TOCTOU, Traps, and Trusted Computing*

Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean W. Smith

Dartmouth College, Hanover, New Hampshire

Abstract. The security of the standard TCG architecture depends on whether the values in the PCRs match the actual platform configuration. However, this design admits potential for *time-of-check time-of-use* vulnerabilities: a PCR reflects the state of code and data when it was measured, not when the TPM uses a credential or signs an attestation based on that measurement. We demonstrate how an attacker with sufficient privileges can compromise the integrity of a TPM-protected system by modifying critical loaded code and static data after measurement has taken place. To solve this problem, we explore using the MMU and the TPM in concert to provide a *memory event trapping framework*, in which trap handlers perform TPM operations to enforce a security policy. Our framework proposal includes modifying the MMU to support selective memory immutability and generate higher granularity memory access traps. To substantiate our ideas, we designed and implemented a software prototype system employing the monitoring capabilities of the Xen virtual machine monitor.

1 Introduction

The *Trusted Computing Group* (TCG) [1] works toward developing and advancing open standards for trusted computing across platforms of multiple types. Their main goals are to increase the trust level of a system by allowing it to be remotely verifiable and to aid users in protecting their sensitive information, such as passwords and keys, from compromise. The core component of the proposal is the *Trusted Platform Module* (TPM), commonly a chip mounted on the motherboard of a computer. A TPM provides internal storage space for storing cryptographic keys and other security critical information. It provides cryptographic functions for encryption/decryption, signing/verifying as well as hardware-based random number generation. TPM functionalities can be used to attest to the configuration of the underlying computing platform, as well as to seal and bind data to a specific platform configuration. In the last few years, major vendors of computer systems have been shipping machines that have included TPMs, with associated BIOS support.

* This work was supported in part by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, and the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. The views and conclusions do not necessarily represent those of the sponsors.

P. Lipp, A.-R. Sadeghi, and K.-M. Koch (Eds.): TRUST 2008, LNCS 4968, pp. 14–32, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The key to TCG-based security is: *A TPM is used to provide a range of hardware-based security features to programs that know how to use them.* TPMs provide a hardware-based *root of trust* that can be extended to include associated software in a chain of trust. Each link in this chain of trust extends its trust to the subsequent one. It should be noted that the semantics of this extension for each link of the chain are determined by the programmers (including the BIOS programmer). More specifically, the programmer defines the conditions applying to the system's state S_{i+1} that are checkable in the state S_i under which the transition $S_i \to S_{i+1}$ is deemed to preserve trust. These conditions strongly rely on our understanding of the relationship between the software active in S_i and S_{i+1}. For example, a developer may trust a process in S_{i+1} that is created from an ELF file after verifying in S_i that either the entire file or some of its sections such as code and data hash to a known good value. Implicit in this decision is the assumption that the hash measurement is enough to guarantee the trustworthy behavior of the process.

In this work, we explore an additional set of TPM-based security architecture features that programmers can take advantage of to secure data that they perceive as sensitive and enforce a new class of policies to ensure their software's trustworthiness.

In particular, we note that the current TCG architecture only provides load-time guarantees. Integrity measurements are taken just before the software is loaded into memory, and it is assumed that the loaded in-memory software remains unchanged. However, this is not necessarily true—an adversary can exploit the difference between when software is measured and when it is actually used, to induce run-time vulnerabilities. This is an instance of the *time-of-check time-of-use* (TOCTOU) class of attacks. In its current implementation, the TPM holds only static measurements and so these malicious changes will not be reflected in its state. Code or data that is correct at the time of hashing may be modified by the time of its use in a number of ways, e.g., by malicious input. Change-after-hashing is a considerable threat to securing elements in the TCG architecture.

This paper explores this TOCTOU problem. Section 2 places it in the context of the TCG architecture, Section 3 demonstrates this vulnerability. Section 4 explores a solution space: making the TPM aware when memory that has been measured at load-time is being changed in malicious ways at run-time. Section 4 then speculates on a hardware-based solution, but also presents a software proof-of-concept demonstration using Xen's memory access trapping capabilities. Section 5 evaluates the security and performance of our solution. Section 6 explores related work. Section 7 concludes with some avenues for future work.

2 TOCTOU Issues in the TCG Architecture Perspective

Generally, the focus of current trusted platform development efforts has been on mechanisms for establishing the chain of trust, platform state measurement, protection of secrets and data, and remote attestation of platform state. Clearly,

without cheap and ubiquitous implementations of these basic mechanisms, commodity trusted platforms would not happen.

However, as Proudler remarks in [2], "Next-generation trusted platforms should be able to enforce policies". With the introduction of policies, the trusted system engineer's focus must necessarily extend from the above mechanisms to *events* that are classified and controlled by the policy.

Accordingly, it becomes natural to formulate the concept of what constitutes the measured state at the upper levels of the trust chain in terms of events subject to the policy: a sequence of policy-allowed events starting from a measured "good" state should only lead to another "good" state.

In fact, the concept of the underlying system of controlled events is central to the policy: whereas *policy goals* are defined in terms of the system's states, events determine the design of the underlying OS mechanisms and the policy language. For instance, in case of SELinux MAC policies, events are privileged operations realized as system calls hooked by the Linux Security Modules (LSM) framework.

One can argue (see, e.g., [3]) that LSM's implicit definition of the class of controlled events has substantially influenced both the scope and language of SELinux policies, making certain useful security goals such as, e.g., "trusted path," hard to express, and leading to a variety of alternative more manageable methods being adopted by practitioners. SELinux's example shows that defining a manageable set of controlled events is crucial to engineering the policy.

Another necessity faced by a policy designer is a workable definition of measured state. Quoting Proudler [2] again, "We know of no practical way for a machine to distinguish arbitrary software other than to measure it (create a digest of the software using a hash algorithm) and hence we associate secrets and private data with software measurements."

However, the semantics of what constitutes measured, trusted and private data in each case is inevitably left to the program's developers. Thus the particulars of what [2] refers to as *soft policy* are left to the developer and ultimately relies on his classification and annotation of different kinds of code constituting the program and data handled by the program with respect to their importance for policy goals. We refer to such annotation that allows the programmer to distinguish different kinds of data (such as public vs. private vs. secret, in the above-mentioned paper's classification) as "secure programming primitives" and note that introduction of new kinds of such primitives (e.g., read-only vs. read-write data, daemon privilege separation, trusted execution path patches, etc.) usually lead to improving the overall state of application security. In each case, it was up to the programmer to design the software to take advantage of the new security features; as we remarked above, TPMs are no different.

Recent alternative attestation schemes (e.g., [4]) move from measurement of a binary to an attestation that the measured binary is trustworthy, but still are based on static measurement. However, trustworthiness does not depend merely on what the binary looks like when it is loaded, but also on what it does (and what happens to it) when it executes.

It is from these angles that we approach the current TCG specification and propose to leverage it to control a set of memory-related events in which an application programmer can express a meaningful security policy.

We note that our proposal does not change the "passive" character of the TCG architecture: its scope is still restricted to providing security features to programs that are written to take advantage of them. Our contribution lies in introducing a framework of events and their respective handlers that invoke the TPM functionality. To base a policy of this framework, the programmer, as before, needs to separate program code and data into a number of classes based on their implications for the trustworthiness of the program as a whole, and specify the trust semantics for each class.

In this paper, we report our initial exploration, focusing on policies that enforce selective immutability of code and selected data in a program. The policy can be specified as immutability requirements for specific sections of the program's executable file and bundled with it, e.g., included as a special section in the ELF format.

Why extend security policy to memory events? Many architectures include OS support for trapping certain kinds of memory events. For instance, the ELF format supports annotation of program segments to be loaded into memory as writable or read-only, as well as executable or not intended for execution. Most of these annotations are produced automatically by the compiler–linker–loader chain that also takes care of aligning the differently annotated segments on page boundaries (since the x86 hardware supports these annotations by translating them to the appropriate PDE and PTE protection bits, which apply at page level).

We note that programmer's choice in this annotation has been traditionally kept to a minimum and not always exactly matched the programmer's intentions: e.g., constant data would be placed in the `.rodata` section, which would then be mapped, together with the `.text` section and other code sections to the loadable segment designated as non-writable *and* executable.

We further note that these automatic mappings of code and data objects[1] to different loadable segments are typically meant to apply at load time and persist throughout the lifetime of the process. More precisely, once an object has been assigned to one of these sections based on the programmer's guessed intentions, the programmer can no longer easily change its memory protections without reallocating it entirely.

We also note that "service" sections of process image such as `.got`, `.dtors`, etc., involved in such extremely security-sensitive operations as dynamic linking (and thus specifically targeted by many exploitation techniques[2]) do not, as a rule, change their protections even after all the relevant operations are completed, and

[1] We speak of "code objects" to distinguish between, e.g., the `.text`, `.init/.fini`, and `.plt` sections that all contain code dedicated to different purposes, similar to the more obvious distinction between the `.data`, `.ctors/.dtors`, and `.got` data objects sections.

[2] E.g., `http://www.phrack.com/issues.html?issue=59&id=9`, `http://www.security-express.com/archives/bugtraq/2000-12/0146.html`, etc.

write access to them is no longer necessary (but can still be exploited by both unauthorized code and authorized code called in an unanticipated manner).

Yet programmers may well conceive of changing roles of public, private or secret data throughout the different phases or changing circumstances of their programs' execution, and may want, *as a matter of security policy goals*, to change the respective protections on these objects in memory. Indeed, who else other than programmer would better understand these transitions in data semantics?

We contrast this situation with that in UNIX daemon programming before the wide adoption of privilege drop and privilege separation techniques. Giving the programmers tools to modify access privileges of a process according to the different phases of its execution resulted in a significant improvement of daemons' trustworthiness, eliminating whole classes of attacks. We argue that a similar approach applied to the sensitive code and data objects would likewise benefit the programmers who take advantage of it, and formulate their security goals in terms of memory events.

Secure programming primitives controlling memory events would provide assurance that the program could be trusted to trap on those events that the programmer knows to be unacceptable in any given phase, resulting in more secure programs. For example, programmers using a custom linking mechanism (such as that used by Firefox extensions) will be able to ensure that their program be relinked only under well-defined circumstances, defeating shellcode/rootkit attempts to insert hooks using the same interfaces.

Thus we envision a programming framework that provides a higher granularity of MMU traps caused by memory access events, and explicit access policies expressed in terms of such events. A modified MMU accommodating such a policy would provide additional settable bits that could be set and cleared to cause a trap on writes, providing a way to "seal" memory objects after they have been fully constructed, and to trap into an appropriate handler on events that could violate the "seal". The handler would then analyze the event and enforce the programmer's intentions for the data objects, now explicitly expressed as a policy (just as privilege drops expressed an implicitly assumed correct daemon behavior).

We discuss an interesting practical approach to achieving higher memory trapping granularity in Section 6—an example from a different research domain where the same need to understand and profile programs' memory access behaviors posed the same granularity problem as we and other security researches face.

In the following discussion we assume feasibility of higher granularity memory event trapping framework as described above, and explore the opportunities that such MMU modifications and respective trap handlers would provide for TOCTOU-preventing security policies when used in combination with the TCG architecture.

3 Vulnerability Demonstration

We'll begin by considering *attestation* of a platform's *software*. The TPM's PCRs measure the software; the TPM signs this configuration (using a protected key).

Then the user passes this configuration on to a third party which presumably uses this configuration information to decide whether or not to allow the measured platform to run a piece of software or join a network session.

Notice that a key component of this system is that if measures a binary at *load* time. If a binary changes after it has been loaded, the configuration of the system will not match the attested configuration. If a program can change this segment of RAM *after* it has been loaded, than its behavior can be modified, even though the measurement in the TPM shows that the program is in its original state. Such a change in the program's behavior can be accomplished in different ways, such as by an exploit supplied by an adversary in crafted input to the process itself, or though manipulation of the process's address space from another process through a lapse of kernel security. We note that in this paper we do not consider TOCTOU on hardware elements or carried out via hardware elements.

We must consider the potential of an attacker achieving malicious changes to the code or data of the running trusted process created from a measured executable at the end of the TPM-based chain of trust. We note that these scenarios are no less relevant for TCG compliant trusted systems, since "trusted" does not mean "trustworthy." Commodity operating systems have a long history of not being trustworthy, even if users choose to trust them. (Indeed, the term *trusted computing base* arose not because one *should* trust it, but rather because one had no choice *but* to trust it.)

We take the worst case: a kernel vulnerability that allows the attacker limited manipulation of x86 Page Tables (PT).[3] In Linux (as in other operating systems), these structures also contain information on the permissions needed by a process to read or modify the particular memory segment, interpreted by the MMU on every memory access while converting virtual addresses to physical ones.

To demonstrate[4] TOCTOU, we wrote a kernel module that replaced data at a given virtual address in a process image with the given process ID. In order to show that simply monitoring a process's page tables was not a sufficient defense against this attack, our module did not modify the target process's data structures at all, but rather changed other process' pages containing one `.text` segment or PTs. Section 5 provides the details of several such attacks.

We have constructed a small example login program to demonstrate this attack. After we have loaded and measured it, we used our module to overwrite the opcode 0x74 of the critical `je` instruction in the password check routine of the running process with 0x75 (`jne`), remapping PT entry to point to the target page, and resulting in login with any wrong password. When the module is unloaded, process image is restored to its original state, so that looks pristine to future TPM-based or other measurements.

[3] Several Linux kernel vulnerabilities allowed attackers to bypass memory mapping and IPC restrictions are summarized, e.g., in [5].

[4] For our initial demonstration, we considered an IBM NetVista PC equipped with a Atmel TPM v1.1b running the Trusted GRUB boot-loader and the 2.6.15.6 Linux kernel that included the statically compiled TPM driver. We later repeated the attack on PCs equipped with STMicro v1.2 TPMs.

Thus applications that use the TPM's ability to *seal* a credential against specific PCR values can be subverted in what was measured in the PCRs changes; applications that measure data and configuration files, as well as software, can also be subverted. Consider, for example, the open source LiveCD *Certification Authority* package [6] that uses a TPM to hold the CA's private key and to add assurance that the key would only be used when the system was correctly configured as the CA—by wrapping the private key to specified values in a specified subset of the PCRs. The TPM decrypts and uses that key only when the PCRs have those values. If a user has means to modify arbitrary regions of memory, they can render the measurements of the TPM useless, unless the TPM somehow keeps continuous measurements of the loaded program's memory. Because of TOCTOU, any hole in the OS breaks the trust.

4 Solution and Prototype

To address this problem, we need the TPM to "notice" when measured memory changes. As our proof-of-concept showed, the TPM needs to worry not just about writes to the virtual address and address space in question, but also about writes to any address that might end up changing the memory mapped to the measured region. Thus, we need to connect the TPM with memory management, so that the MMU would trap on memory operations that can affect TPM-measured memory regions, and inform the TPM of them via the corresponding trap handler.

To evaluate the feasibility and effectiveness of this idea, we need to actually try it—but experimental modifications to modern CPUs can be a large task. So instead, we build a software proof-of-concept demonstration.

4.1 Components

Since we needed a lightweight way to experiment with virtual changes to machines, we decided to start with the *Xen* virtual machine monitor [7], which allows for the simultaneous execution of multiple guest operating systems on the same physical hardware.

Xen. Xen is being used in this project not for its virtualization features, but as a layer that runs directly below the operating system—similar to the placement of the hardware layer in a non-virtualized environment. Its placement helps us study possible hardware features. In a Xen based system, all memory updates trap into the thin hypervisor layer—making it easy to monitor and keep tabs on changing memory. Redesigning the MMU hardware is tricky, so we do not want to attempt that until we were certain that the end goal was useful. A potentially better alternative to using Xen would have been to use an open-source x86 emulator (such as Bochs [8] or QEMU [9]). However, as of their current implementation, none of these emulators have support for emulating a TPM. Also, the only currently existing software-based TPM emulator [10] does not integrate with any of these. Integrating them would be a major task in itself.

Virtual TPMs. For our prototype we will be using the unprivileged Domain-1 as our test system. Unprivileged VMs cannot access the system's hardware TPM, and so, to provide Domain-1 with TPM access, we need to make use of virtual TPMs (vTPM) [11].

4.2 Design Choices

We considered two ways to use XEN in our implementation.

- First, the strategic placement of the thin Xen hypervisor layer between the machine's hardware and the operating system could be seen as a way to prototype changes that could be made in hardware (i.e. in the MMU). With this approach, the purpose of Xen would be to solely demonstrate a proposed hardware change, and would not be intended to be integrated into the *TCG Software Stack* [5] (TSS). Xen's role would be that of a "transparent" layer, manifesting features that would ideally be present in hardware. *Effectively, we use Xen to emulate a hardware trap framework for intercepting memory events of interest to our policy.*
- Alternatively, Xen could be used with the purpose of incorporating it into the TSS. The trusted boot sequence would now include the measurement of the Xen hypervisor executable, the Domain-0 Kernel and applications running in Domain-0, subsequent to the system being booted by a trusted boot-loader. In this model, our *Trusted Computing Base* (TCB) will be extended all the way up to the hosting virtual machine environment. The TCG trust management architecture is currently defined only up to the bootstrap loader; in this alternative approach, we would need to extend the chain of trust up to applications running in Domain-0.

Several recent projects (see Section 6) are exploring using hypervisors for integrity protection. However, as the hypervisor layer is not currently part of the TCG trust management architecture, incorporating it into the TSS will necessitate a revision of the TCG specification. Consequently, we went with the first approach.

We also considered two ways to hook memory updates to the TPM. In the **dynamic TPM** approach, we would update the TPM's state every time the measured memory of an application is changed. At all times then, the TPM's state will reflect the current memory configuration of a particular application, and of the system as a whole. This would allow a remote verifier to be aware of the current state of the application in memory, and to make trust judgments based on these presently stored PCR values.

When the hypervisor detects a write to the monitored area of an application's memory, it would invoke a re-measurement of the application in memory. The re-measurement would involve calculating a SHA1 hash of the critical area of the binary in memory (as opposed to the initial measurement stored in the PCR, which was of the binary image on disk). This re-measured value would

[5] The TCG Software Stack is the software supporting the platform's TPM.

be extended to the TPM. In this case, monitoring of memory writes would be enabled for the entire lifetime of an application, as the TPM state would need to be updated each time the application's measured memory changed.

In the **tamper-indicating TPM** approach, we would update the TPM's state only the first time that the measured memory of an application is changed. This would allow a remote verifier to easily recognize that the state of the application in memory has changed, and hence detect tampering. When the hypervisor detects the first write to a critical area of an application's memory, it would *not* invoke a re-measurement of the application; instead, would merely extend the TPM with a random value. In this case, monitoring of memory writes could be turned off after the first update to the TPM, as that update would be sufficient to indicate tampering. Monitoring subsequent writes (tampering) will not provide any further benefit. This strategy will not have as much of a negative impact on performance as the first approach.

For performance reasons, we chose the second approach.

4.3 Implementation

Our prototype implementation consists of three primary components: the instrumented Linux Kernel for reporting, the modified Xen hypervisor for monitoring, and the invalidation in the TPM.

Reporting. We instrumented the paravirtualized Kernel of the Domain under test (in our prototype – Domain-1) to allow it to report to the hypervisor the PTEs, and physical frames that these PTEs map to, of the memory to be monitored, as shown in Figure 1 (a).

To enable this feature, we added two new hypercalls. `HYPERVISOR_report_ptes` reports to the hypervisor a list of PTEs that map the memory that needs to be monitored. The PTEs are essentially the entries that map the `.text` section of the binary into memory. `HYPERVISOR_report_frames` reports to the hypervisor a

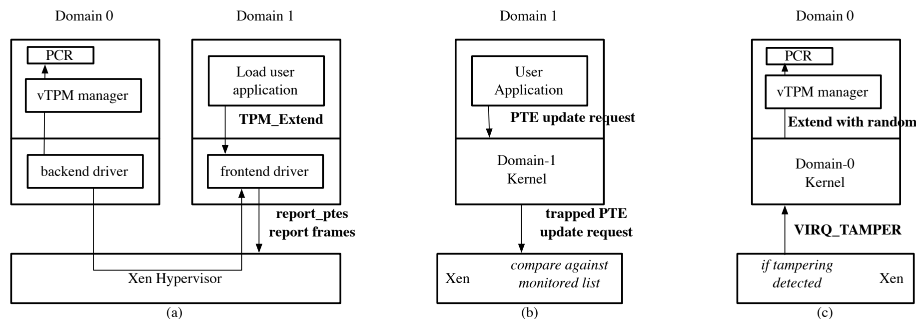

Fig. 1. (a) reporting to the hypervisor the PTEs and frames to be monitored; (b) monitoring the reported PTEs and frames for update; (c) updating PCR in vTPM of Domain-1 on tamper detection

list of physical memory addresses that need to be monitored. The addresses are the physical base addresses of each frame that contain memory that needs to be monitored.

These hypercalls make use of a new function that we have added to the kernel, `virt_to_phys()`, which walks a process's page tables in software to translate virtual addresses to physical addresses. We pass to this function the start and end virtual addresses of the `.text` section of the binary to be monitored. Using the fact that there are 4096 bytes of data on each page[6], it calculates the number of virtual pages spanned by the address range passed to it. It then accesses an address on each page of the range, so as to have it mapped into memory. This step is required to overcome potential problems due to *demand loading.*[7] At this point, the whole of the `.text` section of the binary is mapped into memory. This step however, has performance implications in that it slows down application start-up; unfortunately, dealing with this requires modification of the existing dynamic linker-loader to support deferred loading of trusted libraries. Although complex, it appears to be a promising direction of future research.

The function then walks the page tables of the process to translate the virtual addresses to physical addresses (physical base address) of each frame in the range. A data structure containing a list of these addresses is returned to the calling function.

Also, on program exit (normal or abnormal), we need to have the monitored PTES and frame addresses removed from the monitored list. To do this, we instrumented the Kernel's `do_exit` function to invoke a new hypercall.

`HYPERVISOR_report_exit` reports to the hypervisor when an application that is being monitored exits. The hypervisor's monitoring code then deletes the relevant entries from its monitored lists.

Monitoring. Once the required PTEs and frame addresses are passed down to Xen, it will monitor them to detect any modifications made to them, as shown in Figure 1.

Writes to these physical memory addresses, or updates to these PTEs to make them map to a different subset of memory pages or make them into writable mappings, will be treated as tampering. The reason for this is that since we are monitoring the read-only code section of an application, neither of the above updates are legitimately required.

The most convenient and reliable method of detecting these types of updates is to 'hook' into Xen's page table updating code. As mentioned earlier, all page table updates in a Xen system go through the hypervisor. This enables us to put in code that can track specific addresses and PTEs.

The default mode of page table updates on our experimental setup is the Writable Page Table mode. In this mode, writes to page table pages are trapped and emulated by the hypervisor, using the `ptwr_emulated_update()` function. Amongst other parameters, this function receives the address of the PTE that

[6] Our experimental system has a 4Kb page size.

[7] Demand loading is a lazy loading technique, where only accessed pages are loaded into memory.

needs to be updated and the new value to be written into it. After doing a few sanity checks, it invokes Xen's `update_l1e()` function to do the actual update.

We instrumented `update_l1e()` to detect tampering. Amongst other parameters, this function receives the old PTE value and the new PTE value that it needs to be updated to. To detect tampering, we perform the following checks:

- **For PTEs:** we check to see if the *old* PTE value passed in is part of our monitored list. If it is, it means that a 'trusted PTE' is being updated to either point to a different set of frames, or to make it writable. The alternate set of frames are considered as potentially malicious frames, and the updated writable permission leaves the corresponding trusted memory open for overwriting with malicious code.
- **For frames:** We first check to see if the *new* PTE value passed in has its writable bit set. If it does, we calculate the physical address of the frame it points to. We then inspect if this physical address is part of our monitored list. If it is, it means that a 'trusted frame' is being mapped writable by this new PTE. The writable mapping, created by this new PTE is interpreted as a means to overwrite the 'trusted frame' with potentially malicious code.

Once the tampering is detected in the hypervisor layer, we need to be able to indicate this fact to Domain-0. We do this by creating a new virtual interrupt, `VIRQ_TAMPER`, that a guest OS may receive from Xen. `VIRQ_TAMPER`, is a global virtual *Interrupt Request* (IRQ), that can be allocated once per guest, and is used in our prototype to indicate tampering with trusted memory.

Invalidating. Once tampering of trusted memory is detected in the hypervisor layer, the Domain under test needs to have its integrity measurements updated. This is done by way of updating the Domain's platform configuration in its virtual TPM, as shown in Figure 1.

Our intention is to have the hardware (MMU) cause this update by generating an trap, which would invoke an appropriate trap handler. The latter, having verified that the memory event is indeed relevant to our policy goals, will in turn perform the TPM operation.

Considering that, in our prototype, the hypervisor together with Domain-0 are playing the role of the hardware, we need to have either of them perform the update action. However, as there are no device drivers present in the hypervisor layer, the hypervisor is unable to interface with the virtual TPM of Domain-1, and so this task is redirected to the privileged Domain-0.

The hypervisor will indicate tampering to Domain-0 by sending a specific virtual interrupt (`VIRQ_TAMPER`) to it. A Linux Kernel Module in Domain-0 will receive this interrupt, and will proceed to extend the concerned PCR in the virtual TPM of Domain-1 with a random value.

We have to make use of the virtual TPM Manager (`vtpm_managerd`) to talk to the virtual TPM of Domain-1. In its current implementation, the virtual TPM manager only delivers TPM commands from unprivileged Domains to the

software TPM. Domain-0 is not allowed[8] to directly interface with the software TPM. However, for our prototype, we need Domain-0 to have this ability, and so we have to mislead the virtual TPM Manager into thinking that the TPM commands from Domain-0 are actually originating from Domain-1.

In Domain-0, we construct the required TPM I/O buffers and command sequences required for a `TPM_Extend` to a PCR of Domain-1. As described earlier, there is a unique instance number associated with each vTPM. To enable Domain-0 to access the vTPM instance of Domain-1, we prepend the above TPM command packets with the instance number associated with Domain-1. This effectively help us forge packets from Domain-1.

5 Evaluation

Our prototype on x86, runs on a Xen 3.0.3 virtual machine-based system. Xen's privileged and unprivileged domains run Linux Kernel 2.6.16.29. Our evaluation hardware consists of a 2 GHz Pentium processor with 1.5 GB of RAM. Virtual machines were allocated 128 MB of RAM in this environment. Our machine has an Atmel TPM 1.2.

We implemented three attack scenarios subverting measured memory by exploiting the previously mentioned TOCTOU vulnerability. These attacks, seek to change the `.text` section of a loaded binary. The `.text` section is mapped read-only into memory, and so, is conventionally considered safe from tampering.

Scenario 1. The attacker overwrites the trusted code of a victim process by creating writable page mappings to the victim process's trusted frames from another process, as shown in Figure 2 (a).

We carried out this attack by modifying[9] a PTE in our malicious process to map to a physical frame in RAM that the victim process's trusted code was currently mapped to. We modified the PTE to hold the frame address of the victim process page that we wanted to overwrite. The PTE that we chose to update already had its writable bit set, so we did not need to update the permission bits. Using this illegitimate mapping we were able to overwrite a part of the trusted frame with arbitrary data.

It is interesting to note that this attack was possible without having to tamper with any of the victim process's data structures.

Our prototype detects that a writable mapping is being created to a subset of the physical frames that it is monitoring, and randomizes the relevant PCR to indicate tampering.

Scenario 2. The attacker modifies the trusted code of a victim process by updating the mappings of its `.text` section to point to rogue frames in RAM, as shown in Figure 2 (b).

[8] Domain-0 is only allowed to access the actual hardware TPM or the software TPM, but not the vTPM instances of other unprivileged domains.

[9] The attack could also be carried out by creating a new PTE that maps to the victim process's frames.

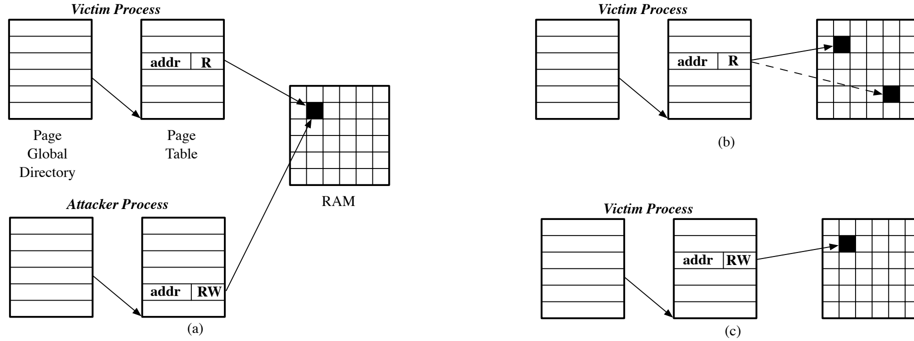

Fig. 2. (a) attacker manipulates PTE(s) of his process to map to trusted frames of victim process, and overwrites memory in RAM; (b) attacker manipulates PTE (address portion) of victim process to map to rogue frames in RAM; (c) attacker manipulates PTE (permission bits) of victim process to make frames writable, and overwrites memory in RAM.

We carried out this attack by using our malicious process to update the address portion of a PTE in the victim process that was mapping its code section. The updated address in the PTE mapped to rogue physical frames in RAM that were part of our malicious process. Due to these updated mappings, the victim process's trusted code was now substituted with the content of our rogue frame.

Our prototype detects that a subset of its monitored PTEs are being updated to point to different portions of RAM, and randomizes the relevant PCR to indicate tampering.

Scenario 3. The attacker overwrites the trusted code of a victim process by updating the permission bits of its `.text` section to make them writable, as shown in Figure 2 (c).

We carried out this attack by using our malicious process to update the permission bits of a PTE in the victim process that was mapping its code section. We updated the permission bits to set the writable bit making the corresponding mapped frame writable. We used this writable mapping to modify the trusted code in the victim process with arbitrary data.

Our prototype detects that a subset of its monitored PTEs are being updated to make them writable, and randomizes the relevant PCR to indicate tampering.

Limitations. It should be noted that the system of events we intend to capture and, therefore, the properties that we can enforce with it, deals at this point exclusively with memory accesses to code and data objects that can be distinguished by the linker and loader based on annotation in the executable's binary file format. As usual, the choice of events that we can trap to interpose our policy checks, limits our model of the system's trust-related states and transitions between those states, in particular, of transitions that bring the system into an untrusted state. In other words, the choice of a trappable event system essentially

determines the kinds of exploitation scenarios that the policy based on it can and cannot stop.

As the analysis above demonstrates, our choice defeats a range of classic scenarios. It does not, however, prevent other types of attacks that do not depend on modifying protected code and data objects.

For example, our event system does not equip us for dealing with exploits that modify the control flow of an unchanged binary by providing non-executable crafted input data[10], for the essential reason that "bad" transitions in the state of software associated with these exploits are hard to express in terms of these events. The same goes for cross-layer and cross-interface input-scrubbing application vulnerabilities, such as various forms of SQL or shell command injection, where malicious commands are passed to a more privileged and trusted (and therefore less constrained) back-end. Obviously, such vulnerabilities should be mitigated by a different complementary set of secure programming primitives.

One case where a secure programming primitive based on our event system may help is that of protecting writable data known to the programmer to persist unchanged after a certain well-known phase of normal program execution. In particular, the programmer can choose to place such data in a special ELF loadable segment[11] that can be sealed after the completion of that execution phase. This, in fact, is a direct analogue to UNIX daemon privilege separation, which provides the programmer with the means to effectively disallow privileged operations that he knows to be no longer needed.

As described, our system of events also does not prevent return-to-library[12] or return-to-PLT[13] attacks in which no executable code is introduced (but, rather, existing code is used with crafted function activation frames placed on the stack to effect a series of standard library calls to do the attacker's work).

However, in this case our event system is by design more closely aligned with the actual events and transitions of interest: library calls and PLT stub invocations can be easily distinguished from other programmatic events based on the information contained in the binary format (such as that in the `.dynamic` section tree and symbol table entries). Accordingly, they can be trapped as cross-segment memory accesses, with minimal additional MMU support. This, in turn, enables policy enforcement based on intercepting such events and disallowing all but those permitted by the policy.

Although a detailed description of such an event system and policy mechanism and its comparison with other proposed return-to-library and return-to-special-

[10] E.g., `http://phrack.org/issues.html?issue=60&id=10` for exploitation of integer overflows, `http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-iss-sourceaudit.ppt` for a range of other exploitable conditions.

[11] In particular, the GNU tool chain offers a compiler extension to map variables to ELF sections. See, e.g., `http://www.ddj.com/cpp/184401956`.

[12] E.g., `http://www.milw0rm.com/papers/31`, `http://phrack.org/issues.html?issue=56&id=5`

[13] E.g., `http://phrack.org/show.php?p=58&a=4`, `http://www.phrack.org/issues.html?issue=59&id=9`. Note, in particular, the use of the dynamic linker to defeat load address randomization.

ELF-section countermeasures is beyond the scope of this paper, we intend to continue the study of the underlying trapping framework in another publication.

Performance. Although the primary goal of our software prototype is to investigate the future hardware features needed to provide the proposed secure programming primitives, we also measured varios performance overheads of the prototype itself, to quantify some of the design choice trade-offs discussed in Section 4.2. Our results, found in [12], show that the actual overhead of our prototype system is almost negligible, making it a very usable and deployable.

6 Related Work

Attacks. Kursawe, et. al [13] look at a number of passive attacks against TPMs by monitoring signals across the bus that the TPM resides on in their test machine and hint at more active attacks. Sadeghi, et. al [14] also discuss testing TPMs for specification compliance.

Bernhard Kauer [15] demonstrated how to fool a TPM into thinking the whole system has been rebooted. Sparks also documents this and provides a YouTube video [16]. Rumors exist that other attacks are coming [17]. Sparks also presents evidence [18] that the CRT-based RSA operations in TPM may be susceptible to Boneh-Brumley timing attacks [19].

Attestation. IBM designed and implemented a TPM-based *Integrity Measurement Architecture* (IMA) to measure the integrity of a Linux system. Their implementation [20] was able to extend the TCG trust measurement architecture from the BIOS all the way up into the application layer. Integrity measurements are taken as soon as executable content is loaded into the system, but before it is executed. An ordered list of measurements is maintained within the kernel, and the TPM is used to protect the integrity of this list. Remote parties can verify what software stack is loaded by viewing the list, and using the TPM state to ensure that the list has not been tampered with.

The *Bear/Enforcer* [21,22] project from Dartmouth College developed a *Linux Security Module* (LSM) to help improve integrity of a Linux system. This LSM calculates the hash of each protected file as it is opened, and compares it to a previously stored value. If a file is found to be modified, Enforcer does some combination of the following: denies access to the file, writes an entry in the system log, panics the system or locks the TCG hardware.

Sadeghi et al. (e.g., [4]) developed a *property-based attestation* extension to the TCG architecture that allows binary measurements to be mapped to properties the relying party cares about, and these properties to be reported instead. Haldar et al. [23] propose an approach based on programming language semantics.

Copilot [24] is a run-time kernel integrity monitor that uses a separate bus-mastering PCI add-in card to make checks on system memory. The Copilot monitor routinely recomputes hashes of the kernel's text, modules, and other critical data structures, and compares them against known good values to detect for any corruption.

BIND [25] is a service that performs fine-grained attestation for establishing a trusted environment for distributed systems. Rather than attesting to the entire contents of memory, BIND attests only to a critical piece of code that is about to execute. It narrows the gap between time-of-attestation and time-of-use, by measuring code immediately before it is executed, and protects the execution of the attested code by using a sand-boxing mechanism. It also binds the code attestation with the data that it produces. It requires programmer annotations, and runs within a Secure Kernel that is available in the new *LaGrande Technology* (LT)-style CPUs.

Additionally, as discussed back in Section 4.2, several recent projects use a secure hypervisor to extend trust to systems' runtime. *Overshadow* [26] and *SecVisor* [27] use the hypervisor's extra level of memory indirection/virtualization for protecting the runtime integrity of code and data. We were also made aware of the HP Labs effort [28] that extends the software chain of trust wih a trusted VMM hypervisor for fine-grained immutability protection of both OS kernel and applications.

7 Conclusions and Future Work

We show that current assumptions about the run-time state of measured memory do not properly account for possible changes after the initial measurement. Specifically, previously measured memory can be modified at run-time, in a way that is undetectable by the TPM. We argue that these assumptions of the OS and application trustworthiness with respect to memory operations can and should be backed up by a combination of a memory event trapping framework and associated TPM operations performed by its trap handlers, to ensure that the programmer's expectations of access patterns to the program's sensitive memory objects are indeed fulfilled and enforced.

We demonstrated several software-based TOCTOU attacks on measured memory, considered ways to detect such attacks—by monitoring the relevant PTEs and physical frames of RAM—and presented a Xen-based proof-of-concept.

One avenue of future work is to explore how to have the TPM reflect other avenues of change to measured memory. Currently, we only protect against physical memory accesses that are resolved by traversing the page tables maintained by the MMU; one future path is protecting against *Direct Memory Access* (DMA) as well. In our initial prototype, we have not implemented any functionality related to paging to disk, and leave that to future work.

We would also like to explore more carefully monitoring *data objects* as well as software, elevating it to a subset of the program's policy goals. In particular, we suggest that the programmer should be provided with the secure programming primitives to reflect the changing access requirements of sensitive data objects. In particular, besides critical code, it would be beneficial to monitor important data structures, such as those involved in linking: various function pointer tables (*Global Offset Table* (GOT), *Procedure Linkage Table* (PLT) and similar

application-specific structures. We argue that the programmer should be given tools to express the security semantics of such objects.

In their current specification and implementation, the sealing/wrapping and signing facilities do not bind secrets and data to their 'owner' process. (By 'owner' we refer to the application that either encrypted/signed a piece of data or generated a key.) This lack of binding could have security implications. Any running application on the system could potentially unseal the data or unwrap the key, and use it for unauthorized purposes. The TCG specification does have a provision to guard against this – specifying a password [14] that will be checked against at the time of unsealing or unwrapping, in conjunction with checking the value in the PCRs. However, if no password is specified, or if it is easily guessable, it leaves open the possibility for unauthorized use.

One way of resolving this problem would be to have a dedicated resettable PCR on the TPM that would be extended with the hash of the binary of the currently executing process on the system. This PCR would be included in the set of PCRs that are used for sealing/wrapping against. As a consequence, the 'owner' process would be required to be currently active on the processor for unsealing/unwrapping to be successful. Every time there is a context-switch (indicated by a change of value in the `CR3` register), the above-mentioned PCR would first be reset, and then be extended with the relevant hash value. This mechanism would prevent a process that is not the 'owner' of a particular sensitive artifact from accessing it.

Implementing the dynamic TPM, as described in Section 4, would be a radical step forward in the way TPMs currently operate. It would enable the TPM to hold the run-time memory configuration of a process, and hence allow for more accurate trust judgments.

Essentially, the TPM itself is a "secure programming primitive", a new tool for software developers to secure critical data and enforce policy. Our proposed TOCTOU countermeasures—and future research built on them—are an extension of this tool for developers to take advantage of these fundamental new primitives to secure data and enforce policy. We note that extending the TCG architecture with additional primitives for ensuring trustworthy memory behaviors appears to be a natural direction towards policy-enforcing next generation trusted systems as per vision outlined in [2].

We also argue that allowing developers to express the intended properties of their memory and code objects as a matter of policy, and providing a TCG architecture-based mechanism for enforcing such policies will help developers to manage previously neglected aspects of their program's trustworthiness. We draw historical parallels with other successful secure programming primitives that provided developers with capabilities to express the intended privilege and access behaviors in enforceable ways, and that have undoubtedly resulted in improving programs' trustworthiness.

---

[14] The password is stored in a wrappedKey data structure associated with the corresponding asymmetric key.

Acknowledgments

We would like to thank our colleages Dr. Douglas McIlroy, Dr. Andrew Campbell, Ph.D. candidates John Baek and Alex Iliev, and our other helpful colleagues at ISTS and Dartmouth PKI/Trust Laboratory for useful discussions. Preliminary versions of these results have appeared in the second author's thesis.

References

1. Trusted Computing Group: Homepage, http://www.trustedcomputinggroup.org
2. Proudler, G.: Concepts of Trusted Computing. In: Mitchell, C. (ed.) Trusted Computing, IET, pp. 11–27 (2005)
3. Bratus, S., Ferguson, A., McIlroy, D., Smith, S.: Pastures: Towards Usable Security Policy Engineering. In: ARES 2007: Proceedings of the The Second International Conference on Availability, Reliability and Security, Washington, DC, USA, pp. 1052–1059. IEEE Computer Society, Los Alamitos (2007)
4. Sadeghi, A.R., Stüble, C.: Property-Based Attestation for Computing Platforms: Caring about Properties, not Mechanisms. In: New Security Paradigms Workshop (2004)
5. Arce, I.: The Kernel Craze. IEEE Security and Privacy 2(3), 79–81 (2004)
6. Franklin, M., Mitcham, K., Smith, S.W., Stabiner, J., Wild, O.: CA-in-a-Box. In: Chadwick, D., Zhao, G. (eds.) EuroPKI 2005. LNCS, vol. 3545, pp. 180–190. Springer, Heidelberg (2005)
7. Xen: Virtual Machine Monitor, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/
8. Bochs: IA-32 Emulator Project, http://bochs.sourceforge.net/
9. QEMU: Open Source Processor Emulator, http://www.qemu.com/
10. Strasser, M.: Software-based TPM Emulator for Linux. Department of Computer Science. Swiss Federal Institute of Technology Zurich (2004)
11. Berger, S., Caceres, R., Goldman, K., Perez, R., Sailer, R., van Doorn, L.: vTPM – Virtualizing the Trusted Platform Module. In: 15th Usenix Security Symposium, pp. 305–320 (2006)
12. D'Cunha, N.: Exploring the Integration of Memory Management and Trusted Computing. Technical Report TR2007-594, Dartmouth College, Computer Science, Hanover, NH (May 2007)
13. Kursawe, K., Schellekens, D., Preneel, B.: Analyzing trusted platform communication (2005), http://www.esat.kuleuven.be/cosic/
14. Sadeghi, A.R., Selhorst, M., Stüble, C., Wachsmann, C., Winandy, M.: TCG Inside - A Note on TPM Specification Compliance.
15. Kauer, B.: OSLO: Improving the security of Trusted Computing. Technical report, Technische Universitat Dresden, Department of Computer Science (A later version appeared at USENIX Security 2007) (2007)
16. Sparks, E.: TPM Reset Attack, http://www.cs.dartmouth.edu/~pkilab/sparks/
17. Greene, T.: Integrity of hardware-based computer security is challenged. NetworkWorld (June 2007)
18. Sparks, E.: A Security Assessment of Trusted Platform Modules. Technical Report TR2007-597, Dartmouth College, Computer Science, Hanover, NH (June 2007)
19. Boneh, D., Brumley, D.: Remote Timing Attacks are Practical. In: Proceedings of the 12th USENIX Security Symposium (2003)

20. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: USENIX Security Symposium, pp. 223–238 (2004)
21. Marchesini, J., Smith, S.W., Wild, O., Stabiner, J., Barsamian, A.: Open-Source Applications of TCPA Hardware. In: Yew, P.-C., Xue, J. (eds.) ACSAC 2004. LNCS, vol. 3189, pp. 294–303. Springer, Heidelberg (2004)
22. Marchesini, J., Smith, S.W., Wild, O., MacDonald, R.: Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Dartmouth College, Computer Science, Hanover, NH (December 2003)
23. Haldar, V., Chandra, D., Franz, M.: Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In: USENIX Virtual Machine Research and Technology Symposium (2004)
24. Petrom Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In: 13th USENIX Security Symposium, pp. 179–194 (2004)
25. Shi, E., Perrig, A., van Doorn, L.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: IEEE Symposium on Security and Privacy, pp. 154–168 (2005)
26. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 2–13. ACM, New York (2008)
27. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: SOSP 2007: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 335–350. ACM, New York (2007)
28. Cabuk, S., Plaquin, D., Dalton, C.I.: A Dynamic Trust Management Solution for Platform Security Using Integrity Measurements. Technical report, Hewlett-Packard Laboratories (April 2007)