Entrapment: Tricking Malware with Transparent, Scalable Malware Analysis

Paul Royal
Georgia Tech Information Security Center
paul@gtisc.gatech.edu

ABSTRACT

The detection of malware analysis environments has become popular and commoditized. Detection techniques previously reserved for more sophisticated forms of malware are now available to any novice cyber criminal. The use of next-generation virtualization-based malware analysis technologies considerably reduces the number of possible transparency shortcomings, but still fails to handle pathologically resistant malware instances that will only run on physical hardware.

Traditionally, the execution of malware on physical (or baremetal) hardware has been useful for one or a handful of malware samples of interest. However, this activity was manually driven and time intensive (e.g., infect, study, format, reinstall). This paper proposes a way to resolve these long-outstanding shortcomings by describing the design and implementation of a scalable, automated baremetal malware analysis system, which can be constructed using inexpensive commodity hardware and freely available technologies.

1. INTRODUCTION

Malware is the centerpiece of current threats to the internet. Examples of malicious software behavior include the generation of unsolicited email, distributed denial-of-service attacks, and the theft of personal information and intellectual property. By aggregating the resources of compromised systems around the world, malware is used to construct criminal infrastructure that fulfills the financial goals associated with organized crime.

Malware analysis is the basis for understanding the intentions of malicious programs. As malware analysis technologies have evolved, so too have the mechanisms to detect them and mitigate their effectiveness. The commoditization and increasing popularity of analysis environment detections mandates that security practitioners develop mechanisms which extract intelligence from environment-aware malware in a tenable manner.

In an effort to address the intelligence-gathering gap created by analysis-resistant malware, this paper presents an approach to scalable, automated malware analysis that uses an ensemble of technologies to physicalize virtual machines and thus does not contain any of the detectable artifacts associated with virtualization. To ensure the accessibility of this approach by the general information security community, the corresponding system–called *NVMTrace*–uses inexpensive, commodity hardware and open-source software.

The remainder of this paper is organized as follows. Section 2 describes ways in which traditional malware analysis environments are detected by malware and presents previously unpublished detection attacks. Section 3 introduces the concept of physical (or baremetal) malware analysis and details the design and implementation of *NVMTrace*, a prototype baremetal malware analysis system. Section 4 briefly provides some concluding remarks.

2. MALWARE ANALYSIS DETECTION

This section describes a few of the ways in which modern malware detects analysis environments. It introduces examples of popular environment-aware malware, discusses detection commoditization, and concludes with the presentation of previously unpublished environment detection methods.

2.1 Popular Environment-aware Malware

Recent years have witnessed the integration of analysis environment detection in popular malware. For example, Conficker–one of the largest botnets ever created–checks for a relocated Local Descriptor Table [11]. This detection primarily affects more traditional x86 reduced privilege guest virtual machine monitors (e.g., VMware without Intel VT-x or AMD SVM).

Newer environments (e.g., KVM) use hardware virtualization extensions to operate a guest and thus do not need to relocate data structures such as the Local Descriptor Table or Interrupt Descriptor Table. However, these environments still employ device emulation for non-CPU components and can thus be detected. As an example, in 2011 variants of TDL4 began using the WMI Query Language (WQL) to identify the presence of an emulated hard disk drive [10].

Finally, environment detection can be a useful mechanism of restraint for malware that would otherwise propagate indiscriminately. As an example, Bredolab–which spreads itself through email–uses the DeviceIoControl() Windows API to achieve hard drive emulation detection similar to TDL4 [8]. By refusing to run in popular virtual environments (e.g., VMware, Virtual Box, QEMU), Bredolab can reduce unwanted attention when instances of the malware are automatically collected through spam traps and processed in malware analysis farms.

2.2 Detection Commoditization

While the traditional role of an underground obfuscation tool is to make the malicious portions of program code appear as seemingly benign data, more and more versions include anti-debugging, anti-instrumentation, and virtual machine detection techniques. Through these inexpensive tools, analysis detection methods previously reserved for more sophisticated forms of malware are now available to any novice malware author or botnet operator. As an example, RDG

Tejon Crypter, which lets the user detect any or all of ten popular environments that can be or are used for malware analysis, is available for less than US $50. A screenshot of the analysis detection portions of this tool is shown in Figure 1.

Figure 1: Analysis environment detections offered by the RDG Tejon Crypter obfuscation tool.

2.3 Detecting Specific Environments

Transparency shortcomings arise as a fundamental side effect of virtualizing an environment. The virtualization process may include emulating an x86 processor (e.g., QEMU), software binary translation and data structure relocation (e.g., VMware), or the use of extensions to the x86 ISA that create additional rings of privilege (e.g., KVM). Although their discovery and derivation can be involved, the resulting detection vulnerabilities are relatively easy to understand.

QEMU, a CPU emulator that serves as the core of popular malware analysis technologies such as Anubis (and VirtICE [4], presented at Black Hat 2010), is vulnerable to techniques that detect unfaithful CPU emulation. One previously unpublished example is the execution of the IRETD instruction with the 0x26 prefix [6]. As this prefix is not valid for IRETD, attempted execution of the corresponding instruction should generate a handleable exception. However, in QEMU, this illegal instruction produces an effect similar to a successful execution of IRETD, which causes the program to exit with an unhandled exception. Code for this detection attack, which works on all tested versions of QEMU (including 1.0.1), is presented in Appendix A.

Older versions of VMware, whose operation included software binary translation, likewise produce detectable differences in behavior. For example, some VMware products treat the SYSRET instruction as a NOP when it is executed in ring 3, instead of correctly raising an exception [6].

Finally, while the use of hardware virtualization extensions (e.g., Intel VT-x) significantly reduces the detection footprint, that same use can introduce opportunity for detection. Under hardware virtualization extensions, some instructions executed in the guest produce VMExits that must be handled by the software hypervisor. Unfortunately, these exits are not always handled correctly. As an example, guest execution of the VMREAD instruction on older versions of KVM (e.g., the version installed by Debian 5.0, codename Lenny) produces an unhandled VMExit, which results in termination of the virtual machine.

3. BAREMETAL MALWARE ANALYSIS

This section describes how to perform scalable, automated malware analysis without virtualization. It includes discussion of the challenges associated with using physical hardware for malware analysis, the technologies used to address those challenges, and concludes with a description of *NVMTrace*, a prototype software controller that automates the baremetal malware analysis process.

3.1 Baremetal Challenges and Solutions

Historically, baremetal malware analysis has been used only for one or a handful of malware samples because of its time-intensive nature (e.g., infect a physical system, study sample behavior, format, and reinstall). To be effective relative to the volume of genuinely new malware released each day, physical hardware must be used in a scalable, efficient manner. In addition to the general conveniences lost through the absence of virtualization, the challenges created by this mandate include resetting a physical system running untrusted software, quickly restoring its disk to a sterile condition for the next sample, and ensuring the longevity of the hardware used.

In order to manage physical system state, hardware with the Intelligent Platform Management Interface (IPMI [3]) is used. This technology allows a physical system to be remotely turned on, off, or reset regardless of that system's software state. As anecdotal evidence suggests that a power supply unit's duty cycle can quickly be reached when (during sample processing) it is turned on and off several hundred times each day, resets are used instead of power on and power off directives.

Instead of outfitting a physical system with an actual hard disk, a write-protectable USB stick is used. This stick is flashed with a modified version of the gpl'ed Preboot eXecution Environment (gPXE [2]), which acts as a lightweight network bootloader. When the system boots, gPXE is loaded, and a special piece of metadata returned with the DHCP lease instructs the system to disklessly boot off of an ATA-over-Ethernet (AoE [5]) target. This target corresponds to a Windows XP partition that resides on a Linux host as a copy-on-write block device. As it is created using the Linux Device Mapper [1], this block device can be destroyed and a clean version reassembled in a few hundred milliseconds.

The aggregate use of the technologies described above reduces and minimizes the inefficiencies traditionally associated with baremetal malware analysis. Through IPMI, a system's power state can be reliably reset without compromising hardware longevity. Similarly, through gPXE, AoE, and the Linux Device Mapper, a clean Windows environment can be quickly constructed for each malware sample. Finally, as these technologies are freely available, anyone can leverage them to build baremetal malware analysis systems.

3.2 Prototype Baremetal System

NVMTrace is the implementation of a software controller that facilitates automated baremetal malware analysis. It is used to operate one or more baremetal malware analysis clusters, each of which comprises one Linux host, eight baremetal processing nodes, and a network switch. For deployments at the Georgia Tech Information Security Center, a SuperMicro 5016I-MTF is used as the Linux host, while SuperMicro 5015A-PHFs act as the baremetal processing nodes; the switch is a Cisco WS-C2960-24TC-S.

The above components were selected carefully in order to minimize cost. Fully configured, the Linux host costs about $1200. Each baremetal processing node, which includes an integrated Intel Atom dual-core processor, is less than $350. The switch used to network everything together averages $420. Assuming a standard execution timeout, each *NVMTrace* cluster can process 2,850 malware samples per day. To achieve the desired processing scale, multiple clusters can be deployed; concurrency across clusters is a managed through a central database.

As part of processing each malware sample, *NVMTrace* makes the corresponding baremetal node's network traffic and disk contents available for analysis. While not as granular as some forms of virtualization-based malware analysis (e.g., Ether [7], which offers fine-grained tracing), there nonetheless exists opportunity to collect a rich amount of threat intelligence. For example, network traffic can be mined to extract structured representations of the sample's network-level activities (e.g., domains queried, HTTP requests made, emails sent). Similarly, the baremetal node's disk contents, which exist as a mountable block device on the Linux host, can be used to identify and record changes to the filesystem (e.g., files created or modified, registry keys created or changed). These outputs can be subsequently analyzed (e.g., using machine learning) to identify malicious infrastructure that an analysis-resistant sample would not have contacted if it were processed in a virtual machine.

4. CONCLUSION

In order to be effective, malware analysis tools must remain transparent to the samples they analyze. Fulfillment of this requirement is complicated by analysis-resistant malware that will only run on physical hardware. In an effort to handle environment-aware malware without an unacceptable loss of efficiency, this paper has presented a way to perform scalable, automated baremetal malware analysis. To encourage its use by the security community, *NVMTrace*–the corresponding prototype system–has been open sourced [9].

Acknowledgements. The author would like to thank Robert Edmonds, Michael Lee, Artem Dinaburg, and David Dagon for their advice and feedback.

5. REFERENCES

[1] Device-mapper Resource Page. http://sources.redhat.com/dm.
[2] Etherboot/gPXE Wiki. http://etherboot.org/wiki/start.
[3] Supermicro Intelligent Management. http://www.supermicro.com/products/nfo/IPMI.cfm.
[4] Q. N. Anh and K. Suzaki. Virt-ICE: Next Generation Debugger for Malware Analysis. In *Proc. of Black Hat USA 2010*, 2010.
[5] B. Coile and S. Hopkins. AoE (ATA over Ethernet). http://support.coraid.com/documents/AoEr11.txt.
[6] A. Dinaburg. Personal Correspondence. March 2010.
[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of the 15th ACM Conference on Computer and Communications Security*, 2008.
[8] S. Fagerland. Anti-VM check through IOCTL_STORAGE_QUERY_PROPERTY. http://blogs.norman.com/2011/malware-detection-team/anti-vm-check-through-ioctl_storage_query_property, 2011.
[9] P. Royal. NVMTrace: Proof-of-concept automated baremetal malware analysis framework. http://code.google.com/p/nvmtrace.
[10] w4kfu. TDL4, IDA. http://blog.w4kfu.com/post/news_tdl4_ida, 2011.
[11] B. Zdrnja. More tricks from Conficker and VM detection. https://isc.sans.edu/diary.html?storyid=5842, 2009.

APPENDIX

A. QEMU DETECTION CODE

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

int seh_handler(struct _EXCEPTION_RECORD *exception_record,
                void *established_frame,
                struct _CONTEXT *context_record,
                void *dispatcher_context)
{
        printf("Malicious code here.\n");
        exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {

        unsigned int handler = (unsigned int) seh_handler;

        printf("Attempting QEMU detection.\n");

        __asm("movl %0, %%eax\n\t"
              "pushl %%eax\n\t"::
              "r" (handler): "%eax");

        __asm("pushl %fs:0\n\t"
             "movl %esp, %fs:0\n\t");

        __asm(".byte 0x26, 0xcf");

        __asm("movl %esp, %eax");
        __asm("movl %eax, %fs:0");
        __asm("addl $8, %esp");

        return EXIT_FAILURE;
}
```