SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms

Sergey Bratus¹, Michael E. Locasto², Brian Schulte²

¹ Institute for Security, Technology, and Society, Dartmouth College, Hanover, NH

² Computer Science Dept., George Mason University Arlington, VA

Abstract. The TPM is a fairly passive entity. As a result, it can be difficult to involve the TPM in measurements of software trustworthiness beyond simple load-time hashing of static program code. We suggest an approach to dynamic, runtime measurement of software trustworthiness properties as they relate to code-data ownership relationships. We outline a system, SegSlice, that actively involves the TPM in fine-grained labeling and measurement of code slices and the data that these slices operate on. SegSlice requires no changes to x86 hardware, and it relies on the relatively underused x86 segmentation mechanism to mediate access to data events.

1 Introduction

Measuring and enforcing the trustworthiness of a piece of running software is a non-trivial problem to which few satisfactory solutions exist. Being assured that software is behaving as expected (one plausible definition of “trustworthy”) faces the central technical problem of defining what behavioral features to measure (and how often to measure them, not to mention the problem of communicating such results to the user in an understandable fashion). Although the TPM provides a tempting, tamper-resistant, high-assurance resource for verifying periodic measurements of system state, to date it has largely been used to verify hashes of static program code at load time.

Assuring that the dynamic properties of a running process are within a particular behavioral envelope presents a much more daunting challenge, and it is unclear how the TPM might be pressed into service to such an end. In this paper, we suggest an architecture for doing so that does not require changes to the TPM or changes to the underlying machine architecture (in this case, x86). We are able to take advantage of the x86 segmentation mechanism, the flexibility of the ELF binary format specification, and small modifications to the Linux kernel to provide a general framework for efficiently measuring and enforcing code-data ownership properties in a running program.

Periodic measurements of program state that are open to TOCTOU attacks; as a result, one closely related piece of prior work suggests an architecture for supplying *selective memory immutability* [1] as a new secure programming primitive for a TCG platform. We propose a new family of runtime primitives that are intended to build on the same programmer intuitions, and of which selective memory immutability [1] design is strictly a specific case.

1.1 Background and Previous Work

Bratus *et al.* [1] point out one inherent weakness in the TCG approach to software trustworthiness: namely, its lack of OS and hardware support for using the TPM – itself a *passive* security primitive, for programmers to avail themselves of – to enforce dynamic security policies that require preservation of attestable properties (say, integrity of a particular memory region containing a set of critical

data structures) throughout a program’s runtime. The paper showed that this weakness presents multiple opportunities for TOCTOU attacks.

In order to address this weakness the authors proposed a broad approach to creating TCG architecture-based *runtime* policies, which was to complement the TPM’s software measurement support (that is, for cryptographic checksumming of certain memory regions where the software was loaded) with a *memory trapping functionality* that would cause a trap at *memory events* that threatened the integrity of the measured software. The trap handler would take advantage of the (passive) TPM to re-measure software and thus eliminate a potential TOCTOU attack.

Thus a TCG platform with the proposed trapping architecture would go beyond assuming that trustworthiness of software equals its measurement on loading, and become capable of enforcing dynamic properties of software, such as *selective immutability* of memory regions, by making sure that all relevant memory events – in particular, those dealing with page table management and virtual address mapping – are mediated by appropriate, TPM-aware handler logic.

The authors based their prototype implementation on the memory trapping (in particular, page table management mediation) functionality of the Xen, but noted that similar results could be achieved with other general or dedicated hypervisors, or, better yet, with specialized MMU support.

These ideas were developed further in subsequent position papers [2, 3], outlining the prospects and promise of how an MMU enhanced with logic for trapping flexibility can bring within reach broad new classes of security policies that cannot be efficiently enforced on modern hardware, but that would allow software developers to express *policy-critical* properties of the software just like they express *correctness-critical properties while debugging* with the modern tools such as DTrace, Pin, SystemTap, etc., at modest hardware and OS support costs.

We believe that the future of trustworthy computing will belong to platforms that support Boolean logic-enhanced, object-granular, and developer-friendly expression of trappable conditions. It may, however, be years until hardware vendors develop and deploy such hardware support. In the meanwhile, we set out to distill the best trustworthiness-related programming practices – especially from such trust- and security-conscious communities as BSD kernel programmers – to the point at which they could be implemented on a modern x86 MMU, albeit with some loss of efficiency.

We propose to extract, allow the developers to specify, and then to enforce at runtime the “ownership” relationships between units of data and code (a.k.a. “slices”). We point out that these relationships are already present in the ELF binary format, between its standard code and data sections, and that their violations are tell-tale signs of attacks. We also describe a mechanism for developers to specify their own code and data units such that their relationships will be enforced throughout runtime, and their violations would be trapped and processed by the TPM-aware handler in a similar manner to [1].

Code–data relationships extend selective immutability. In fact, selective immutability in the sense of [1] can be considered an almost-trivial subcase of a code–data relationship. Namely, selective immutability merely says that *no* code is allowed to write the data other than in select circumstances, thus being, in essence, a simple *write-ownership* policy with an empty set of owners for the selectively immutable data segment.

We explain the rationale for our proposal in the next section, and then describe the prototype implementation in Section 3.

2 Code–data relationships and trustworthiness

Privileges. Trustworthiness of software (and, more generally, the trust we put in software) is ultimately described in terms of its behavior being aligned with our expectations. We point to the

arguments in [2, 3] noting that debugging and testing can be seen as trust-related activities – because they are essentially *procedures to establish a connection between the expected and the actual behaviors of a program*.

Of course, without a definition of *what behaviors* are relevant to security expectations, such considerations remain theoretical. Classic works define such behaviors as operations on certain systems objects, and formulate expectations in terms of *privileges* to perform these operations. Security models are then equated with privilege management schemes (such as “least” or “role-based” privilege).

However, we believe that such view – despite having worked well in the past, and still working well in environments where security goals can be expressed in terms of read or write access to files – is quite limiting. For example, this view perfectly suited MLS goals (“no read-up, no write-down”), but cannot account for the emerging attacker maxim of “Code running in userspace can always run as Ring0”³ (implying that the kernel reference monitor itself is in danger if arbitrary code execution can be achieved within a user process), because in fact **none of the behaviors characterising the phases of exploitation leading to the code execution achievement can be described in terms of file access privileges**.

In particular, these exploitation steps mostly happen outside of the filesystem namespace, but rather on with the kinds of runtime objects dealt with linkers and loaders, such as segments mapped and allocated in the process’ virtual memory space.

Even though the underlying OS manages them (e.g., when creating the process from an ELF binary file), it has no system of privileges granular enough to describe “privileges” of operations on them, such as memory mapping operations. **We attach crucial significance to this observation, and posit that to improve trustworthiness, we will need to enforce at runtime known exclusive access relationships between data and code units (“slices”) of a program.**

Code–data relations reflected in standard ELF structure. The modern usage of the ELF format as used by GNU/Linux and OpenSolaris makes many fine semantic distinctions between the components of a binary (see, e.g., [4]).

For example, a typical ELF format executable on a modern GNU/Linux system contains about 30 memory sections, which correspond to semantically different contents of contiguous areas of memory to be interpreted by the runtime toolchain (such as GNU binutils, the OS loader, and the dynamic linker). Fundamental shared objects such as `/lib/libc.so.6` may contain upward of 70 (!) sections. This semantic diversity, even though not recognized by every runtime tool (which typically requires only one of its facets as represented by the sections’ properties for its operation), testified to the granularity of information the OS *could* be using at runtime.

Notably, the relationships between these semantic units are explicitly expressed in the ELF section and segment header tables (specifically, their **Info** and **Link** fields). Some of these relationships are “metadata-to-data”, specifying that a particular section shall be used to interpret and transform another section, e.g., by way of relocation (say, `.rel.text` to `.text`), whereas others are strictly code–data ownership, such as the executable `.init` and `.fini` to their respective driving data `.ctors` and `.dtors`.

Ultimately, these relationships arise from the programmers’ efforts to control program *complexity* such as decomposition of functionality into generic, toolchain-provided, well-tested standard logic (such as `.plt` dynamic linking stubs and `.init` constructor stubs, and their program-specific parametrizations (such as, respectively, `.got` and `.ctors`). The very same techniques are used to

³ See, e.g., DailyDave posting <http://lists.immunitysec.com/pipermail/dailydave/2010-January/006000.html>, as based on the recent Linux kernel NULL pointer vulnerability discovered by Tavis Ormandy, e.g., <http://blog.cr0.org/2009/08/linux-null-pointer-dereference-due-to.html>

increase the programs’ *trustworthiness* – of which complexity is arguably the worst enemy, and good complexity management is arguably the best cure.

It is no surprise, therefore, that violations of these code–data relationships signal exploitation, from the direct destructor pointer replacement⁴ to heap boundary tag manipulation.⁵

2.1 Developer ways of keeping systems tractable.

We further note that the (static) expression of code–data ownership relations has become indispensable to large C projects, in particular, to operating systems kernels. In particular, the use of file-local symbols (both variables and functions) has been steadily growing in the Linux kernel as shown in Table 1.

	2.0.1	2.2.0	2.4.0	2.6.0
#	736	1833	3336	6675
total				
lines	512,825	1,205,990	2,241,755	3,853,242
%	.114	.152	.149	.173

Table 1. File-scoped symbol declarations in the Linux kernel

We note that the underlying separation is merely static and can only be detected during the build process, not the runtime. We argue, however, that had the kernel runtime permitted it, the enforcement of the same access restrictions would describe a security property of the code’s behavior, that is, a useful trustworthiness property, analogous to the “least privilege” at the kernel object level.

We note that one of the earliest Linux kernel exploitation techniques was based on patching kernels in such ways that subverted or bypassed loadable kernel module support, or imitated it in case when it was purposefully disabled.⁶ Furthermore, subsequent advanced attacks were based on manipulating virtual memory translation mechanisms through kernel code units that were never meant to affect it, such as, e.g., [5] – which succeeded specifically against an MLS-hardened system. A history sketch of these exploits can be found in [6].

3 Implementation

How can we provide the ability to partition applications at a very fine-grained level, where each “slice” of an application is dynamically defined by the collection of data it “owns”? Slices can cross thread boundaries: we define them to be the elements (instructions, control statements) of a program concerned with the manipulation of a certain exclusive set of data structures and variables. We would like to perform this partitioning without requiring any hardware changes, even though

⁴ E.g., <http://www.packetstormsecurity.org/papers/unix/manipulating.dtors.txt>, <http://www.securiteam.com/unixfocus/6H00I150LE.html>

⁵ <http://www.phrack.com/issues.html?issue=57&id=8>, <http://www.phrack.com/issues.html?issue=57&id=9>

⁶ <http://vxheavens.com/lib/vsc07.html>, <http://www.phrack.com/issues.html?issue=58&id=7>, <http://www.phrack.com/issues.html?issue=58&id=8>

the enforcement of such fine-grained access control seems to entail hardware support that is not currently present in IA-32 or x86_64. Nevertheless, we turn to x86 segmentation for our prototype implementation.

3.1 Isn't x86 segmentation dead?

While preparing this paper, we received the following comment that very succinctly summarizes the reasons why x86 segments *as they exist today* are suboptimal for implementing policy mechanisms [7]:

- segmentation makes all memory references slower: using non-flat segments cost a least a cycle on AMD CPUs;
- not all x86_64 CPUs have segmentation support;
- reloading a segment takes a trap/syscall and GDT access;
- only 4 segments (DS, ES, FS, GE) can be accessed at the same time
- only 8K different variables can be protected;
- running with $CPL = 2$ is tricky, as paging assumes superuser privileges in that case;
- the mechanism should guard against attack code loading a flat segment and overwriting the kernel with it;
- wrong error codes in the page fault handler.

We readily agree that the x86 segments would need to change if they were to allow efficient policy enforcement and that their current state does not allow a clean implementation free of ad-hoc fixes. However, we believe that segments are still the closest we have to a more expressive MMU trapping of memory events in x86, and it would probably be good for the security of the platform if the segmentation mechanism were re-optimized to reduce the performance hits and the need for special case band-aids. *The example of x86 virtualization, which arguably started out in a worse state, and – having demonstrated its utility with less-than-perfect prototypes, received great hardware and developer support – gives us hope that segmentation may also be similarly restored and reinvented.*

The paging mechanism might seem to be more appropriate for implementing memory protections within the MMU. However, we believe that a conceptual separation of the MMU's functionality into that supporting security primitives and that focusing on performance is desirable. We also note that natural granularity requirements for strong security *isolation* and multi-level storage *performance optimization* differ substantially, and therefore the paging mechanism should not be overloaded with security functions, as a matter of separating security and performance concerns. The segmentation mechanism appears to offer a more natural expression of any protections granularity required by existing code–data relationships, whereas the size of a page should be determined primarily by performance considerations.

3.2 Prototype internals

The purpose of this implementation is to augment the Linux kernel's ability to execute 32-bit ELF binaries with a way to automatically wrap individual variables and data structures in their own x86 segment and restrict access to them without hardware modification. We seek to provide a way of mimicking a variable number of "privilege rings" (in IA-32 there are only 4 such rings, limited by the two DPL bits in the segment descriptor and two CPL bits in the CPU). While we could conceptually add a full register for keeping track of these bits, and thus provide 2^{32} possible privilege rings (rather: non-monotonic levels of access), we cannot do this to pre-existing commodity hardware (unless we emulate it in software such as Bochs or QEMU).

Instead, we can modify the OS kernel. The key insight is that if every variable is in its own segment, and we change the DPL bits in the corresponding segment descriptors to be at the privilege level 2 (rather than the normal 3 of user-space), we can trap to the kernel for validation along with a great deal of flexibility to place variables in separate containers. While we cannot directly create new privilege ring levels, we use privilege ring level 2 as a layer of indirection to force a trap and then a subsequent check in the kernel software on whether or not this instruction should be in fact allowed to access the requested data (in essence, we are checking whether the instruction belongs to the “slice” owning that particular data). Thus, we can define per-data instruction slices (a fairly high number of them, limited only by the maximum number of segments on x86 – roughly about 8000).

3.3 Segmentation in x86 Linux

Linux’s use of segmentation on a 32-bit machine is limited. There are four main segments utilized by the operating system, two for a user and kernel code segment and two for a user and kernel data segment. These entries are contained in the Global Descriptor Table (GDT). In order to keep track of what code and data segments are currently being used by the operating system there is a code and data segment register. These registers contain segment selectors that point to entries inside either the GDT or the Local Descriptor Table (LDT). We summarize this system in Figure 1.

Fig. 1. The x86 segment selectors and tables.

The LDT is a separate descriptor table that is mostly used to hold segments used locally by a particular process. These can be custom segments created by the programming through the system call *modify_ldt*. The *modify_ldt* system call provided with a *user_desc* struct, which is comprised of information about the segment descriptor, creates a new entry in the LDT.

3.4 The Anatomy of the Mechanism

The goal of SegSlice is to protect important variables by having them reside at a privilege level of two instead of the conventional level of three (user land). A check is made by way of a custom system call (for experimental purposes) for the validity of instructions modifying protected variables. This system call will also be responsible for returning the user land process to the default level of three. We recognize that having a custom system call is not the optimal way to handle this, but is most likely the cleanest to experiment with. Other options could be using a hardware exception, taking advantage of certain policies of segmentation and getting the trap for “free”, or possibly having a virtual device in */dev* to handle the accesses of protected variables.

Each guarded variable will be provided with its own data segment that encapsulates only itself. These custom segments will be contained inside of the Local Descriptor Table (LDT) of the current process. The system call *modify_ldt* will be used to write these descriptors to the table. In order to allow these segments to reside at ring two, a field in the *user_desc* struct (used by *modify_ldt* to write the LDT entry) had to be created to allow for the modification of the privilege level (by default it is set to three).

Before each attempt to modify the protected variables, a few things have to happen. Eax has to be pushed onto the stack for restoration at a later point. This register is used to hold the segment selector that will be loaded into the data segment register (DS). This is because a direct *mov* to segment registers is not allowed. Then, the custom system call must be called. The segment selector that is used to access the variable has to be passed as the single, lone parameter.

The custom system call is responsible for performing the check to certify the potentially modifying instruction. There are two ways to perform this check. The first is to see if the instruction in question is in a special hash table in the kernel. The table contains instruction addresses and the corresponding segment selector in the LDT for that particular code segment slice. This table represents code slices that have already proven to be valid. If the instruction is present in the hash table, the corresponding segment selector will be used to overwrite the CS value saved on the stack.

The other check is to see if the instruction is inside the *.text* section of the corresponding ELF. This check only occurs if an entry cannot be found in the hash table for the instruction. In order to perform this check, a modification of the loading of ELF's needs to be made. Inside of the *load_elf_binary* function, each section of the ELF is loaded into memory one at a time. When the *.text* section is loaded, it is copied into a newly added field in the *task_struct* of the process. Then the exception handler will have access to a list of all valid instructions. In order to search the *.text* section in an efficient way, we use the value saved from EIP of the faulting instruction and begin searching there for the instruction we want to validate. If the *.text* section contains the instruction, a new LDT entry is created and an entry is placed in the hash table.

If the instruction is proven to be valid, the process of altering the stack to return to the process at ring two begins. We overwrite the old value of DS on the stack with the value that was passed as a parameter to the system call. From the hash table of instructions and their corresponding segment selectors, the appropriate value is used to overwrite the old CS value on the stack. By overwriting these values, this forces the kernel to perform its normal return to the calling process, but instead of loading the default user land CS and DS, loading our custom segment selectors allowing for execution at ring two.

After the system call is done overwriting DS and CS, we return back to the user land process. After the instructions are finished modifying the now unguarded variable, we make a call to the system call once again. We pass it a special parameter (0 or another value that will not be used as a segment selector) and the system call consequently overwrites CS and DS on the stack (which are now pointing to ring two descriptors) with the default user land selectors for ring three. Returning back to the process after this second call re-armors the variables.

Each protected variables segment has a base of its own address, so some binary rewriting is needed to correct the offsets in the ELF. Originally, the compiler produces an offset from address 0 (the base of the normal data segment). If we leave the original offset in place, it will try to calculate the position of the variable as *[the variable's address + the original offset]* which will undoubtedly lead to a segmentation fault or other unknown results. For beginning experiments, the binary can simply be sifted through for these instructions, and have the offset changed to 0. For a more automated approach, a binary rewriting tool such as Diablo can possibly be used.

Integration with the TCG platforms. The above segmentation-based design provides the trapping framework for capturing and mediating events of code-data accesses of interest, either between the elements of a platform's ELF ABI or between the programmer-defined units.

This SegSlice framework provides a way for measuring related *dynamic* properties of a process throughout its runtime. As mentioned above, the measurement will be invoked by a trapped event, in the corresponding trap handler function.

A TCG platform can take advantage of these measurements in several ways:

1. reset the TPM's PCRs in the trap handler if the measurement suggests violation of the program's desired trustworthiness properties,
2. seal a runtime memory object to the TPM's state, e.g., allowing relinking or re-keying of the program only in trustworthy states, and
3. otherwise maintain a (limited) state record of an application to ensure fulfillment of temporal properties (e.g., enforcing a requirement on the order of certain events).

In all of these cases, the handler will either call on the TPM as a provider of cryptographic services for a TCG-side of a trustworthiness-related multi-step secure computation, or simply signal to the TPM that the trustworthiness property has been irrevocably lost.

3.5 Future Work

Once a working prototype is established for testing, much can be done to improve the project. To move to a more unified mechanism across all machines, a change will have to be made in how the instruction validation is performed. Using a custom system call does not allow for consistency across various systems. Using existing mechanisms in x86 protected mode is an option to allow for a "free" trap to the kernel. The attempt to load a data segment register with a DPL that is lower than the current CPL of the code segment register results in a general protection fault (GPF). This fault is handled by the handler *do_general_protection* where it is possible to intercept the special cases that are caused by SegSlice and handle them appropriately. The clean up process to return to user land is more complicated than a simple system call because the kernel never intends to return from a GPF. For this reason, a custom system call will be used for experimental purposes.

Ideally, this entire process should be transparent to programmers. A dynamic way of identifying particular code slices and important variables through automated binary rewriting would be optimal. Most likely, this would involve a custom binary rewriter that would search through the binary to identify particular code slices that touch important variables in the program. The rewriter would

then insert the instructions to encase the to-be guarded variables inside their own segments and also instructions to handle the trap to the kernel and restoration of ring three after the trap.

4 Related work

We owe our approach to a large quantity of related work. The intuitions on what constitutes an efficient, developer-friendly secure programming primitive tend to travel around a lot, and, before they settle into a successful, productive form of their own, they often make appearances as useful peripheral features of other projects. Thus their full and fair attribution would require a broad survey that is unfortunately beyond the scope of this paper.

We separate this section into two parts: in the first, we discuss the projects that we consider most closely related to ours in either their use of the x86 segmentation or in their emphasis on code and data units in a process. In the second part, we discuss the general trends in related work that we believe support our approach.

4.1 Projects most closely related to SegSlice

We trace the idea of using the program’s natural separation into modules as a means of also partitioning it into security-related contexts to [8], in which the authors presented a mechanism to enforce principles of modularity while protecting client code from library code. This is accomplished by two designs: *protected libraries* and *context specific libraries*. When accessing a library routine, through a defined access point, access to the client code is revoked and upon exiting the routine is reinstated. Context-specific libraries (CSL) allow for various sharing policies between client and library code. A CSL may share data with multiple clients, so that each client sees the same data at the same location. CSL’s may also share data with a client and service such that the actual contents of the shared region may be associated with either the calling (client) or the called (service) region.

In [9], leading the comparatively recent re-examination of security advantages of segmentation, the authors take advantage of protection mechanisms in segmentation and paging to protect core programs from their extensions, both at the user and kernel level. In the kernel level, kernel service or module’s extensions are kept at a privilege level of one, instead of the default level of zero for kernel routines. This protects the kernel services from the extensions, but not vice versa. In order to change the context between the two, a call gate is used which takes advantage of their inter-segment or inter-privilege level procedure calls.

For the user level protection, a combination of segment and paging protection is used. User level applications reside at a privilege level of two from the virtual address space of 0–3GB and have a page protection level (PPL) of zero (more privileged than the other level of one). Extensions are marked with a privilege level of three and a PPL of 1 so that they cannot access the code or data of the original application. In order to have shared code or data, another section is marked to have a privilege level of two but a PPL of one so in order to access it, the software extension must promote itself to privilege level two first.

In the previous sections we considered the criticism that the x86 segmentation is a relic of the past, and has in fact been substantially deoptimized by CPU makers, who are ready to completely abandon it. While we agree that the danger of such abandonment is real – and believe that it would be grievous to the state of platform security – we do not believe that this mechanism is less than useful, and point to the following work as supporting evidence.

The recent Vx32 [10] is a robust and efficient sandboxing mechanism that utilizes segmentation in the x86 architecture to run guest code safely on top of the host software. No kernel extensions or special privileges are used by vx32. The vx32 sandbox runs standard x86 instruction so any language may be used. Because it is run completely in user land, it cannot rely on the kernel privilege checking mechanism to make sure the guest does not infect the host, so it must translate all operations to “safe” operations at the loss of performance. The guest software is also limited by a restricted system call API. The host decides for each guest what is considered safe and allowed system calls.

The use of segmentation allows each guest to be isolated inside its own segment. Each guest’s segment address appears to be at the address of 0 and cannot access beyond its segment (the segment limit). Vx32 uses other data segment registers (FS or GS) to contain the segment selectors for the guests’ individual data segments. On 32-bit systems, the code segment register is never changed. When guest code execution begins, it simply jumps to the fragment of code inside the default code segment that contains the guest code. This is safe because vx32 handles any jumps outside of the guest’s code, whether it be to other translated code fragments or back to vx32 for operations such as system calls.

Vx32 provides a sandboxing mechanism that can support one or more guest plug-ins, each local to their own specific segments. The control over each guest’s system call API provides an extra layer of protection to ensure the host remains isolated. By using x86 architecture to isolate memory accesses and translating instructions to “safe” operations, the authors maintain that vx32 is a lightweight, efficient sandboxing mechanism for the x86 architecture.

The Nooks project [11] makes use of the x86 segmentation system to improve OS reliability through isolation of driver code and data in the kernel, recognizing the fact that driver instability is the leading cause of modern commodity OS instability (as well as of security vulnerabilities). We note that Nooks combines both the idea of code and data isolation and transparent trapping of related (driver) failures, and makes use of the x86 segmentation to achieve this goal.

4.2 Other related work

Systems like Valgrind [12] and Pin [13] have recently emerged that enable a programmer or software tester to interweave complex programmatic instrumentation at runtime into an existing software system. These systems use dynamic binary rewriting and do not require access to the source code. Similar environments include the Rio architecture [14] and Dyninst [15].

Program shepherding [16] focuses on ensuring that control flow transfers of a process remain within the bounds of some policy. For example, the technique uses the Rio [14] system to ensure that code in library routines is only accessed via the entry point of the particular library function. Control Flow Integrity (CFI) [17] is a similar idea in which a program’s static control flow graph acts like a policy for the runtime behavior of the system.

In what we believe to be a seminal call to action for the operating systems research community, Roscoe *et al.* [18] argue that current OS research utilizing hypervisors should move away from endlessly refining traditional approaches aimed at Unix/Windows ABI model compatibility. In essence, the hypervisor presents a useful backwards-compatible interface, and Roscoe’s paper argues that the problems that we currently tackle at the VMM level (such as inter-VM communication, resource sharing among VMs, VM isolation) have all been solved at the OS-level, and adding more functionality in the VMM is largely wasted effort. Furthermore, adding needless functionality to the VMM simply increases the size of the trusted computing base (TCB), and hence it becomes harder to prove VMM correctness or security properties. The size and nature of this complexity are discussed

in an article by Karger and Safford [19]; they make many of the same points (and ably illustrate the various interactions) we do with respect to the complexity of VMM I/O systems.

Finally, the object oriented programming discipline is largely concerned with expressing some types of code-data relationships (typically composition and inheritance rules that help govern visibility of data members of classes). Such relationships are typically expressed through the class heirarchy mechanism and the corresponding rules derived from type-checking these relationships at compile time (and in the case of interpreted languages [20], at runtime. Such runtime checking, however, does not incorporate the use of a TPM or other mechanism to provide assurance that the structure and layout of both code objects (methods) and data objects (variables, class instances) remain unmodified by malicious attack (or even random errors [21]).

5 Security Properties of “Slices”

The intention of slices as a programming primitive is to help the programmer decompose the program’s code and data units into groups (that is, “slices”) of different sensitivity with respect to the programmer’s security goals, and to explicitly specify relationships between these groups to be enforced at runtime. When combined with the TCG architecture’s measurements, these relationships will constitute an *attestable* property of the system.

In this regard SegSlice can claim descent from a few tried-and-true security primitives, as we explain below.

5.1 UNIX kernel–userland separation

Most importantly, most of UNIX’s security and reliability properties ultimately rely on the separation of execution contexts into userland and kernel, with the kernel data being directly accessible only to the trusted kernel code. It is this isolation and the resulting trust in the integrity of kernel data that fueled the subsequent development of UNIX security policy enforcement mechanisms, up to and including SELinux.

“Trustworthy data” became synonymous with “data held by kernel”. However, developing new kernel interfaces for each new kind of trust-related data is cumbersome, does not agree with existing application programming practices – essentially, it would imply developing a dedicated companion kernel module for each trusted application, requiring the programmers to become proficient with kernel as well as userland programming environments at the same time. By comparison, SELinux merely requires developers to specify the kernel-loaded companion list of the application’s allowed file operations – a sort of a companion program which describes access-focused behaviors of the main program – and even this task has been an obstacle to developers, who had trouble adapting to the specialized language and environment of type/domain specifications.

At the same time, designing a general enough kernel interface for both safe-keeping and application-specific trusted operations on generic application data that would fit a majority of application programming needs appears to be a very hard task – imagine unifying several large *ioctl(2)* and *setsockopt(2)* interfaces into a semblance of coherence!

5.2 Privilege drop

SegSlice can claim descent from UNIX system calls that allow a process to drop privileges. In particular, such a system call *demarkates code that interacts with certain high-integrity data from the code that is known to have no such need*, and allows the OS to enforce this separation of code into units by access behaviors.

Moreover, an attempt by the process to perform a privileged operation after the privilege drop – that is, a violation of the explicit code boundary set by the call – indicates that the process has entered an untrustworthy state. We also note that the boundary is essentially static.

Unfortunately, even though extremely useful (and widely accepted as a de facto requirement for Internet-facing daemons), such privilege demarcation of code units only lends itself naturally to designs in which the set of privileges “monotonously” shrinks. However, units of code may have other natural sets of privileges than those that nest neatly as subsets of larger sets, and managing them does not fall neatly into the “drop” paradigm.

Even allowing for recovery of privileges (e.g., for temporary drops) opens a can of worms (e.g., [22]), and requires extra care by the programmer that defeats the intuitiveness and ease-of-use (see, e.g., [23], for discussion and proposed solutions). Mechanisms such as *privilege bracketing*⁷ aim to solve this problem.

By reformulating “privilege” as code units’ enforceable relationship with data units, “slices” offer a model that accommodates non-nesting data access privileges. Slice definitions are contained within the program itself, and thus do not require external configuration of new user accounts with corresponding rights, avoiding additional burden on system administrators (but, as a downside, they are not transparent to the system administrator, unlike user and group definitions).

6 Two Sides of the Same Coin: Slices and Return-oriented Programming

The formulation of SegSlice presented in this paper (segments denoting code slice ownership of specific data backed by TPM-supported checking of code provenance) is most directly applicable to preventing either foreign injected code or inappropriately re-tasked existing code (a la return-oriented programming) from accessing data and variables that it does not own. In this sense, Segslice provides an automated, fine-grained isolation mechanism supporting legitimate code slices as a defensive technique.

In contrast, the dynamically composed sequences of executing instructions employed by return-oriented programming attacks represent inappropriate or illegitimate “slices” of code functionality: slices which ultimately seek to access or modify data that does not belong to any such slice. Thus, Segslice seeks to enable programmers to define slices of program execution in terms of legitimate data access, whereas ROP-style attacks seek to dynamically compose substrings of existing slices to effect a “malicious computation.”

Exploit programming relies on using elements of the target execution environment to accomplish a “malicious computation” (see [24], [25] for discussion of the term) deemed impossible or improbable under the target’s trust model.

We found that it helps to think about exploits as programs written in “*weird instructions*” – fragments of standard mechanisms present in the runtime environment, such as libraries, parts of the ABI or calling conventions, dynamic linking logic handling the Global Offset Table (GOT) and Procedure Linkage Table (PLT), compiler-supplied pre-entry and post-exit wrappers for a program, and many other kinds – that are accessible for “off-label” uses unanticipated by the trust model, and accomplish specific tasks, possibly with many side-effects.

The effects of these *weird instructions* could range from overwriting a word at an address controlled via the input with an integer from a library function’s state (e.g., [26, 27]) or given by a neighboring word (e.g., [28, 29]), to loading and linking an entire missing library via jumping to a part of the dynamic linker code (e.g., [30]). Typically, practical exploits mix and match such “meta-instructions” to both achieve the computation and mitigate its side-effects.

⁷ See <http://docs.sun.com/app/docs/doc/816-1042/6m7g4ma52> in Trusted Solaris Developer’s Guide

The nature of exploit programming tends to be mischaracterized. The most famous example is probably the historically common association of exploits with introduction of “*malicious code*” into the system, one way or another. This misleading association turned out to be quite tenacious (cf. [24, 25]). In fact, the idea that such a computation could be accomplished, with sufficient generality and flexibility, without introducing *any* new executable code into the target can be traced back at least to the 1997 hacker publications [31, 32] and the subsequent series of Phrack articles [33, 30], and reached its full impact in academic research with [34] (cf. [25], etc.)

We note that a (and, possibly, *the*) hallmark of exploit programming is **violation of expected code–data relationships involved in trustworthiness assumptions**. Each “weird instruction” essentially depends on such violation (or, from the exploit programmer’s point of view, creates a new and unanticipated one). It is evident in all examples quoted above. For example, in the case of “double free”-based overrides, heap manager/allocator code writes outside the heap area; “return-to-library”’s sequential calling of the crafted frame chain almost certainly involves some function’s code manipulating another function’s stack frame, and so do “return-oriented” gadgets.

Our proposed “slice”-based view of trustworthiness acknowledges this hallmark. We recognize that many exploit programming primitives require too granular analysis of the code and data units involved (e.g., at the level of stack frames), which makes their relationships impractical to describe and enforce with SegSlice. “Slicing” is certainly not a silver bullet that can prevent a malicious computation from happening.

However, SegSlice may provide a programmer with a means of *isolating* a malicious computation, and an indication that the process within which a violation of the SegSlice-enforced properties has occurred is no longer trustworthy.

7 Conclusion

We define a new class of runtime-measured software trustworthiness properties, based on intended, exclusive relationships between the program’s code and data units (“slices”). We describe SegSlice, a trapping framework that supports its measurement and enforcement through using the x86 segmentation system in a “segment virtualization” technique. These units can be defined by the programmer by way of using SegSlice API, or – at a loss of granularity, but still sufficient to capture fairly general kinds of attacks – taken from the semantics of ELF ABI elements. This framework extends the previously proposed TCG platform dynamic secure programming primitive of selective immutability, and provide the TCG with a broader class of trustworthiness properties to be measured throughout the lifetime of a process.

Acknowledgments

Bratus was supported in part by the National Science Foundation, under grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors. Locasto is supported in part by grant 2006-CS-001-000001 from the U.S. Department of Homeland Security under the auspices of the I3P research program. The I3P is managed by Dartmouth College. The opinions expressed in this paper should not be taken as the view of the authors’ institutions, the DHS, or the I3P.

References

1. Bratus, S., D’Cunha, N., Sparks, E., Smith, S.: TOCTOU, Traps, and Trusted Computing. In: Proceedings of the TRUST 2008 Conference. (March 2008) Villach, Austria.

2. Bratus, S., Locasto, M.E., Ramaswamy, A., Smith, S.W.: New Directions for Hardware-assisted Trusted Computing Policies (Position Paper). In D. Gawrock, H. Reimer, A.-R. Sadeghi, & C. Vishik, ed.: *Future of Trust in Computing*, Volume . ISBN 978-3-8348-0794-6. Vieweg+Teubner Verlag — GWV Fachverlage GmbH, Wiesbaden, 2009, p. 30. (2009) 30–+
3. Bratus, S., Locasto, M.E., Ramaswamy, A., Smith, S.W.: Traps, events, emulation, and enforcement: managing the yin and yang of virtualization-based security. In: *VMSec '08: Proceedings of the 1st ACM workshop on Virtual machine security*, New York, NY, USA, ACM (2008) 49–58
4. Levine, J.: *Linkers & Loaders*. Morgan Kaufmann/Academic (2000)
5. Last Stage of Delirium Research Group: Kernel Level Vulnerabilities: Behind the Scenes of the 5th Argus Hacking Challenge. Black Hat Briefings, Amsterdam (November 2001) <http://www.blackhat.com/presentations/bh-europe-01/LSD/bh-europe-01-lsd.ppt>.
6. Arce, I.: The kernel craze. *IEEE Security and Privacy* **2** (2004) 79–81
7. Kauer, B.: Private communication.
8. Banerji, A., Tracey, J.M., Cohn, D.L.: Protected shared libraries: a new approach to modularity and sharing. In: *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association (1997) 5–5
9. Chiueh, T.c., Venkitachalam, G., Pradhan, P.: Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *SIGOPS Oper. Syst. Rev.* **33**(5) (1999) 140–153
10. Ford, B., Cox, R.: Vx32: lightweight user-level sandboxing on the x86. In: *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Berkeley, CA, USA, USENIX Association (2008) 293–306
11. Swift, M.M., Martin, S., Levy, H.M., Eggers, S.J.: Nooks: an architecture for reliable device drivers. In: *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, New York, NY, USA, ACM (2002) 102–107
12. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*. (June 2007)
13. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *Proceedings of Programming Language Design and Implementation (PLDI)*. (June 2005)
14. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: *Proceedings of the International Symposium on Code Generation and Optimization*. (2003) 265–275
15. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications* **14**(4) (Winter 2000) 317–329
16. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure Execution Via Program Shepherding. In: *Proceedings of the 11th USENIX Security Symposium*. (August 2002)
17. Abadi, M., Budi, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity: Principles, Implementations, and Applications. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. (2005)
18. Roscoe, T., Elphinstone, K., Heiser, G.: Hype and Virtue. In: *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HOTOS XI)*. (May 2007)
19. Karger, P.A., Safford, D.R.: Security and Performance Trade-Offs in I/O Operations for Virtual Machine Monitors. In: *IBM Research Technical Report RC24500 (W0802-069)*. (February 2008)
20. Gosling, J., Joy, B., Jr., G.L.S., Bracha, G.: *The Java Language Specification, Third Edition*. Addison Wesley (2005)
21. Appel, A., Govindavajhala, S.: Using Memory Errors to Attack a Virtual Machine. In: *IEEE Symposium on Security and Privacy*. (2003)
22. Chen, H., Wagner, D., Dean, D.: Setuid demystified. In: *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association (2002) 171–190
23. Tsafir, D., Silva, D.D., Wagner, D.: The murky issue of changing process identity: revising “setuid demystified”. In: *USENIX ;login:*. (June 2008)
24. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In Syverson, P., Jha, S., eds.: *Proceedings of CCS 2008*, ACM Press (October 2008) 27–38

25. Hund, R., Holz, T., Freiling, F.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: Proceedings of the 18th USENIX Security Symposium. (2009)
26. scut / team teso: Exploiting format string vulnerabilities (March 2001) Version 1.0.
27. gera, riq: Exploiting Format String Vulnerabilities. Phrack **59**(7) (July 2002)
28. anonymous author: Once upon a free(). Phrack **57**(9) (August 2001)
29. MaXX: Vudo malloc tricks. Phrack **57**(8) (August 2001)
30. Durden, T.: Bypassing PaX ASLR protection. Phrack **59**(5) (July 2002)
31. Designer, S.: Getting around non-executable stack (and fix). Bugtraq mailing list (August 1997)
32. Wojtczuk, R.: Defeating solar designer non-executable stack patch. Bugtraq mailing list (1998)
33. nergal: Advanced return-into-lib(c) exploits (PaX case study). Phrack **58**(4) (December 2001)
34. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In De Capitani di Vimercati, S., Syverson, P., eds.: Proceedings of CCS 2007, ACM Press (October 2007) 552–61