Dynamic Malware Analysis Using IntroVirt: a Modified Hypervisor-Based System

Stephen R. Pape, Joshua S. White, Adam T. Meily, Richard M. Gloo
{papes, whitej, meilya, gloor}@ainfosec.com
Assured Information Security, 153 Brooks Road, Rome, NY USA

ABSTRACT

In this paper, we present a system for Dynamic Malware Analysis which incorporates the use of IntroVirt™. IntroVirt is an introspective hypervisor architecture and infrastructure that supports advanced analysis techniques for stealth-malware-analysis. This system allows for complete guest monitoring and interaction, including the manipulation and blocking of system calls. IntroVirt is capable of bypassing virtual machine detection capabilities of even the most sophisticated malware, by spoofing returns to system call responses. Additional fuzzing capabilities can be employed to detect both malware vulnerabilities and polymorphism.

Keywords: Introspection, Hypervisor, Virtualization, VMM, Malware Detection

1. INTRODUCTION

1.1 Malware Analysis Approaches

Software which deliberately fulfills the harmful intent of an attacker is known as malware. Malware comes in various forms, classified as worms, viruses, bots, trojans, and can be further subcategorized by malicious intent. Categorizing malware types is just the first step to discovering these threats. Anti-virus tools first must use manually created signatures to identify malicious software. Before such a signature can be created, an analyst must determine whether or not the sample is actually malicious. This step can be automated, however all automated techniques contain flaws which creative malware writers can then exploit.

The study of malware analysis is broken down into two (2) types: 1.) static analysis, 2.) dynamic analysis. Malware analysis, both static and dynamic, is a very well studied and understood area of research, with new methods frequently being developed. Included in this body of knowledge are a number of comprehensive works on the subject of malware analysis techniques.

1.1.1 Static Analysis

Static analysis is the study of a malicious program without actually executing it. A number of tools and methods exist including viewing the executable with a Hex Editor, disassembling the binary, binary analysis to look for packers, and various mathematical clustering algorithms to look for similarities to known malware.

Static analysis techniques go further than just analyzing malicious code; if the source for a particular piece of software is available, static analysis tools can be used to prove the correctness of software engineering models and even find security related flaws in the code. Static analysis can also be used to extract call graphs from a code base. This gives the analyst a list of functions that can be invoked within the code.

This approach to malware analysis is not without problems. Usually the malicious program’s software code is not available for an analyst, only the binary. This reduces the number of static techniques that may be applied to build a signature. The analysis of binaries can be fraught with problems. These problems range from malware that requires system values that can not be statically determined, such as specific system date:time combinations, to packed and encrypted code segments that are not recognizable as such. Other works have even shown that the application of opaque constants can make a malicious binary virtually invulnerable to static analysis.
1.1.2 Dynamic Analysis

Dynamic analysis takes the opposite approach and focuses on methods which execute the malicious program. Most dynamic approaches have traditionally consisted of monitoring a computer’s file system, registry, network I/O and running processes. There are a number of tools which assist with this, including the most well known, Sys Internal’s Process Monitor, Process Explorer, TCPView, all of which support runtime monitoring. Active Debuggers like IDA Pro allow the details of encrypted samples to be analyzed.

The most commonly used dynamic analysis techniques rely on function call monitoring. A function is made up of code which performs a given task, such as calculating the average of an array of values. Functions allow for code re-use, and can make a code base easier to maintain. The most interesting part of functions is that they can abstract details about how a program is implemented. These abstractions lead to a semantically richer representation of what exactly a piece of software is intended to do.

To better understand this, it is believed that how a malicious binary accomplishes its goals is not as important as the actual goals. This is true when we consider that there are various ways to delete, move, and change data. Knowing the specifics of how a file is deleted within a specific malicious sample will only help in the development of a signature if the malware writer never creates a variant which utilizes other methods.

More often than not, dynamic malware analysis is being done in virtual machines (VM). This is the result of numerous reasons, including that a VM can be restored to a previous clean state, VMs can be paused for analysis, and the VMs filesystem can be inspected by the host operating system. However, there are problems that exist in the dynamic analysis of malware using VMs, including:

- Some malware can detect that it’s running in a virtualized environment
- Some malware can detect breakpoint insertion and pausing of the virtual environment
- Some malware are designed with polymorphic behavior, which changes based on how and when it is executed

Virtual Machine Introspection is the latest in a long line of approaches of dynamic malware analysis that is specifically aimed at solving some of the problems that plague traditional dynamic analysis. Virtual machine introspection is the process of monitoring everything about a virtual machine through interaction with the HyperVisor. Introspection provides additional visibility into each virtual machine. This can be useful for implementing security and analyzing running programs. Introspection can even provide functionality not possible in a dedicated operating system and hardware environment.

1.2 HyperVisor Introspection

The objective of Introspect is to provide an introspective hypervisor architecture and infrastructure to support advanced analysis techniques and introspective capability development. The Xen hypervisor provides virtualization of hardware resources to run multiple distinct guest computer systems. By residing architecturally below the target system, an introspective hypervisor can monitor or control a guest virtual machine without direct modification, cooperation, or detection. To date over 17 proof-of-concept tools have been developed that range from analysis of Windows system calls, registry access, network activity, ioctl communications, and open file handles. The Introspect framework can be used to overcome challenges in the applications areas of dynamic forensics, reverse engineering, malware analysis, and guest protection.

The hypervisor provides technology to analyze software behavior through virtual machines (VM) introspection. Introspect can be extended to monitor up to a hundred simultaneously executing VMs. Each of which will generate thousands of distinct software behaviors every second, on commodity hardware. The architecture employs a modified Xen hypervisor to collect data which analysts can use to extract and reconstruct software behaviour. Virtual machines introspection is the process of looking at the memory contents of a running virtual machine. By applying knowledge of the guest operating system, introspection can be used for a variety of applications, including reverse engineering malware, debugging software, and security the guest VM.

Currently, Introspect is built to utilize Xen as the underlying virtualization framework. Xen is a virtual-machine monitor (VMM), also known as a hypervisor, which is loaded directly from the boot loader when a machine is powered on. Xen is able to manage multiple virtual machines simultaneously, and launches a special “privileged” guest referred to as Dom0. Dom0 is a paravirtualized guest, this means that it is aware of the Xen hypervisor that it runs on and can interact with
it to manage system resources. From Dom0, users can launch additional, unprivileged VMs which are referred to as DomU. DomU virtual machines can either be paravirtualized (PV) guests requiring special support from the hypervisor, or a hardware virtual machines (HVM) guest, using special processor instructions to support virtualization. Both Intel and AMD provide their own extensions for this, however IntroVirt currently supports only Intel.

2. INTROSPECTIVE HYPERVISOR DESIGN

IntroVirt is currently built around three (3) major components: 1.) the hypervisor, 2.) libIntroVirt, and 3.) libWintroVirt. The IntroVirt architecture is shown in Figure 1.

2.1 IntroVirt Hypervisor Hooks

IntroVirt currently uses a modified version of the open source Xen hypervisor. The changes to the Xen hypervisor have been designed to be small and unobtrusive, and are essentially just minimal hooks inserted into the open source Xen hypervisor code. The IntroVirt team sought to limit the changes required within the Xen hypervisor due to security issues, maintenance, and the Xen open source license. By minimizing the changes to the open source Xen hypervisor the team is able to keep pace with the Xen open source tree with minimal overhead as well as maximize the amount of AIS code within the proprietary IntroVirt framework.

2.2 libIntroVirt

The libIntroVirt library provides the communication mechanisms for Dom0 to interact with the hypervisor. It is guest OS agnostic, merely allowing higher-level libraries to communicate with the hypervisor. It is important to note that libIntroVirt provides the abstraction layer between the IntroVirt tools, library stack, and the hypervisor. This design ensures that the IntroVirt framework is not overly tied to the Xen hypervisor, and minimal changes would be required to support additional hypervisors.

2.3 libWintroVirt

The libWintroVirt library relies on libIntroVirt, and provides introspection features specifically tailored for the Microsoft Windows operating system. It is able to parse several key Windows kernel objects, such as the OBJECT_ATTRIBUTES, EPROCESS, ETHREAD, and PE (Portable Executable) file format structures. Tools can be developed to perform introspection tasks using libWintroVirt, without requiring the developer to learn the low-level details of how Microsoft Windows is implemented.
2.4 libXtroVirt
The libXtroVirt component within the diagram represents the future development of operating system specific parsing libraries that are required to develop introspective capabilities for operating systems other than Microsoft Windows.

2.5 Application Layer
The application layer (represented by Tool X, Tool Y, and Tool Z in Figure 1) is the specific implementation of the introspective tools that are developed for using the IntroVirt architecture.

2.6 Implementation
The IntroVirt 2012 AIS, Inc. IRaD was focused on Microsoft Windows system call hooking, and system call return hooking. The IntroVirt framework has been developed and extensively tested on the Intel VMX platform with preliminary support and limited testing on the AMD SVM architecture.

2.7 libWintroVirt implementation
With knowledge of system call numbers for different versions of Windows, and the ability in libWintroVirt to parse kernel structures, tools can pull out specific information from the system call without the need for a low-level understanding of Windows internals. Tools can simply register a handler for system calls, and will be handed a Windows KPCR (Kernel Processor Control Region) object, the object contains information about the process making the call, such as process ID, thread ID, and the name of the process. It can then use a libWintroVirt-provided parser for the system call in question. For example, the ZwCreateFile class is able to retrieve the name of the file being opened from the corresponding Windows system call without the user knowing implementation details. Likewise, system call returns are provided via a callback method in the same manner.

3. RELATED WORK
We have presented IntroVirt, a system designed for virtual machine introspection. This work is not the only system to implement introspection. Garfinkel and Rosenblum discussed Introspection in their work A Virtual Machine Introspection Based Architecture for Intrusion Detection in which they presented an architecture for building a host-based intrusion detection system (IDS). This work specifically concentrated on three (3) components given by the VMM: 1.) isolation, 2.) inspection, and 3.) Interposition. Isolation provided by the use of virtualization, ensured that the VM could not be subverted in such a way as to access the IDS. Inspection capabilities were provided by the VMM. Access to CPU registers, memory and all I/O such as disk, was monitored through the VMM. Finally interpositioning was used by virtual machine operations to execute privileged instructions. Their IDS utilized the VMM’s ability to monitor code, which attempts to access a specific register. This method for host based IDS creation is highly effective because it utilizes a priori knowledge about system structure to determine software state, given hardware state measurements.

Since the aforementioned work was published, a number of others have followed suit including Lie and Litty’s work entitled Using Hypervisors to Secure Commodity Operating Systems which discusses the use of hypervisors to protect specific pieces of software against attacks. This work concentrated on using one trusted VM to monitor another VM’s actions. The authors presented two (2) systems: 1.) ProxOS, which provided better isolation for applications in such a way as to not constrain interaction with the guest OS, 2.) Manitou and Patagonix provided a means of introspection which paused executing code and compared it against a database of known binaries. Not unlike IntroVirt, this system was initially built on the DomU/Dom0 design that Xen provides.

As VM introspection becomes more main stream, formal methods have been developed. Pföh, Schneider and Eckert presented the first of such in their paper A Formal Model for Virtual Machine Introspection. Pföh, et. al. work included an exploration of challenges that needed addressing before introspection could be leveraged to secure VM applications. While their work focused on formalizing intrusion detection methods to describe how an attacker might enter the OS, the model can be expanded to discuss all facets of VM introspection. Pföh et. al. go on in their work entitled A Universal Semantic Bridge for Virtual Machine Introspection to discuss the semantic gap that exists when analyzing memory maps from guest VM’s. Memory maps need to be converted to associate operating system specific functions. However, doing so requires an expert system developer, to create these correlations. The work presented in this paper discusses InSight, a system which
uses existing source from the Linux operating system to bridge this memory map to function gap. While this may seem directly applicable to our own work on IntroVirt, it is not due to the non-live nature of the system which is being analyzed. Introvirt is currently being utilized in a number of different projects. As it relates to this work, Hall and Taylor have utilized Introvirt in A Framework for Network-Wide Semantic Event Correlation and Software Analysis in the Semantic Web, both of which use Introvirt for collecting system call events to populate OWL-based semantic models.22–24 These models are used to more accurately cluster malware samples based on their high level behaviors.

4. CONCLUSIONS AND FUTURE WORK

We have presented IntroVirt, an introspective hypervisor, with an emphasis on its use as a dynamic malware analysis system. IntroVirt is both an architecture and infrastructure that supports advanced analysis techniques for stealth-malware-analysis. This system allows for guest monitoring and interaction including the manipulation and blocking of system calls. While IntroVirt as presented in this work is for use in dynamic malware analysis, it has various other uses. The IntroVirt hypervisor analyzes software behavior using virtual machine introspection for. IntroVirt retrieves privileged information from the guest operating system in real time, which can be used for reverse engineering, malware analysis, software debugging and securing of guests. Currently IntroVirt employs a modified Xen hypervisor to collect data which analysts can use to extract and reconstruct software behavior from. Future work will emphasize the use of IntroVirt in the areas of: reverse engineering, forensics, asset protection, training and simulation.

ACKNOWLEDGMENTS

IntroVirt - Copyright ©2012 Assured Information Security, Inc. (AIS)

REFERENCES


