest executes the next instruction. By repeatedly setting the guest's trap flag and preempting the resulting exception, the host can externally trace a target process in the guest with instruction-level granularity.

2.4 Coarse-Grained Tracing

In contrast to fine-grained tracing, high-level or *coarse-grained* tracing involves monitoring the behavior of a process at the API or system call levels. Discrete events in coarse-grained tracing often represent easily identifiable actions, such as file or (Windows) registry access, process and thread creation, and network activity. Tools and services that use coarse-grained analysis include automated malware analysis environments [1,16] and behavioral antivirus [5]. Through novel exploitation of the fast system call facility present in modern x86 processors, Intel VT can also be used to perform external coarse-grained tracing.

In the x86 versions of Windows (XP+) and Linux (2.6+), the SYSENTER instruction is used to transition from ring 3 (user space) to ring 0 (kernel space) whenever a process makes a Native API or system call. During execution, SYSENTER reads from model specific registers to set OS-defined values for the kernel's system call handler. Among these registers, SYSENTER_EIP_MSR is used to set the instruction pointer to the address of the handler's entry point.

First proposed by Dinaburg [7], the host's MMU duties can be combined with the guest's use of the fast system call facility to perform external coarse-grained tracing. To enable tracing, the host first sets the value of the guest's SYSENTER_EIP_MSR to an unallocated address in kernel memory space. At each subsequent system call, SYSENTER will set the instruction pointer to the unallocated address. When the guest attempts to fetch the first instruction following its transition to kernel space, a page fault will occur that can be preempted by the host. The host can then restore the guest's instruction pointer to the original value of SYSENTER_EIP_MSR and resume its execution.

2.5 Azure

Named after the rootkit [3] that relies on similar principles for its operation, Azure is a proof-of-concept malware analysis tool for Windows XP-based guests that functions externally through the use of Intel VT. It was implemented using KVM [4] (a Linux-based virtualization solution) as a base. Azure uses virtual machine introspection to identify a target process and fine-grained tracing to monitor its behavior; coarse-grained tracing is left as future work.

To identify the target process, Azure performs a series of guest reads to obtain the upcoming process name whenever a context switch occurs in the guest. After the host sets the guest's page directory pointer, Azure reads at a fixed offset from FS:[0] to obtain the guest virtual address of ETHREAD, which contains the address of EPROCESS and the process' name. If the process name is a match, Azure records the process' CR3 and performs additional reads using members of EPROCESS to obtain information from structures such as the process' PEB.

Upon identifying the target process, Azure performs fine-grained tracing until the target process is switched out (tracing resumes whenever the target process is switched back in). To begin tracing, Azure sets the trap flag in the guest's FLAGS register and updates the *exception bitmap* to provide preemptive notification to the host whenever the debug exception caused by the trap flag is raised. The guest is then resumed, and immediately after execution of the next instruction, a debug exception occurs that returns control to Azure. This process (setting the trap flag, preempting receipt of the corresponding exception) is repeated until the target process terminates or is switched out.

3. EXPERIMENTATION AND EVALUATION

This section examines the effectiveness of using Intel VT to externally monitor the behavior of modern malware. To perform evaluation, Azure's usefulness as an automated unpacker was compared against Saffron [12] and Renovo [11], approaches that employ in-guest components and whole-system emulation, respectively. The results show that the external malware analysis techniques used by Azure offer significant transparency.

3.1 Automated Unpacking

While outside the scope of this paper, Azure's fine-grained tracing could be used to create a precision automated unpacker. During tracing the unpacker could read, disassemble, and track

memory-write instructions executed by the target process. Whenever the instruction pointer contains an address in the set of locations that were previously written, a series of guest reads could be used to snapshot the dynamically generated code. By clearing the set of write locations after each snapshot, the unpacker could also detect and extract the unpacked code of malware instances that contain multiple packing layers.

To test Azure alongside other approaches for automated unpacking, the exact set of synthetically packed samples used to evaluate Renovo was obtained. This set represents packers used to obfuscate the vast majority of modern malware. The ability of each approach to successfully unpack a given sample was determined by searching the unpacked code layer(s) for the original program's code. If the unpacked code contains a 32 byte string representing instructions located at a fixed offset from the entry point of the original (non-packed) program, the corresponding packed sample is marked as successfully unpacked.

Due to time limitations, an alternative approach was used to evaluate Azure's ability to perform automated unpacking. Instead of detecting and outputting snapshots of unpacked code, after each instruction Azure reads 32 bytes starting at the address of the guest instruction pointer. This data is then compared to the aforementioned 32 byte string found in the original, non-packed sample. A match indicates that Azure successfully traced the packed sample through execution of the original program's code.

3.2 Unpacking Results

Packing Tool	Azure	Renovo	Saffron
Armadillo	yes	no	no
Aspack	yes	yes	yes
Asprotect	yes	yes	yes
FSG	yes	yes	yes
MEW	yes	yes	yes
Molebox	yes	yes	part
Morphine	yes	yes	yes
Obsidium	yes	no	part
PECompact	yes	yes	yes
Themida	yes	yes	part
Themida VM	yes	part	part
UPX	yes	yes	yes
UPX S	yes	yes	yes
WinUPack	yes	yes	yes
yoda's Prot.	yes	yes	yes

Table 1: Effectiveness of Automated Unpackers

The results of experimentation are shown in Table 1. A label of *yes* indicates that the approach successfully unpacked the sample produced by the corresponding packer. The *part* label was given when an approach produced unpacked code, but none of the unpacked layer(s) contained the 32 byte string of the original program. A label of *no* indicates that the approach produced no unpacked code.

An analysis of the results indicates that Renovo was unable to produce unpacked code for Armadillo and Obsidium due to incorrect system emulation; the reason behind its Themida VM result is unclear. The results for Saffron suggest that some packers detected its in-guest presence and evaded before execution of the original program code. In contrast, Azure successfully identified the original program's code for all samples, which demonstrates its use as an effective malware analysis tool.

4. CONCLUSION

In order to be effective, malware analysis tools must remain transparent to an increasing number of malicious programs that employ anti-debugging, anti-instrumentation, and anti-VM techniques. Most malware analysis approaches reside in the guest OS or emulate part of its underlying hardware, which often makes them vulnerable to detection and attack. This paper has presented an alternative, external approach to malware analysis that uses hardware virtualization extensions to identify, instrument, and trace a malware instance. The results of experiments with packers used to create a large percentage of modern malware show that this approach offers significant transparency.

Acknowledgements. The author would like to thank Artem Dinaburg, Monirul Sharif, Wenke Lee, and Danny Quist for their advice and feedback.

5. REFERENCES

[1] Anubis: Analyzing Unknown Binaries. http://anubis.seclab.tuwien.ac.at.

[2] Intel Virtualization Technology. http://www.intel.com/technology/virtualization.

[3] Introducing Blue Pill. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[4] Kernel Based Virtual Machine. http://kvm.qumranet.com/kvmwiki.

[5] ThreatFire AntiVirus. http://www.threatfire.com.

[6] P. Bacher, T. Holz, M. Kotter, and G. Wicherski. Know your enemy: Tracking Botnets. http://www.honeynet.org/papers/bots, 2005.

[7] A. Dinaburg. Personal Correspondence. January 2008.

[8] P. Ferrie. Attacks on Virtual Machine Emulators. Symantec Advanced Threat Research, 2006.

[9] P. Ferrie. Anti-Unpacker Tricks. In *Proc. of the 2nd International CARO Workshop*, 2008.

[10] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of the 10th Network and Distributed Systems Security Symposium*, 2003.

[11] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proc. of the 5th ACM Workshop on Recurring Malcode*, 2007.

[12] D. Quist and Valsmith. Covert Debugging: Circumventing Software Armoring. In *Proc. of Black Hat USA 2007*, 2007.

[13] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *ISC*, pages 1–18, 2007.

[14] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proc. of the 23rd the Annual Computer Security Applications Conference*, 2006.

[15] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *Proc. of the 21st the Annual Computer Security Applications Conference*, 2005.

[16] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[17] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. of the ACM Conference on Computer and Communication Security*, 2007.