

# ***ECE 1776: Progress Report***

## **Patagonix: Dynamically Neutralizing Malware with a Hypervisor**

H. Andrés Lagar-Cavilla - andreslc@cs.toronto.edu

Lionel Litty - llitty@cs.toronto.edu

### **Introduction**

In most operating systems widely in use today, it is possible for malicious software (malware) that has gained administrator privileges on the system to arbitrarily modify the kernel. Recently, Windows-based malware has been increasingly able to wedge itself deeper into the system by taking advantage of this failure of the operating system (OS) to protect itself. This results in the malware being both increasingly stealthy and hard to uninstall, because code running in user space cannot rely on the kernel's integrity anymore. In the extreme, pernicious malware could make it close to impossible to repair the system, and a full reinstall of the system may be necessary.

In this project, we explore the feasibility of neutralizing malware by leveraging a hypervisor, a thin layer of software running below the operating system. Ideally, by leveraging knowledge of the architected constraints of the platform, a hypervisor might be able to neutralize malware using only minimal knowledge of the inner workings of the monitored OS. Our project focuses on implementing this idea on the Xen hypervisor using hardware support for virtualization (Intel Virtualization Technology VT-x) to run unmodified host operating systems. Such an approach would then enable one to dynamically clean systems of malware that has taken steps to hide itself/prevent its removal from user space without even requiring a reboot of the system.

In previous work, we have implemented this idea on a paravirtualized version of Linux[1]. The paravirtualized kernel makes hypervisor calls to modify the content of the page tables used by the hardware to translate virtual addresses to physical addresses. When a Hardware Virtual Machine (HVM) is used instead of a paravirtualized one, Xen maintains shadow page tables that reflect the content of dummy page tables that are modified by the kernel. This means that the techniques used in [1] need to be modified to fit HVMs. In this document, we describe the changes that will need to be made to the shadow page table management code in Xen.

### **Shadow page table code changes**

To disallow malware execution, we interpose on every attempt to execute code and validate this code against a whitelist of trusted software. The main challenge is to keep track of the permission bits the (potentially compromised) kernel assigns to its pages in the shadow page tables, and replace them by permission bits that will make the MMU trigger page faults on any attempt to execute unchecked software. On a page fault, control is handed to the hypervisor, at which point we use content-based hashing techniques to validate the code. To achieve this, we need to add an extra data structure (the frame table) and modify code paths in the shadow page table and page fault handling subsystems of the Xen hypervisor.

### **Frame table**

The purpose of the frame table is to handle the possibility that a single physical page may have several virtual mappings. Since permission bits are associated with the virtual mappings of a page and since we want to enforce the writable XOR executable principle for physical pages, we need to apply this principle across virtual mappings of that physical page. To this end, we could perform a full examination of the

page tables every time a new mapping is created, or we could implement a reverse frame table. Instead, we implement a simpler frame table. We surmise that this is the most efficient mechanism for this situation and will test this assumption through benchmarking.

For each physical memory page, we keep track of the use the HVM is doing of that page by assigning the page a type. This type may be either writable, executable or read-only (which is the default type and the type each physical page initially has). A page can only have one type, but additional read-only mappings of a page can exist even if the page type is writable or executable. For example, there may be three virtual mappings of a page that are executable and two other mappings that are read-only. In no case can there be simultaneous writable and executable mappings. If such a situation arises, all executable mappings are converted to writable or vice versa, depending on the situation.

In addition to the type, we keep track of the number of mappings that exist with that type for the writable and executable types. New read-only mappings do not impact the system and therefore no updates to the frame table are required when they occur.

### **Shadow page table updates**

When the OS updates an entry in the dummy page tables, corresponding updates need to be made to the real page tables used by the hardware. In order to do so, the type and type count of the page needs to be updated in the frame table. The following situations may arise:

\*New mapping with existing type: If the type is writable or executable, the type count is incremented.

\*Executable or writable mapping becomes read-only or is removed: the type count is decremented. If the type count hits 0, the type is changed to read-only.

\*New executable mapping of read-only page: a hash of the content of the page is taken. If the hash appears on a list that corresponds to code that is allowed to execute, the type of the page becomes executable and the executable permission bit is set for that page table entry. If the dummy mapping of the page is also writable, the writable permission bit is cleared for the real mapping. If the hash is not on the list, the executable permission bit is cleared and the writable permission bit is left untouched.

\*New non-executable but writable mapping of read-only page: The type is changed to writable and the count is increased.

\*Executable mapping becomes writable: The type count is decremented. If the type count is now 0, the type is changed to writable, the type count is set to 1 and corresponding updates are made to the real page table entry. If the type count is not 0, other executable mappings of that page are found by searching the page tables and these mappings are changed to be non-executable. This situation is expected to be rare.

\*Writable mapping becomes executable: A hash of the content of the page is taken and checked against the list. Assuming the hash appears on the list, the type count is decremented. If the type count is now 0, the type is changed to executable, the type count is set to 1 and corresponding updates are made to the real page table entry. If the type count is not 0, other writable mappings of that page are found by searching the page tables and these mappings are changed to be non-writable. This situation is expected to be rare.

### **Page fault handling**

When an action of the HVM results in a page fault, Xen handles the resulting hardware exception. The code to handle page faults that are the result of an instruction fetch on a non-executable page, or of an attempt to write to a non-writable page, needs to be modified.

If the fault is the result of trying to execute code from a non-executable page, and if the permission bits in the dummy page table indicate that the OS believes this page should be executable, then a hash of the content of the page is taken, and we check if this hash appears on the list. If it does, we update the shadow page table entry for that page to make the page executable but non-writable. The previous section describes the actions that may need to be taken to perform this update.

If the fault is the result of trying to write to a non-writable mapping, and if the permission bits in the dummy page table indicate that the OS believes this page should be writable, we set the writable bit in the page table entry and clear the executable bit, if it is set. Actions that may need to be taken to perform this update are described in the previous section.

## **Conclusion**

At this stage of our project, we have a clear idea of what changes need to be made to the Xen code to support Patagonix. With the imminent release of Xen 3.0.3, we have started examining the Xen code more closely and are deciding where to implement our modifications. The next steps will be to actually modify the code and test Patagonix. If all goes well, we are hoping to examine at least one piece of malware running on Windows and we will try to disable it.

## **Bibliography**

[1]Lionel Litty and David Lie. Manitou: A Layer-Below Approach to Fighting Malware. In Workshop on Architectural and System Support for Improving Software Dependability (ASID), October 2006