MoRE Shadow Walker: TLB-splitting on Modern x86

Jacob Torrey
@JacobTorrey
Assured Information Security
torreyj@ainfosec.com

ABSTRACT

MoRE, or Measurement of Running Executables, was a DARPA Cyber Fast Track effort to study the feasibility of utilizing x86 translation look-aside buffer (TLB) splitting techniques for realizing periodic measurements of running and dynamically changing applications. It built upon PaX, which used TLB splitting to emulate the no-execute bit and Shadow Walker, a memory hiding rootkit; both designed for earlier processor architectures. MoRE and MoRE Shadow Walker are a defensive TLB splitting system and a prototype memory hiding rootkit for the current Intel i-series processors respectively – demonstrating the evolution of the x86 architecture and how its complexity allows software to effect the apparent hardware architecture.

Keywords

Hypervisor, Code Measurement, Real-time Analysis, Code Isolation, Harvard Architecture, Split-TLB, Critical Software

1. INTRODUCTION

MoRE examined the feasibility of utilizing TLB splitting as a mechanism for periodic measurement of dynamically changing binaries. The effort created a proof-of-concept system to split the TLB for target applications, allowing dynamic applications to be measured and can detect code corruption. Documented in the following sections is relevant background (Section 2) on the technologies used, the design and implementation process of the MoRE system (section 3) and the results of the effort (section 4).

2. BACKGROUND

2.1 Problem

Currently, on a running computer system, there is no method of ensuring the trust in executing applications. When an application is loaded, it is possible to generate a hash of the code before it is executed as the data portions of the executable are initialized to a known value. Once the application has been running, and the data has been changed, a hash value becomes meaningless. To combat this, most user-land applications are compiled and linked in such a fashion that the data and code reside on different pages in memory (called 'static' throughout this abstract). The portable executable (PE) format used by the Windows OS for applications subscribes to this paradigm, having a section ('.text') which is read-only for code and another section ('.data') for data. When the PE is loaded into memory, the attributes of the pages are such that a write to the .text section will generate a fault and the kernel will abort the process to prevent corruption.

Other, more privileged executables such as boot-loaders, OS kernels, hypervisors, and SMI handlers are usually not compiled in such a way, and have data and code intermixed (called 'dynamic' throughout this paper, as the executable changes). Once code is allowed to execute for a period of time, measuring to ensure code trustworthiness is not a viable option. The MoRE effort solved this problem: enabling the real-time verification of system code state of dynamic executables (e.g. OS kernel or boot-loader) to detect compromise or code-injection attacks.

2.2 Portable Executable and PE Relocations

Modern Windows OS applications and kernel drivers are binaries in a format known as portable executables (PE). This format provides the application or driver loader with the requisite information to load it into memory, adjust any addresses which may have been changed and link in any shared libraries before execution. It divides the binary into a number of different sections, some for code (.text), some for data (.data) and other informational sections. Of these extra sections, the most important to this effort is the .reloc section, which lists the number and location of addresses which must be updated at load time if the compile-time address was different. Shared libraries and system call functions can vary from one computer (or even session) to another, especially with security technologies such as address-space layout randomization. Most compilers use the same address when initially linking the program, and as such, any addresses to global variables are referenced in the .reloc section.

2.3 Paging & Translation Look-aside Buffer

Modern CPUs provide the ability to provide each process with a unique view of memory. This feature, known as virtual memory or paging eases the OS's task of isolating different applications and providing each application with a consistent view of memory. When a virtual memory address is accessed by an application, the CPU uses a number of data structures to automatically translate the virtual address into a physical address. This process, outlined in Figure 1, uses the CPU's CR3 register to find a page directory and optionally a page table, which holds the physical address. In most modern OSes, each process is given its own set of page translation structures to map the 4GB flat memory view provided to the system's physical memory.

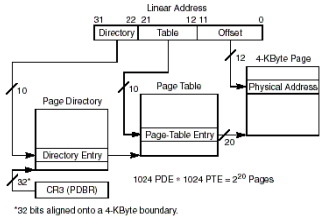

Figure 1: Paging translation on x86[1]

The translation look-aside buffer, or TLB, acts as a cache for these paging translations. Due to relatively high memory latency compared to cache-access speed, a page translation lookup is expensive in terms of time; therefore, these operations are optimized by caching the virtual to physical mappings in the TLB. While logically, the TLB stores the translations for all accessed addresses in the same area, the physical implementation splits the TLB (Figure 2) into two: one for instruction addresses (I-TLB)

[1] http://viralpatel.net/taj/tutorial/image/paging.gif

and one for data addresses (D-TLB). This implementation detail is important, as it allows the TLB to point to different addresses for instruction fetches as compared to data accesses.

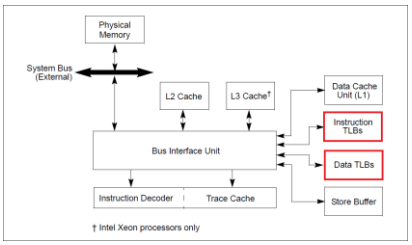

Figure 2: Core 2/Previous CPU Architecture (TLBs Highlighted)[2]

There has been some past work which took advantage of this split-TLB nature for malicious purposes in the past, namely PaX [1] and the Shadow Walker root-kit [2] and work done to prevent self-verifying applications from detecting corruption [3]. Shadow Walker is designed to hide the presence of a kernel driver through TLB splitting. When the code is accessed as data, such as by an anti-virus tool, Shadow Walker points the D-TLB towards the un-modified kernel region. When the target section is executed, the I-TLB is filled with the address of the malicious driver's code, allowing it to run as expected. A similar technique is used in [3] to prevent self-hashing applications from detecting the malicious modification of the application.

While paging can be used to simply isolate processes, most OSes use it to manage memory use by paging-out memory when not in use. The OS then invalidates the translation(s) which point to that physical memory region by altering the page structures to note that region as paged out. When a process tries to access one of those translations, the CPU causes an exception known as a page fault. The OS can then copy the data from the disk back into a free physical page and update the paging structures with the new physical address.

The page fault handler must also be written in such a way to ensure the in-memory paging structures and the TLB remain synchronized. The x86 architecture provides the INVLPG instruction which invalidates an entry in the TLB, forcing the CPU to re-walk the paging structures next time that address is requested. Additionally, when the CR3 register is changed, all TLB entries are invalidated unless they are specifically marked as global. Global pages are most commonly used for shared libraries and OS kernel functions exported to user-land applications and thus benefit from remaining in the TLB.

2.4 VMX & EPT

A growing trend in IT is the use of virtual machines (VMs), which enables datacenter consolidation. A hypervisor or virtual machine monitory (VMM), allows multiple OSes to run simultaneously on the same physical system, each isolated from the others. While some hypervisors require changes to the guest OS (para-virtualization) to function properly, many leverage newer CPU extensions to allow an unmodified OS to run with minor interactions from the hypervisor. These extensions, known as virtual machine extensions, or VMX, improve performance by empowering the CPU and chipset to perform more of the isolation and VM memory management in hardware. VMX allows the hypervisor to set a number of different exit conditions for each guest VM which, when met, trigger a VM EXIT returning control to the hypervisor for processing.

In the latest version of VMX, Intel and AMD have released the extended page table (EPT) or rapid virtualization indexing (RVI) technologies, respectively. These allow the hypervisor to take even less of a role in the memory management and isolation of each guest by providing another layer of paging structures to translate the physical addresses which the VM OS believes to be the physical address (guest physical address) to the machine physical address. The CPU can automatically translate these in a similar fashion to conventional paging and provide a VM EXIT which is analogous with the page fault to the hypervisor. These translations can be stored in the TLB, and tagged with each guest's VM process ID (VPID) so they need not be flushed on VM context switch.

These aforementioned technologies significantly aid the hypervisor in running multiple VMs in an isolated fashion with relatively minor performance impacts. The MoRE VMX implementation extensively leverages these technologies to perform the TLB splitting for user-land Windows applications.

3. DESIGN

3.1 Design Goals & Testing

MoRE was designed to allow periodic measurement of running executables and ensure the measurements remain meaningful as the data in dynamic applications changes. In order to accomplish this goal, MoRE worked to take advantage of TLB splitting for defensive purposes – create a copy of the executable and redirect data accesses to the data copy while instruction fetches are routed to the static version (Figure 3). This approach was taken to determine whether using this architectural method would be feasible and what the caveats and performance impacts would be [4].

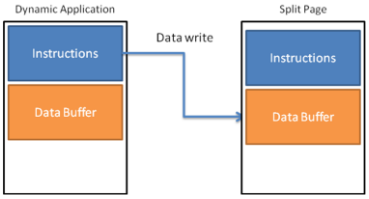

Figure 3: MoRE Split TLB Goal

To empirically test and simulate a dynamic application, a kernel-mode page was split: when the page was accessed as data, it would display the opcodes for instructions to halt the system, when executed it returned without system interruption. A test suite of user-land applications was also created to aide in performance testing. The test application's PE files were modified in such a way that the .text section was writable and the tests would use the .text section as a memory buffer to simulate a dynamic application. While the test suite is non-similar to the practical use cases, it provides a useful simulation framework for testing and performance analysis.

3.2 PFH Implementation

The initial implementation was inspired by Shadow Walker, in that it hooks the page fault IDT entry and uses its privilege position in the system to split the TLB for certain memory pages. Figure 4 provides a flowchart of the PFH semantics which works as a filter on page faults. If the page fault was caused by MoRE, it

[2] Image from Intel Software Developer Manual 3A

will perform the requisite task and return to the target application; otherwise the PFH will forward the exception to the OS's handler.

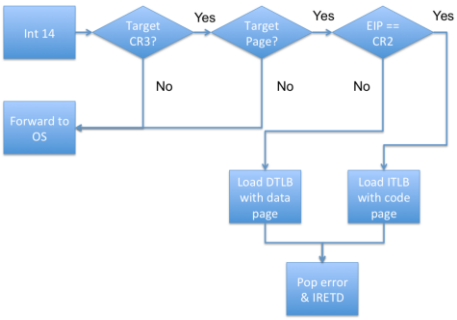

Figure 4: PFH Flowchart

More concretely, when a target page is split it results in an immutable executable page (IEP) and a writable data page (WDP). The paging structures for the target page are marked as paged-out, while leaving them resident in memory. When the target page is accessed, the CPU will generate a page fault, transferring execution control to the MoRE PFH filter. The MoRE PFH then checks the CR2 value and stores the faulting address to determine if the fault was caused by the paging structure manipulation or if it was a genuine page fault. If a genuine page fault, MoRE forwards the fault to the OS for handling. If the faulting address is the target page, the MoRE PFH can determine whether the fault was an instruction fetch or data access by comparing the exception return address (EIP) with the faulting address or checking the error code provided by the x86 architecture [3].

The MoRE PFH then adjusts the paging structures to show the target page as paged in (present in memory) and adjusts the physical address to point to either the physical address of the IEP or WDP. Once the structure has been updated, the PFH loads that translation into the TLB, either by simply accessing the first byte in the page for data accesses, or by temporarily overwriting the first byte of the page with the RET (return) instruction (0xC3) and CALLing that page before restoring the correct first byte. These actions load the TLB with the correct translation; the MoRE PFH then alters the paging structures to once again show the target page as not present and returns control to the faulting application, without invalidating the TLB entry (desynchronizing the paging structures from the TLB).

The Windows 7 (the chosen OS for the MoRE prototype) memory manager monitors each user-land application's memory usage, caching possibly needed pages in memory before they're requested (pre-loading) and paging-out pages which are not likely to be used in the near future. While the system is running, the memory manager checks each process's working set to see if it can be trimmed or should have pages cached. If a discrepancy between the paging structures and Windows' internal structures is found, the kernel bug checks (triggering a blue screen of death) the system to prevent memory corruption – preventing the PFH from TLB splitting a user-land process. It is important to emphasize that if an operating system architect was designing a new system, or a Windows kernel developer modified this behavior, the existing hardware would support TLB splitting of applications.

3.2.1 Intel Nehalem Architecture Differences

During testing of initial prototypes, it was discovered that the TLB splitting did not function properly on the newer Intel Core-i series processors. Further research determined that Intel had changed their TLB architecture and added a shared TLB (S-TLB) which functioned as a shared L2 cache for the data and instruction TLBs. When either of the I-TLB or D-TLB is full, the least-recently-used translation is evicted and replaced with the new translation. In this new architecture, the evicted translation is moved to the S-TLB in case it will be needed again shortly, where it can rapidly be replaced without re-walking the paging structures. While this improves performance, the shared nature of the S-TLB violates the separation MoRE relies on, discarding the older of the similar translations. To support this new architecture, a VMX hypervisor was implemented.

3.2.2 VMX Design

After further research into the possible solutions for overcoming the S-TLB issues, it was discovered that the optimal solution would be to leverage VMX functionality to both bypass the Window memory manager and S-TLB problem. Due to the fact that a hypervisor is more privileged than the OS, the VMX memory manager is able to manipulate memory without the OS knowing. EPT provided the simplest method to do so and had the least performance impact on the system. EPT also provides the hypervisor's paging structures granting more granular access controls to each page, permitting read-only, execute-only and read/write paging permissions. It was assumed that if the TLB, which can also cache EPT translations, was primed with split entries, each with different permissions, the TLB would not merge them in the S-TLB, violating the security of EPT permissions. To prevent the TLB entries from being invalidated, it was essential to support VPID in the hypervisor.

The VMX hypervisor developed for MoRE is a Windows driver which could load a hypervisor into operation and put the running Windows into a VM without interrupting the system or performing any device emulation. The first steps were to add VPID and EPT support to the hypervisor, and the EPT paging structures would point the guest physical addresses to the identical machine physical addresses (an identity map). This would permit the OS to manage memory as if the hypervisor were not present, and allow MoRE to mark certain physical addresses as non-present in EPT without the Windows memory manager noticing.

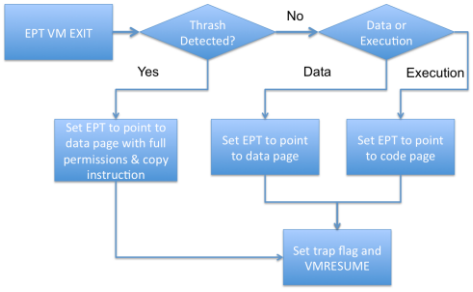

Figure 5: EPT VM EXIT Flowchart

3.2.2.1 MoRE VMX Functionality

With the inclusion of EPT and VPID support into the VMX hypervisor, a similar procedure as in the PFH (Figure 4) could be implemented. The paging out process was done in the EPT structures and the MoRE filter was moved to the VM EXIT

handler for an EPT fault. The major modification that was required was the fact the handler could not prime the TLB itself; with VPID, the TLB entries are tagged with (and only accessible to) the ID of the priming VM, or 0 in the case of the hypervisor. To overcome this hurdle, the EPT handler modifies the paging structures for the EPT fault, and sets the guest trap flag in the EFLAGS, which causes the CPU to trap after a single instruction. The hypervisor is then configured to VM EXIT on the trap exception. MoRE implements a trap flag handler in the VMX hypervisor which disables the trap flag and resets the EPT paging structures to non-present, leaving the VPID tagged TLB primed, but will trap to the EPT handler if an access of a different type occurs. A graphical flowchart of this process is shown in Figure 5 and Figure 6.

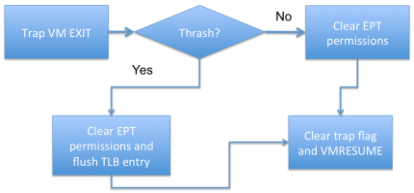

Figure 6: Trap VM EXIT Flowchart

3.2.2.2 Windows PE COW

Once the MoRE PFH functionality was ported to the hypervisor and tested to support TLB splitting of a user-land application, it was noticed that when a dynamic application ran, the physical pages of the data copy of the PE would change. Through further research, it was discovered that Windows marks all code pages (even if they are marked as writable) as read-only and when modifications are detected, performs a copy-on-write (COW) operation. This optimization allows Windows to run multiple instances of the same application without wasting memory on duplicate, rarely changing code pages.

To detect this remapping of the application's pages, the MoRE hypervisor would walk the OS's paging structures each time the CR3 register was changed (each process switch), and if the physical addresses were different for the target application, it would update its list of pages to split. Due to this feature of Windows, the read-only executable copy is kept unchanged, and the data copy which was made is removed and replaced with the Windows COW version.

3.2.2.3 Thrashing Detection and Workarounds

During the testing process of the VMX handler, it was discovered that the S-TLB would not permit two entries for the same address loaded for the same instruction. In other words, when instructions were within the same (4KB) page as the data being accessed, the VM EXIT handler would replace the TLB entry for data with the instruction address and vice versa, causing an infinite loop (thrashing back and forth). To maintain the security guarantees proscribed by this effort, a workaround was developed.

When a thrash was detected (two sequential EPT VM EXITs without a trap VM EXIT), the EPT handler would set the translation to point to the data page, and allow it to be executed and read/write accessed. It would also copy the instruction to be executed to the data page from the execution copy to prevent modification. When the EPT handler returned, the instruction would be executed from the data copy, then trapping back to the trap handler, which would remove the permissions and disable the trap flag – in essence, single-stepping through these 'thrash points'.

4. RESULTS

This section describes the results and software generated by the MoRE effort. The main goal of the MoRE effort was to create a prototype research platform for determining the feasibility of measuring dynamically changing applications through TLB splitting. This technology could be used for a variety of cases, but would at the very least be able to satisfy the following requirements: separate executing code from data, be able to be statically measured and periodically measure applications.

All these requirements were met. Both the MoRE PFH and VMX hypervisor are able to utilize TLB splitting to separate instruction fetches from data accesses, and periodically measure (10 Hz) the target PE application. We ensured that the Windows driver implementation of both the PFH and the VMX hypervisor did not contain any dynamic modules and thus could be measured to ensure that the MoRE code had not been compromised. A second Windows driver was implemented (drivermeasure.sys) to measure the MoRE driver and display its measurement to ensure it remained static. The performance impact of the MoRE hypervisor on the test suite was ~2%. This test suite included both a CLI and GUI version of the following applications, all of which could be configured to be dynamic or static and mixed or isolated to test thrash handling:

- Pi Calculation – Power series estimation of pi
- Wasteful Sort – Random swap & check
- Coin Flipping – Random coin flipping
- Cycle Timers – Setting a timer and calculating instructions per cycle

5. MORE SHADOW WALKER

It is the author's belief that technology itself is not inherently good, evil, defensive or offensive. TLB-splitting neatly shows this in the progression from PaX (defensive), Shadow Walker (offensive) and finally to MoRE (defensive). To further highlight this point, the author will once again swing the technology's application to the offensive side with MoRE Shadow Walker, a memory hiding VMX root-kit that can operate on Intel Nehelam and newer CPUs with the S-TLB.

MoRE Shadow Walker (MSW) is built upon the same code base as the defensive MoRE VMM, but instead of ensuring the code pages are unchanged from the loaded application, it can allow an attacker to load a different page to be executed. This lets a malicious adversary insert malicious code into a kernel code page without alerting Microsoft PatchGuard (PG) as PG will read the unchanged *data* page containing the unchanged kernel instructions.

It is worthwhile to mention the motivation for utilizing a VMX hypervisor to hide a kernel implant as the hypervisor is already operating from a stealthy and highly-privileged position in the system. MSW-hidden kernel implants allow for a greater granularity of introspection into the OS's operation and at less performance and code size costs when compared to a full introspective hypervisor (e.g. IntroVirt™[3]). Additionally, there is

[3] http://www.ainfosec.com/introvirt/

a wealth of existing kernel implants and root-kits that exist for the Windows kernel that can be quickly packaged with MSW to bypass PG.

The code for both MoRE and MSW will be released as open source and available[4] for further research and study.

6. REFERENCES

[1] PaX Team, "PAGEEXEC," 15 March 2003. [Online]. Available: https://pax.grsecurity.net/docs/pageexec.txt. [Accessed 17 February 2014].

[2] S. Sparks and J. Butler, "Shadow Walker: Raising the Bar for Rootkit Detection," in *Blackhat Japan*, 2005.

[3] P. van Oorschot, A. Somayaji and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," in *IEEE TDSC*, 2005.

[4] Intel Corporation. “Intel Software Developer Manuals”. 2014

[4] https://github.com/ainfosec/more