Beyond Ring-3: Fine Grained Application Sandboxing

Ravi Sahita (ravi.sahita@intel.com), Divya Kolar (divya.kolar@intel.com)
Communication Technology Lab.
Intel Corporation

Abstract

In the recent years the types of malware and level of their sophistication has increased dramatically [1]. In 2007, the number of computer viruses increased by 1 million and most of them were new attacks [2]. Unknown code downloaded and executed from the Internet can cause unrecoverable damage to the Operating System via privilege-escalation attacks. Malicious code can be unintentionally and transparently downloaded due to vulnerabilities in the browsers. To mitigate such attacks, researchers have designed a number of sandboxing techniques to isolate unknown system components to restrict their access to kernel components. In this paper, we describe a hardware-based fine-grained application sandboxing technique that allows for programmer-defined virtual privilege levels to be defined beyond the single user-space privilege level available in today's processors. We focus on allowing such privilege separation mechanisms to be applied in-place to the browser as it runs within a commodity operating system.

1. Introduction

Recent technology advancements have led to improvement to existing isolation and sandboxing software and tools. Software sandboxing techniques have proven to be useful to prevent the browser application from tampering with the OS kernel or stealing sensitive information. Attackers have been authoring viruses, worms and rootkits for monetary gain and installing them transparently on the end user platforms. For example, the Shadow walker Rootkit [3] can hide from anti-virus engines as it overwrites the OS default page fault handler thereby fooling software memory scanners. These kinds of low-level attacks pose a challenge to security researchers to prevent operating systems from being subverted.

Several isolation and sandboxing mechanisms have been proposed in the past [4-7]. Most of the previous work uses virtual machine isolation, or copy-on-write techniques. These approaches satisfy the need for coarse-grained application isolation but do not address access-control for sub-components that use OS services from within the applications running in the isolated environment. The sandboxing technique we describe in this paper is called 'Fine-Grain Application Sandboxing'. In our method, we propose hardware-rooted isolation of sub-components within an application. Such a separation model, allows programmer to be able to define access-control policies that will be enforced by hardware for as application sub-components interact with the host platform services. Our design goals are strong sub-application isolation, protection of the integrity of the kernel, with minimal system-resource impact for this access control mechanism.

In Section 2 of this paper we describe the threat model we mitigate with our sandboxing architecture. Section 3 describes our sandboxing reference architecture in detail. Section 4 contrasts our architecture with related work in this area. Section 5 describes initial experiments against low-level attacks launched via a web-browser. Section 6 summarizes our research direction.

2. Threat Model

In our threat model, we assert that a remote attacker can launch an attack on a platform via an application such as a browser by executing malicious code via explicit user choice or automatically over the network by malicious code downloaded from the web and executed in the context of a web-service that needs low-level system access to operate. We also assert that the attacker can compromise the privilege-level separation between the user applications and Operating System kernel and therefore install itself into the highest privilege level within the Operating System kernel. There are many documented methods to achieve this privilege escalation [8]. Consequently our threat model assumes that malicious code introduced via a web-service accessed by a user can achieve full control over the operation of the users' Operating System.

We focus on addressing two critical aspects of browser security:
1. Protecting the host system state from browser (or browser plug-in) originated attacks and,
2. Protecting the browser state from malicious browser plug-ins that may attack the host browser directly or via the OS kernel.
As an increasingly large number of browser plug-ins and widgets are authored, a framework that accommodates both of these requirements is critical to allow web-based software functionality to grow without seriously sacrificing user and system security.

3. Reference Architecture

We provide a very brief introduction to CPU support for virtualization before we describe our reference architecture for fine-grain application sandboxing. For further details refer to the Intel® Software Developers Manual [9]. Hardware Virtualization support, such as available on Intel® VT and AMD-V* processors, enables Virtual Machine Monitor (VMM) software to virtualize the CPU and system resources such as memory and devices. A VMM manages VMs by operating at the highest software privilege level (VMX-root mode). Control is transferred into the VMM for critical events as programmed by the VMM. A guest OS runs in VMX-non-root mode which ensures that critical OS operations cause a VMExit, - this allows the VMM to enforce isolation policies. Our reference architecture has been implemented on current Intel® VT-enabled platforms and uses the hardware memory management events to virtualize application page tables. Our system can further enforce memory access control on sub-components within an application thus restricting the attack surface available to sub-components and plug-ins. Considering the exponential increase in dynamically downloadable plug-ins for improved web experience and web service technologies, it is extremely important to ensure the integrity of the host application (in this case a browser) and the critical kernel components that the base platform consists of.

Our reference architecture consists of the following components that execute in the privileged context of the VMM, which virtualizes the guest OS memory management. The Integrity Manager component is responsible for measuring the integrity of applications or application plug-ins. This component measures the application at the lowest level in physical memory. The Memory Protection Module component maintains the active page tables that the guest OS executes via. This module separates memory page references measured by the Integrity Manager into a separate protected page table. The VMM thus access-controls access to the protected pages by way of injected page faults, and decides if the access should be allowed or disallowed. The protected page tables are managed in VMM protected memory so that malware in the guest OS cannot tamper with them. The Sandbox Manager directs the creation of another set of page tables, called the sandbox page tables, which is an additional set of page tables that further separates unknown application plug-ins from known plug-ins within a protected applications address space. This fundamental level of memory addressing separation allows the browser application to be extended safely without the risk of unknown plug-ins being able to access all the state of the browser application. For instance, a browser plug-in should not be able to access a protected portion of the browser heap area. The sandbox manager directs the creation of these page tables only in response to events received from a protected Sandbox Service which is described below.

The following components execute in the unprivileged guest OS. A Sandbox Service is a kernel service introduced into the guest OS. This component is first measured using the Integrity Manager and protected using the Memory Protection Module. This driver then interacts with the VMM Sandbox Manager to notify it of new programs getting loaded into the OS kernel address space or into a specific applications address space. The Sandbox Service can also apply policies on the OS or applications memory space to mark them restricted or available for loadable modules loaded into the OS or application memory. Examples of restricted memory spaces for an OS kernel may be kernel data structures kernel or system call tables. For an application address space, restricted spaces may be the heap for the application or other private data structures like encryption or signing keys. Sandboxed programs - These are unknown kernel components or applications which include unknown code in its address space. It is assumed that such programs got included in the control flow unintentionally or intentionally by malicious programs. These malicious components could be downloaded via the browser as a result of a cross-site attacks or any form of attack on the dynamic HTML content. We refer to these as “Sandboxed programs” since access control to restricted regions is enforced on these programs/memory regions. Figure 1 illustrates the layering of the components described. For a detailed discussion of the various components described in the architecture the reader is referred to our longer paper which describes our prior work on Virtualization-enabled Integrity Services [10].

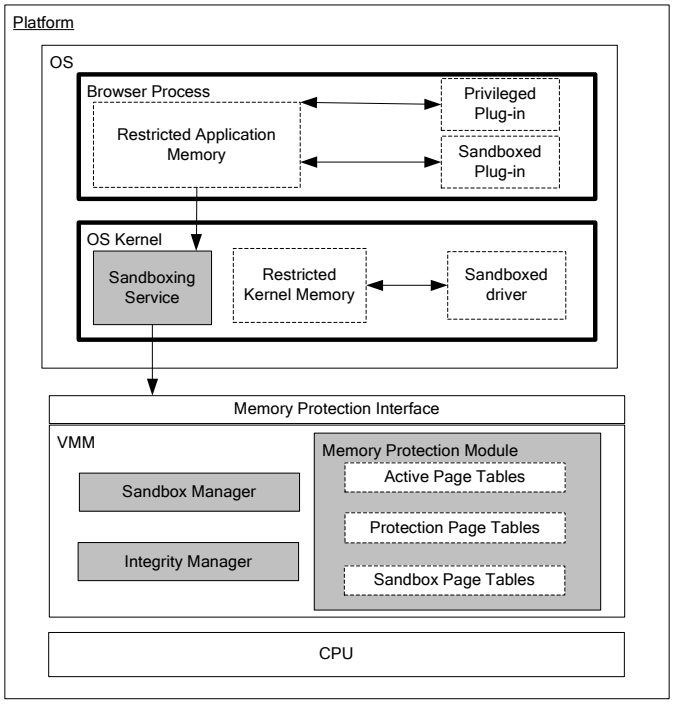

Figure 1: Reference Architecture for Fine-grain Application Sandboxing

The process by which an application can request differing levels of sandboxing as plug-ins are loaded into the process space is as follows. The Sandbox Service is first measured and protected. This process creates the protected page tables to protect the measured execution environment from the rest of the OS. The Sandbox Service can still communicate with the Operating System when protected. The Sandbox Service then registers expected measurements for applications that want to use sandboxing services with the VMM. When the application loads, the Sandbox Service specifies the name of the application that is executing and the address space context (control register CR3) for the application. The application can inform the VMM of the dynamic areas of application memory that it wants to restrict access to – any static memory information can also be contained in the applications measurement database. The Sandbox Service similarly can restrict memory for the OS kernel. When a new driver or application plug-in is loaded into the kernel or into the application address space, the protected sandboxing service communicates the linear address ranges where the plug-in will execute from and the CR3 to the hypervisor. The VMM Sandbox Manager verifies the plug-in integrity if required and creates a sandbox page table for the linear address range where the new driver/plug-in is loaded. Based on application policy the sandbox manager can now ensures that the sandboxed plug-in has none or restricted access to the restricted linear range for the application the plug-in is executing within. Any accesses from the sandboxed plug-ins and/or drivers to the restricted linear address ranges are disallowed and captured before the access goes through. Note that using this approach, two plug-ins in the same application address space may have entirely different sandboxing restrictions. Violation events are reported by the hypervisor to the protected sandboxing driver from where the user can be informed of malicious plug-ins.

4. Experimental results

Based on our threat model, we assumed the following pre-conditions while developing our proof of concept. The host has most common web browsers installed with the ability to execute dynamic HTTP content. The browser is assumed to be able to execute in the administrator account of the host. The user is allowed to download browser components on-demand to enhance the browsing experience.

In our research prototype, we have a client PC connecting to a web server to view web pages hosted on the remote server. We designed the HTML page the user is trying to view in such a way that it installs an ActiveX control on the user's platform. The ActiveX control writes three files on the client's hard disk in the system folder. 1. A system file (driver) which when executed will modify the OS system call table. 2. A browser helper object with the OnClick() event implemented to load the system service

installed in step 1, and 3. A script which registers the browser helper object on the system so that the helper is loaded with every invocation of the browser.

When the user visits the described malicious website, the user effectively installs a browser helper object and a kernel service in kernel context which is capable of modifying the OS system call table. We conducted experiments without enabling the application sandboxing framework. The attack executed transparently and was successfully able to hook into the OS activity on the running client PC, thus installing a basic rootkit. We then enabled our application sandboxing framework and retried the attack on the client PC. In this scenario, our OS Sandbox Service was first protected using the VM Memory Protection Module, and notified the VMM Sandbox Manager of the restricted memory areas of the OS kernel. The protected Sandbox Service then notified the VMM of any new components being loaded into the kernel or application space – any unknown kernel components were sandboxed from accessing the restricted OS area before the kernel invokes them. As a consequence the attack service failed to modify the OS system call table to point to the malicious code. Quantifying the performance overhead of our application sandboxing is future work; however the user experience was not affected when our sandboxing framework was active.

5. Related Work

A lot of work has been done on software sandboxing including Java* [11] virtual machines, and FreeBSD* Jail [12]. Most software methods of application sandboxing view the entire application being sandboxed as a monolithic module and do not allow access-control of system resources between application sub-components. We focus on related work that has been targeted towards any of these two aspects of browser security – 1. Protecting the host system state from browser (or browser plug-in) originated attacks and 2. Protecting the browser state from malicious browser plug-ins that may attack the host browser directly or via the OS kernel.

The Tahoma [6] web browsing system uses the Xen virtual machine monitor for isolating browser instances and its associated policies. In the Tahoma system, a manifest defines the browser policies such as URLs the browser can access and other web application characteristics. A trusted computing base called "Browser Operating System (BOS)" executes the management functions for Xen. The Tahoma BOS has a kernel which manages browser instances and storage for the system. A network proxy enforces network access policies for web applications based on the policies in the manifest. In our approach, the browser application is not moved to a separate virtual machine; instead we enforce memory page level access-control in-place via a hardware-enforced shadow page table. Thus in our system, we execute different browser instances in the same virtual machine at the same time still ensuring that two browsers cannot tamper with each others memory contents thus achieving equivalent memory isolation for the browser instances as Tahoma. Some of the key differences with Tahoma are that our approach does not require graphics or device virtualization – additionally our goal is to ensure sandboxing of plug-ins within the browser to allow for differentiated and controlled access to the platform from such plug-ins.

Feather-weight Virtual Machine (FVM) [13] is an OS-level software virtualization architecture. FVM uses namespace virtualization to rename system resources at the OS system call interface. One of the differences between the virtualization layer in FVM and our approach is that in our approach the virtualization is at a memory access level versus in FVM where the virtualization layer uses OS resource renaming and copy-on-write technique to isolate VMs. The OS based virtualization approaches are themselves susceptible to privileged malicious code that can tamper or unhook the virtualization hooks within the operating system. Such tampering affects all the sandboxes running on the host. In our approach, the hypervisor can also sandbox any unauthorized code detected on the host machine such that malware will not be able to tamper with sandboxing services.

Ozone HIPS [4] is a self protection tool designed to protect the operating system by disallowing tampering with system resources and/or disallow loading of unauthorized code. Ozone uses two protection layers called memory protection layer to guard against attacks that hijack execution by corrupting memory and a process protection layer that sandboxes process to continuously monitor their access to the important system processes. It uses address space randomization to prevent attacks that depend on static address exploits from being successful. In the process protection layer, Ozone enforces access control by sandboxing kernel service calls. Sandboxes in Ozone are described by a policy text file stored on the host. Our approach is similar to the Ozone approach but differs in the architecture for memory protection – we use a higher privilege hardware mode for memory protection.

Our approach also does not require address space randomization for the protected processes since memory-based attacks are mitigated by strict separation of pages referenced from the process page tables.

6. Summary

Unknown stealth code downloaded and executed from the Internet can cause unrecoverable damage to the Operating System, applications and users. We describe a framework for hardware-rooted fine-grained application sandboxing technique that allows for programmer-defined policies to be applied to securely sandbox application plug-ins. In our framework, we focus on addressing both aspects of browser security: protecting the host system state from browser (or browser plug-in) originated attacks and, protecting the browser state from malicious plug-ins. Current software sandboxing schemes cannot protect the OS or privileged services from other malicious privileged services. Our architecture also allows programmers to enforce policies for runtime access-control of application plug-ins, for example, to protect users' data when it is accessed by an application plug-in.

7. References

[1] P. Bustamante, "Rootkits in the mist," Panda Research Blog, 2007.

[2] Symantec*, "Symantec* internet security threat report: Trends for April 2008," 2008.

[3] Eweek*, "Shadow Walker Pushes Envelope for Stealth Rootkits.," in *EWeek* Magazine.*, 2005.

[4] E. Tsyrklevich, " Ozone HIPS: Unbreakable Windows.," in *Blackhat 2005*, 2005.

[5] F. Guo, Y. Yu, and T.-c. Chiueh, "Automated and Safe Vulnerability Assessment," in *ACSAC 2005* Washington DC USA.: IEEE Computer Society, 2005.

[6] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A Safety-Oriented Platform for Web Applications," *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06),* pp. 350-364, 2006.

[7] S. Miwa, T. Miyachi, M. Eto, M. Yoshizumi, and Y. Shinoda, "Design Issues of an Isolated Sandbox Used to Analyze Malwares," in *Lecture Notes in Computer Science: Advances in Information and Computer Security*, Heidelberg, 2007.

[8] K. Kaslin, "Kernel Malware: The attack from within," in *AVAR* New Zealand, 2006.

[9] Intel Corp., *IA-32 Intel(r) Architecture Software Developers Manual*.

[10] R. Sahita, U. Savagaonkar, P. Dewan, and D. Durham, "Mitigating the Lying-Endpoint Problem in Virtualized Network Access Frameworks," in *Lecture Notes in Computer Science: Virtualization of Networks and Services*, 2007, pp. 135-146.

[11] Sun Microsystems*. "Java Security Model," 1997.

[12] The FreeBSD* Documentation Project., "FreeBSD Architecture Handbook," 2000.

[13] Y. Yu, F. Guo, S. Nanda, L.-c. Lam, and T.-c. Chiueh, "A Feather-weight Virtual Machine for Windows* Applications," in *Second International Conference on Virtual Execution Environments*, 2006.