SILVER: Fine-grained and Transparent Protection Domain Primitives in Commodity OS Kernel

Xi Xiong and Peng Liu

Penn State University

Abstract. Untrusted kernel extensions remain one of the major threats to the security of commodity OS kernels. Current containment approaches still have limitations in terms of security, granularity and flexibility, primarily due to the absence of secure resource management and communication methods. This paper presents SILVER, a framework that offers transparent protection domain primitives to achieve fine-grained access control and secure communication between OS kernel and extensions. SILVER keeps track of security properties (e.g., owner principal and integrity level) of data objects in kernel space with a novel security-aware memory management scheme, which enables fine-grained access control in an effective manner. Moreover, SILVER introduces secure primitives for data communication between protection domains based on a unified integrity model. SILVER's protection domain primitives provide great flexibility by allowing developers to explicitly define security properties of individual program data, as well as control privilege delegation, data transfer and service exportation. We have implemented a prototype of SILVER in Linux. The evaluation results reveal that SILVER is effective against various kinds of kernel threats with a reasonable performance and resource overhead.

Keywords: Protection domain, OS kernel, Virtualization

1 Introduction

As commodity operating systems are becoming more and more secure in terms of privilege separation and intrusion containment at the OS level, attackers have an increasing interest of directly subverting the OS kernel to take over the entire computer system. Among all avenues towards attacking the OS kernel, untrusted kernel extensions (e.g., third-party device drivers) are the most favorable targets to be exploited, as they are of the same privilege as the OS kernel but much more likely to contain vulnerabilities. From the security perspective, these untrusted extensions should be treated as *untrusted principals* in the kernel space. In order to prevent untrusted extensions from subverting kernel integrity, many research approaches [7, 12, 25, 31] are proposed to isolate them from the OS kernel. These approaches enforce memory isolation and control flow integrity protection to improve kernel security and raise the bar for attackers. However,

in many situations, strong isolation along is still inadequate and inflexible to secure interactions between OS kernel and untrusted principals, for the following reasons:

First, in commodity OSes such as Linux, kernel APIs (i.e., kernel functions legitimately exported to extensions) are not designed for the purpose of safe communication. Thus, even if untrusted extensions are memory-isolated and constrained to transfer control to OS kernel only through designated kernel functions, attackers can still subvert the integrity of the OS kernel by manipulating parameter inputs of these functions. For example, an untrusted extension could forge references to data objects that it actually has no privilege to access. By providing such references as input of certain kernel functions, attackers could trick the OS kernel to modify its own data objects in undesired ways.

Second, either OS-based or VMM-based memory protection mechanism can only enforce page-level granularity on commodity hardware, which provides avenues for attackers exploiting such limitation. For example, attackers can leverage buffer/integer overflow attacks to compromise data objects of OS kernel by overflowing adjacent data objects from a vulnerable driver in the same memory slab. It is difficult for a page-level access control mechanism to address this problem for its inability to treat data objects on the same page differently.

Finally, current isolation techniques are limited to support sharing and transfer of data ownership in a flexible and fine-grained manner. Considering situations that the OS kernel would like to share a single data object with an untrusted device driver, or accept a data object prepared by a driver, in case of strong isolation, it often requires the administrator to manually provide exceptions/marshaling to move data across isolation boundaries. Although there are clean-slate solutions such as multi-server IPCs in micro-kernels [18] and language-based contracts [13] to address this problem, these approaches are difficult to apply to commodity systems, for the reason that they both require developers to change the programming paradigm fundamentally.

To address these shortcomings, we have the following insight: beside isolation, protection systems should provide a clear resource management of kernel objects, as well as a general method for secure communication. In OS-level access control mechanism such as Linux security modules (LSM), the kernel maintains meta-information (e.g., process descriptors and inodes) for OS-level objects like processes, files and sockets, and it also provides run-time checks for security-sensitive operations. Such mechanism enables powerful reference monitors such as SeLinux [3] and Flume [17] to be built atop. In contrast, there is little security meta data maintained for kernel-level data objects, nor security checks for communication between OS kernel and untrusted kernel principals.

This paper presents the design and implementation of SILVER, a framework that offers transparent protection domain primitives to achieve fine-grained access control and secure communication between OS kernel and extensions. To the best of our knowledge, SILVER is the *first* VMM-based kernel integrity protection system which addresses the above challenges. SILVER's key designs are two-fold: (1) SILVER manages all the dynamic kernel data objects based on

their *security properties*, and achieves fine-grained access control with the support of memory protection and run-time checks; (2) Communication between OS kernel and various untrusted kernel extensions is governed and secured by a set of unified primitives based on existing information flow integrity models without changing programming paradigm significantly. Protection domains in SILVER are enforced by the underlying hypervisor so that they are transparent to kernel space programs. Hence, from the perspective of kernel developers, the kernel environment remains as a single shared address space, and developers can still follow the conventional programming paradigm that uses function calls and reference passing for communication. Kernel program developers could utilize SILVER to ensure neither the integrity of their crucial data would be tampered nor their code would be abused by untrusted or vulnerable kernel extensions, thus prevent attacks such as privilege escalation and confused deputy.

SILVER employs several novel designs to enable our protection domain mechanism. First, in SILVER, protection domains are constructed by leveraging hardware memory virtualization to achieve transparency and tamper-proof. The hypervisor-based reference monitor ensures that security-sensitive cross-domain activities such as protection domain switches will eventually be captured as exceptions in virtualization. Second, we propose a new kernel slab memory allocator design, which takes advantages of SILVER's virtualization features such as page labeling and permission control, with a new organization and allocation scheme based on object security properties. The new memory management subsystem exports API to developers to allow them managing security properties of its allocated objects, and enforce access control rules throughout their life time. Finally, SILVER introduces two new communication primitives: transfer-based communication and service-based communication for securing data exchange and performing reference validation during cross-domain function calls.

We have implemented a prototype of SILVER for the Linux kernel. Our system employs a two-layer design: a VMM layer for enforcing hardware isolation, reference monitoring and providing architectural support for page-level security labeling, as well as an OS-subsystem for achieving the high-level protection mechanism and offering APIs to kernel programs. We have adapted real-world Linux device drivers to leverage SILVER's protection domain primitives. The evaluation results reveal that SILVER is effective against various kinds of kernel threats with a reasonable impact on performance.

2 Approach Overview

In this section we first present several examples of kernel threats to illustrate shortcomings stated in Section 1. We then describe our threat model, and give an overview of our approach.

2.1 Motivating Examples

Kernel heap buffer overflow. Jon [2] illustrates a vulnerability in the Linux Controller Area Network (CAN) kernel module which could be leveraged to trig-

ger controllable overflow in the SLUB memory allocator and eventually achieve privilege escalation. The exploit takes advantage of how dynamic data are organized in slab caches by the SLUB allocator. In specific, the attack overflows a `can_frame` data object allocated by the CAN module and then overwrites a function pointer in a `shmid_kernel` object, which is owned by the core kernel and placed next to the `can_frame` object. Although there are many ways to mitigate this particular attack (e.g., adding value check and boundary check), the fundamental cause of such kind of attack is that the OS kernel is not able to distinguish data objects with different security properties. In this case, data object `shmid_kernel` is owned by OS kernel principal, and it is of high integrity because it contains function pointers that OS kernel would call with full privilege. On the other hand, data object `can_frame` is created and owned by the vulnerable Controller Area Network kernel module principal with a lower integrity level. Unfortunately, Linux kernel does not manage the owner principal and integrity level of dynamic data objects, which results in placing these two data objects on the same `kmalloc-96` SLUB cache with the vulnerability.

Kernel API attacks. As mentioned in Section 1, even with strong isolation and control flow integrity protection, untrusted extensions can still subvert the integrity of OS kernel through manipulating kernel APIs. For example, let us consider a compromised NIC device driver in Linux which has already been contained by sandboxing techniques such as hardware protection or SFI. Due to memory isolation, the untrusted driver cannot directly manipulate kernel data objects (e.g., process descriptors) in kernel memory. However, the attacker could forge a reference to a process descriptor and cast it as `struct pci_dev *` type, which he would use as a parameter to invoke a legitimate function (e.g., `pci_enable_device`). By carefully adjusting the offset, the attacker could trick the OS kernel to modify that particular process descriptor (e.g., change the `uid` of the process to be zero to perform privilege escalation) and misuse its own privilege. We consider such threat as a confused deputy problem caused by insufficient security checks in Linux kernel APIs. Thus, to ensure kernel API security, upon receiving a reference from caller, a kernel function should distinguish the security principal that provides the reference, as well as determine whether that principal has the permission to access the data object associated with the reference.

2.2 Threat Model

In SILVER, kernel developers leverage protection domain primitives to protect the integrity of OS kernel in case that untrusted extensions are compromised by attackers. A compromised extension may attempt to subvert a protection domain in many different ways, which may include: (1) directly modifying code/-data via write instruction or DMA; (2) control flow attacks that call/jump to unauthorized code in kernel; (3) memory exploits such as stack smashing or buffer overflows; (4) confused deputy attack via reference forgery; (5) tampering architectural state such as crucial registers. We discuss how SILVER is designed to defend against or mitigate these attacks throughout the rest of the paper.

In this paper, we primarily focus on the protection of *integrity*. Although we are not seeking for a comprehensive secrecy protection against private information leakage, SILVER could indeed prevent untrusted principals directly read crucial data (e.g., crypto keys) from a protection domain.

SILVER employs a VMM for reference monitoring and protecting the integrity of its components in the OS subsystem. Hence we assume that the VMM is trusted and cannot be compromised by the attacker.

2.3 Protection Domain in SILVER

In this section, we give an overview of key features of protection domain in SILVER.

Data management based on security properties. SILVER maintains security metadata for dynamic data objects in the kernel to keep track of their security properties such as *owner principal* and *integrity level*. Moreover, kernel data objects are managed based on these security properties, and the organization scheme takes advantage of labeling and memory protection primitives provided by SILVER's hypervisor. Such organization guarantees that security-sensitive events will be completely mediated by the reference monitor, which would make security decisions based on security properties of principal and data objects. In this way, SILVER achieves data object granularity in protection domain construction and security enforcement, and addresses challenges stated in Section 1. In Section 4.3, we demonstrate in detail how could these designs prevent various kernel integrity compromises stated in 2.1.

Security controlled by developers. SILVER allows kernel program developers to control security properties of its own code and data in a flexible and fine-grained manner. Security decisions are controlled by developers in the following ways: (1) by leveraging extended allocation APIs, developers can specify which data objects are security-sensitive while others can be globally shared with untrusted principals by assigning integrity labels to its data objects; (2) developers could control the delegation of data object ownership and access permissions with other principals by relying on SILVER's transfer-based communication primitive; (3) developers could ensure data integrity when providing service to or requesting service from other principals by using the service-based communication primitive; (4) developers can control which services (functions) to be exported to which principals by creating entry points both statically and at run-time; (5) developers could use endorsement functions and reference checking primitives to validate received data and reference; (6) developers (and system administrators) could accommodate trust relationships with protection domain hierarchy.

Note that although SILVER's primitive could help both participating security principals to achieve secure communication, the security of a protection domain *does not* rely on other domain's configuration or security status. For example, as long as the OS kernel programmer properly use SILVER's primitives to enforce isolation and secure communication, the integrity of OS kernel would not be

compromised by any untrusted extension which may either fail to use SILVER's primitives correctly or be totally compromised by attacker.

2.4 Abstract Model

In this section we present an abstract model, describing our approach in a few formal notations. The basic access control rules of our model follow existing integrity protection and information flow models [6, 17] with a few adaptations. In our model, a kernel protection domain is defined as a three-tuple: $S =< p, D, G >$, where: (1) p is the principal associated with the domain. For each protection domain S in kernel, p is unique and immutable so that it can be used as the identifier of the protection domain. Thus, we denote a protection domain with principal p as S_p. (2) D is the set of data object owned by the principal. Every data object is associated with an integrity level, which can be either high, low or global shared. We denote the subset of high integrity data objects as D^+ and the subset of low integrity data objects as D^- so that $D = \{D^+, D^-\}$. (3) G is the set of entry point objects, which are essentially entrance addresses through which a principal could transfer its control to another principal. Entry points are specified by the developer on a per-principal basis, yet some of them can also be declared as global shared. For the global shared data objects and entry points, SILVER virtually organizes them in to a global low-integrity protection domain denoted as S_-. We define the set of rules that govern protection domain activities as follows:

Data creation. A principal p can create data objects of either integrity level in its own protection domain. p can also degrade any high integrity data object $d \in {D_p}^+$ to low integrity level so that $d \in {D_p}^-$.

Integrity protection. A data object can only be possessed by only one principal at any time. A principal p can write to a data object d iff $d \in D_p$. p can read from d iff $d \in {D_p}^+$. While p cannot read $d \in {D_p}^-$ directly, p has the capability to increase the integrity level of d via an endorsement API provided by SILVER.

Data communication. In SILVER, data communications are achieved by moving data objects from one protection domain to another. In order to send data to another principal q, p can move its data object $d \in D_p$ to low integrity part of domain S_q so that $d \in {D_q}^-$. However, to ensure that d is safe in regard to the integrity of q, d is kept to be in low integrity and cannot be read by q until q sanitizes and endorses the input data and render d high integrity ($d \in {D_q}^+$).

Cross-domain calls. Another important method for inter-domain communication is through calling remote functions exported by other principals. Exporting functions to a principal q is achieved by creating entry point objects in q's domain. To prevent the abuse of code of a protection domain principal, SILVER guarantees that calling through entry points granted by p is the *only* way to transfer control to principal p. Data transfers through cross-domain calls must obey the previous data communication rules.

Protection domain hierarchy. Besides mutually untrusted principals, SILVER introduces protection domain hierarchy to accurately express one-way trust

in practice (e.g., OS kernel and untrusted extensions). In such case, parent principal has full privilege of its child protection domain in terms of object access and creation.

3 System Design and Implementation

3.1 Overall Design

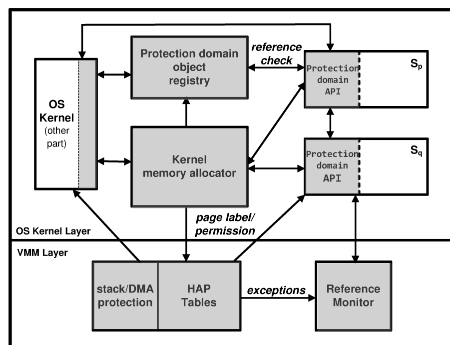

Fig. 1. The architecture of the SILVER framework.

To design a run-time system which enforces our model stated in Section 2.4, SILVER exploits several architectural (hardware and virtualization) features to achieve strong isolation and a coarse-grained, OS-agnostic access control mechanism based on page permissions. On top of these facilities, we design a subsystem for Linux kernel to achieve accountability and fine-grained security control. The kernel subsystem includes a specifically designed kernel memory allocator implementing the core functionality of protection domain primitives, a kernel object registry for accounting kernel objects and supporting reference check, and a set of kernel APIs exported to principals for controlling security properties of their data, performing secure communication and granting capability to other principals. Figure 1 illustrates the overall design of SILVER's architecture, with the components of SILVER in gray. The entire framework is divided into two layers: the VMM layer and OS subsystem layer, respectively. The reference monitor and architectural-related mechanisms are placed in the VMM layer to achieve transparency and tamper-proof.

3.2 The VMM Layer Design

The VMM layer components consist the bottom-half of the SILVER architecture. These components are responsible for enforcing hardware protection to establish protection domain boundaries, as well as providing architectural-level primitives

(e.g., page permission control, control transfer monitoring) for upper-layer components in the OS-subsystem.

Principal isolation. In SILVER, each principal is confined within a dedicated, hardware-enforced virtual protection domain realized by the hypervisor. The protection domain separation is achieved by creating multiple sets of HAP (hardware-assisted paging) tables for memory virtualization, one table dedicated for each virtual protection domain. Upon a protection domain transfer, instead of modifying HAP table entries of the current domain, the hypervisor *switches* to a different HAP table with preset permissions. Using such layer of indirection, each principal could have its own *restricted view* of the entire kernel address space, while the shared address space paradigm is still preserved. Furthermore, by leveraging IOMMU tables, the VMM enables a principal to control DMA activities within its protection domain by explicitly exporting DMA-write permission to other principals and designating DMA-writable pages in its address space. The VMM prohibits any other DMA writes to the protection domain. Finally, to prevent untrusted code tampering with the architectural state (e.g., control registers, segment selectors, and page table pointer) of other protection domains or the OS kernel, the hypervisor saves all the corresponding hardware state of one protection domain before the control transfers to another subject, and restores the saved invariant values once the control is switching back.

Mapping security labels to page permissions. The hypervisor in SILVER also provides a page-based access control mechanism using hardware virtualization. In specific, it exports a small hypercall interface to the OS subsystem of SILVER, allowing it to associate security labels to kernel physical pages. The low-level access control primitives are implemented by mapping security labels to page permissions (i.e., read, write, execute) in each principal's HAP table, which defines whether certain pages can be accessed by the principal via which permissions. In section 3.3, we further describe how SILVER achieves fine-grained data access control on top of these page-based mechanisms.

Securing control flow transfer. By setting up NX (execution disable) bits on corresponding HAP table entries representing pages owned by other principals, the hypervisor is able to intercept all control transfers from/to a protection domain through execution exceptions. Therefore, the reference monitor is fully aware which principal is currently being executed by the processor and uses this information to authenticate principals for the OS subsystem. The reference monitor then validates the <initiating principal, exception address> against the control transfer capability and the set of entry points designated by the owner principal of the protection domain, and denies all the illegal control transfers. To ensure the stack isolation and data safety during cross-domain calls, whenever a call is made by the protected code to an untrusted principal, the hypervisor forks a *private* kernel stack from the current kernel stack for untrusted execution, and it changes the untrusted principal's HAP table mapping of the stack pages to point to the new machine frames of the private stack. Since both virtual address and (guest) physical address of the stack are kept the same, untrusted

code will have the illusion that it operates on the real kernel stack so that the original kernel stack semantics are preserved. After the call finishes, the hypervisor joins the two stacks by propagating legit changes from the private stack to the real kernel stack frames, guaranteeing that only modifications to its own stack frames are committed. In this way, SILVER enforces that all principals have read permission to the entire kernel stack, but only have write permission to their own stack frames.

3.3 OS Subsystem Design

The OS subsystem is responsible for achieving fine-grained protection domain mechanism and providing APIs to kernel programs. It leverages the architectural primitives provided by the VMM layer by issuing hypercalls to the VMM.

Kernel memory allocator The kernel memory allocator in SILVER is responsible for managing dynamic kernel objects according to the rules defined in Section 2.4, as well as providing primitives to kernel principals for controlling security properties of their data objects. It leverages the hypercall interface provided by the VMM layer for labeling physical page frames and manipulating page permissions for different principals. Based on these mechanisms, the allocator achieves the following key functionality: (1) it allows principals to dynamically create objects within specified protection domain and integrity levels. (2) It enables a principal to endorse or decrease the integrity level of its objects at run time; (3) It allows a principal to transfer its data objects to be a low-integrity data object in a contracted protection domain for passing data; (4) It restricts principals from accessing the global name space (i.e., kernel virtual address) to refer objects outside of its domain and provide access control according to the rules.

Our design is an extension to the SLUB allocator [4] of Linux, which manages the dynamic allocation and deallocation of kernel objects. The SLUB allocator maintains a number of cached objects, distinguished by size for allocation efficiency. Physical pages for cache are named *slabs*, which are initialized to have multiple instances of a specific type of objects. Each slab has a `freelist` pointer for maintaining a list of available objects. A slab can have four allocation states: `cpu_slab` (the current active slab for a given cpu), `partial_slab` (portion of the objects are used), `full_slab` (slab objects fully used) and `new_slab` (all objects are available).

Organization. SILVER enhanced the Linux SLUB allocator by introducing heterogeneity to slabs for SLUB caches. In SILVER, each slab is associated with an extra label <*principal, integrity*>, and according to the label, it is restricted to contain kernel objects of the specified integrity level owned by the principal. The memory allocator achieves the slab access control by issuing hypercalls to the VMM layer, labeling and setting up page permissions. Figure 2 illustrates the organization of two `partial_slabs` from the same SLUB cache but with different owner principal and integrity levels. Their heterogeneous labels will eventually

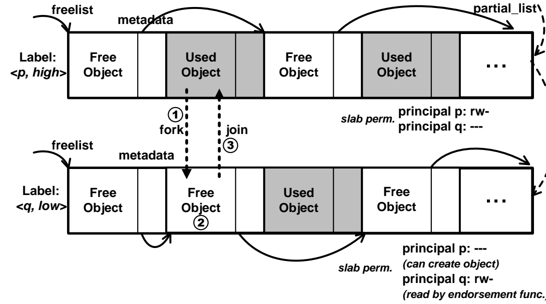

Fig. 2. The layout of two slabs of the same slub cache involved in a service-based communication.

result in different page permissions in principals' HAP table, preventing principals from accessing objects that are disallowed by the access control rules.

Allocation and Deallocation. The kernel memory allocator in SILVER provides a family of secure allocation APIs (e.g., `kmalloc_pd()`) for protection domain principals. These APIs follow the similar semantics of `kmalloc` family functions in Linux, except for having two extra parameters to designate the principal ID and integrity level of the object allocation. The work flow of the allocation procedure is described in Algorithm 1. During slab selection, SILVER must guarantee to pick the slab that matches the security model rather than to choose the first available objects from `cpu_slab` or `partial_slabs`. Once a new slab is created, SILVER must register the label to the VMM to establish principal access control before using it. The deallocation procedure is similar as the SLUB allocator, with extra permission checks on the requested slab. The memory allocator also provides APIs to principals for changing the integrity level of their objects as building blocks for data communication.

Support for secure communication As a major task, the OS subsystem in SILVER is responsible for offering secure primitives to principals for exchanging data, with the strong guarantee of integrity. The data communication is governed by the rules defined in Section 2.4. According to the model, using direct memory sharing to pass high-integrity data is prohibited in SILVER. Instead, SILVER provides primitives for two primary types of data communication: *transfer-based* communication and *service-based* communication. In transfer-based communication, a principal p sends one of its own data object d to another principal q. After that, d will become a (low-integrity) data object of S_q, and can no longer be accessed by p.

In SILVER's implementation, The data object transfer is conducted by the memory allocator by moving data object from one slab to another. In this case, principal p will invoke the API call `pd_transfer_object`, providing its object and q's principal id as input. The memory allocator locates the particular slab (label: $< p, high/low >$) that contains d, removing d from that slab, and copying d to a

Algorithm 1 The procedure for handling allocation requests from a protection domain principal

1: **if** label $< \mathit{principal}, \mathit{integrity} >$ of current `cpu_slab` matches $< \mathit{requesting_principal}, \mathit{integrity} >$ of the requested object **and** `freelist` is not empty **then**
2:     **return** the first available object in the `freelist`
3: **end if**
4: Try to find a `partial_slab` with the matching label
5: **if** `partial_slab` found **then**
6:     Activate this `partial_slab` as the current `cpu_slab`
7:     **return** the first available object in the `freelist`
8: **else**
9:     Allocate and initialize a `new_slab` from the page frame allocator
10:     Associate label <requesting_principal, integrity> to the slab's `page` struct
11:     Issue a `hypercall` to SILVER's hypervisor to label the corresponding physical pages and set up permissions in principals' HAP tables
12:     Activate this `new_slab` and **return** object as of Line 6-7
13: **end if**

slab with the label $< q$, *low* $>$ of the same SLUB cache. The API call will return a new object reference which p could pass to q (but p can no longer dereference to d due to slab access control). Upon receiving the reference, q will leverage SILVER's reference validation primitives (described in Section 3.3) to ensure that the reference is legal, and finally endorse d to complete the transfer. Note that in transfer-based communication, since the object ownership is surrendered, the sending principal must release all the references to the object before calling the `pd_transfer_object`, the same way as it is calling the `kfree` function.

Service-based communication represents the semantic that a principal requests another principal to process its data object, rather than giving up the ownership permanently. In service-based cross-domain call, the original stored location of the data object is not released during the transfer process, instead, a shadow copy of the object is created to be used by the domain that provides the service. After the service call is completed, the updated value of the object is copied back to the original location. SILVER also implements service-based communication based on the SLUB allocator: when a principal p is requesting another principal q to process its own object d, SILVER will first *fork* object d from its current slab to a new object d^* in a $< q$, *low* $>$ slab in the same SLUB cache, and then use the reference of the forked object as the parameter of the cross-domain call. Before the call returns, all the references of d in S_p would dereference to the original d in p's slab. Once the call returns, SILVER will *join* the d^* with d if d^* can be endorsed, committing changes made by q, and free d^* from q's slab. Figure 2 shows the procedure of the corresponding slab operations. Note that in most cases there is no extra hypervisor operation involved during the communication procedure, since both two slabs are pre-allocated so that no labeling/relabeling is required.

Reference validation and object accounting In commodity OS kernel like Linux, fetching data from another principal is usually achieved by obtaining a reference (i.e., pointer of virtual address) to the particular data object. Object references can be passed between principals through function call parameters, function call return values, and reading exported symbols.

As stated in Section 2.1, the absence of reference validation in function parameters could leave avenues for attackers. In order to support reference validation, SILVER must be able to track security information of kernel data objects at run-time so that given any reference, SILVER could identify the object that the reference points to. To further support type-enforcement and bound checking, the type and size information of protected objects must also be known at runtime. By extending the SLUB tracking mechanism, we implemented an accountable resource management layer named object registry, for managing protected objects. The object registry maintains additional metadata for each protected object, and updates metadata upon allocation, deallocation, and communication events. The metadata include allocation principal, owner principal, object size, integrity level, object type and the time of allocation. The object type can be obtained because the SLUB allocator follows a type-based organization, and for generic-sized types, we use the allocation request function/location (the function that calls `kmalloc`) as well as the object size to identify the type of the object.

SILVER ensures that references passed through the `pd_transfer_object` API and service-based communication functions through designated parameters must be owned by the sender principal. In addition, the object registry offers basic primitives to principals for implementing their own reference validation schemes.

4 Evaluation

In this section, we first describe the implementation of our prototype, then we show how to apply SILVER to existing kernel programs for establishing protection domains. In Section 4.3, we demonstrate SILVER's protection effectiveness using security case studies of different kernel threats. We evaluate the performance of SILVER in Section 4.4.

4.1 Prototype Implementation

We have built a proof-of-concept prototype of SILVER. The VMM layer is an extension of the Xen-based HUKO hypervisor [31], with a few hypercalls and exception handling logic added. The OS subsystem is based on Linux kernel 2.6.24.6, and deployed as a Xen guest in HVM mode. Protection domain metadata are maintained in various locations. For each security principal we maintain a security identifier *prid* in the `module` struct, and we encode the slab label <principal, integrity> as additional flags in the corresponding `page_struct`. The object registry is organized in a red-black tree with the object address as the key value. In addition, to facilitate monitoring for the administrator, we export the

run-time status of protection domains in the kernel, including object information and exported functions, to a virtual directory in the `/proc/` file system.

4.2 Protection Domain Deployment

In this section we describe how to adapt existing kernel programs to leverage primitives provided by SILVER. The first step is to establish the protection by declaring a specific LKM as a domain principal using the `pd_initialize()` routine, which will return an unique principal id. The module text range will be used to authenticate the principal during protection domain transition. Entry points of this domain need to be initialized by `pd_ep_create` API.

The second step involves modifying the declaration or creation of security-sensitive program data. There are four kinds of data object associated with a kernel program: global object, stack object, heap object and page object. For static data and stack data, SILVER could automatically recognize them and treat them private to their principal so that modification by other principals must be carried out by calling wrapper functions. For heap and page objects, developers could specify their security property to control how they could be accessed by other principals through calling `kmalloc_pd` and `__get_free_pages_pd` API with an integrity label. For example, unprotected memory sharing of low integrity data could be declared using the `GB_LOW` flag. Note that this process could be performed *incrementally* and *selectively*.

The next step is to handle data communication. The major task is to convert functions that handle exchange of high-integrity data to exploit transfer-based and service-based communication primitives. The example code below is a fragment of `alloc_skb` function that returns an allocated network buffer to NIC driver using transfer-based communication. By adding five lines of code at the end of the function, the owner principal of the `sk_buff` object changes accordingly.

```
out:
-  return skb;
+  if(is_protected(prid = get_caller_prid()))
+    transfer_skb = pd_transfer_object(skb, prid, PD_HIGH, sizeof(struct
sk_buff));
+  else
+    transfer_skb = pd_degrade_object(skb, GB_LOW);
+  return transfer_skb;
```

Service-based communication is used in a similar manner, the data proxying is accomplished by SILVER automatically, but the developer needs to register the function signature and mark the transferring parameter at both the beginning and the end of function using SILVER's APIs. To support reference validation, SILVER provides routine that automatically checks whether a designated parameter reference belongs to the caller principal.

We have converted a number of Linux kernel functions and extensions using SILVER's primitive to secure their interactions. The extensions include the Realtek RTL-8139 NIC driver, the CAN BCM module, a emulated sound card driver, and two kernel modules written by us for attacking experiments. For all cases, the total amount of modification incurs changing less than 10% lines of original code.

4.3 Security

In this section we evaluate the effectiveness of security protection provided by SILVER mechanism with both real-world and synthetic attacks.

Kernel SLUB overflow. In Section 2.1, we mention an exploit described by Jon Oberheide (CVE-2010-2959) to the vulnerable CAN Linux kernel module that achieves privilege escalation through overflowing dynamic data in the SLUB cache and corrupting crucial kernel control data in the same SLUB cache. We ported the vulnerable module to our Linux system, implemented and tested our exploit based on the attack code provided by Jon Oberheide. We then tested our attack in case the module is secured by SILVER's primitives, placing it in an untrusted domain separated from the Linux kernel. As result, dynamic data (e.g., `op->frames`) allocated by the CAN module are labeled with untrusted principal. According to SILVER's SLUB memory allocation scheme, these data object are placed on dedicated slabs for the untrusted CAN module principal, and they could never be adjacent to a high integrity kernel object `shmid_kernel` in the SLUB cache, despite any allocation pattern carried out by the attacker. For this reason, the attack can never succeed in our experiment. Moreover, in case the attacker successfully compromise the vulnerable kernel module (e.g., be able to execute injected code), it still cannot tamper the integrity of OS kernel since the entire kernel module can only exercise permissions of an untrusted principal.

Kernel NULL pointer dereference. The key idea of NULL pointer dereference is to leverage the vulnerability that a kernel module does not check whether a function pointer is valid before invoking that function pointer. As the result, the control will jump to the page at address zero, where the attacker maps a payload page containing the malicious code from user space before hand. Once get executed, the payload code could modify crucial kernel data or invoke kernel functions to achieve malicious goals such as privilege escalation. Such vulnerabilities are quite common in buggy extensions and even the core kernel code (CVE-2009-2692, CVE-2010-3849, CVE-2010-4258).

In our experiment, dereferencing a NULL pointer in a buggy untrusted module could not succeed in SILVER, primarily for two reasons. First, in SILVER, executing user-level code by an untrusted principal is prohibited according to access control rules. This is because NX bits are set for user pages in the untrusted principal HAP table. Second, even if the attack code got executed, it is still executed on behalf of untrusted principal with restricted permissions. As a result, attacking efforts such as privilege escalation (e.g., setting the `task->uid`, calling

the `commit_creds` function) would be intercepted by the reference monitor and the integrity of core OS kernel is preserved.

Attacks through Kernel API. In Section 2.1, we show that even with protection schemes like memory isolation or SFI, attackers can still compromise kernel integrity by launching confused deputy attacks over legitimated kernel APIs. Note that this kind of attacks is very rare in practice, for the reason that currently few Linux systems employ protection/sandboxing approaches inside OS kernel so that kernel attackers do not need to resort to this approach at all. To demonstrate SILVER's protection effectiveness against kernel API attacks, we implemented a kernel API attack module based on the RTL-8139 NIC driver. The attacking module provides a crafted reference of `struct pci_dev *` and uses it as input to the exported routine `pci_enable_device`. The reference is actually pointing to a calculated offset of the current process descriptor. By calling legitimate kernel API with such reference, the uid to current process will be set to 0 (root). SILVER prevents such attack by looking up the security property of the object referred by the actual pointer value. The reference monitor then detected that the caller principal actually does not owned the data object provided, and it raised an exception denying the attack attempt.

4.4 Performance Evaluation

In this section, we measure the performance overhead introduced by using SILVER's protection domain primitives. First, we would like to measure the time overhead of calling the extended or new APIs of SILVER by relying on a set of micro-benchmarks. Then we would like to use macro-benchmarks to measure the overall performance impact on throughput when a kernel NIC driver is contained. All experiments are performed on a HP laptop computer with a 2.4GHz Intel i5-520M processor and 4GB of memory. The VMM layer is based on Xen 3.4.2 with a Linux 2.6.31 Dom0 kernel. The OS kernel environment was configured as a HVM guest running Ubuntu 8.04.4 (kernel version 2.6.24.6) with single core and 512MB memory.

Run-time performance. Table 1 reports the microbenchmark results of selected APIs of SILVER. The first four rows denote the performance of the native Linux kernel SLUB memory allocator running on unmodified Xen. The fast path happens when the object requested is exactly available at the current `cpu_slab`. The rest of rows shows the performance of SILVER's dynamic data management primitives. There are three major sources of overhead added by SILVER's run-time system: (1) "context switch" between protection domains, (2) labeling a physical page through hypercalls, and (3) updating the object registry and data marshaling. Row 5 and 6 show the overhead of allocation and free when the caller is kernel itself, which only incurs overhead caused by (3). Row 7-8 show the overhead of calling `kmalloc_pd` and `kfree` by protection domains other than kernel. In this case, besides overhead (3), a protection domain switch (1) is also involved, and page labeling (2) happens occasionally when a new slab

is required. The relatively expensive guest-VMM switches in (1) and (2) make allocations/free operations by untrusted principals much more expensive.

To perform evaluation on application performance, we use SILVER to contain a `8139too` NIC driver, and leverage secure communication primitives to protect *all* of its object creation and data exchanges (skb pipeline) with the Linux kernel. We use the following macro-benchmarks to evaluate performance impact of SILVER towards different applications: (a) Dhrystone 2 integer performance; (b) building a Linux 2.6.30 kernel with defconfig; (c) apache ab (5 concurrent client, 2000 requests of 8KB web page) and (d) `netperf` benchmark (`TCP_STREAM`, 32KB message size, transmit). Figure 3 illustrates the normalized performance results compared to native Linux on unmodified Xen. We observed that our current SILVER prototype has a non-negligible overhead, especially in terms of throughput when system is loaded with saturated network I/O. This is primarily caused by very frequent protection domain switches and transfer-based communication. We measured protection domain switch rate of the `apache` test to be around $32,000$ per second. The overall performance also depends on how much data are specified as security-sensitive, how often security-sensitive data are created and the frequency of protected communication with untrusted principals. With SILVER, many of these security properties are controlled by the programmer so that she can manage the balance between security and performance. Hence, we expect SILVER to have better run-time performance in case of protecting only crucial data rather than the entire program. We also believe that our prototype can be greatly improved by optimizing Xen's VMEXIT and page fault exception handling to create a specialized path for SILVER's protection domain switch to avoid the unnecessary cost of VM switches.

Linux (Xen)	`kmalloc` SLUB fast path	$1.4\mu s$
	`kmalloc` SLUB slow path	$7.7\mu s$
	`kfree` SLUB fast path	$0.7\mu s$
	`kfree` SLUB slow path	$6.2\mu s$
SILVER (called by kernel)	`kmalloc`	$16.2\mu s$
	`kfree`	$14.4\mu s$
SILVER (called by other principal)	`kmalloc_pd` average	$56.7\mu s$
	`kfree` average	$64.1\mu s$

Table 1. Micro-benchmarks results for dynamic data management APIs of SILVER, average of 1000 runs. The data object size of allocation is 192 bytes.

5 Limitations and Future Work

Our current prototype has several limitations. First, for a few functions, we found difficulties in directly applying service-based communication on them, as they move complex data structures across function calls instead of transferring a single

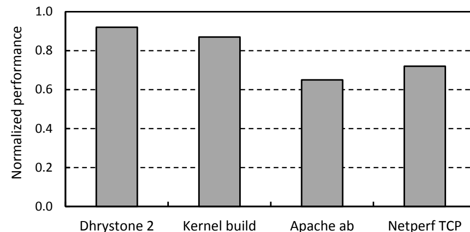

Fig. 3. Application benchmark performance, normalized to native Linux/Xen.

data object. Dealing with these functions may require us to manually write data marshalling routines. Fortunately, most of these functions are provided by the OS kernel, which usually configures as the parent domain of the caller principal and can directly operates on these data structures without data marshalling.

Compared with language-based and other static isolation approaches, SILVER's run-time mechanism is more accurate in resource tracking than static inference. However, our approach also has shortcomings for not providing verification and automatic error detection to programmers. For example, programmers must pay extra attention for not creating dangling pointers when using object transfer and endorsement primitives of SILVER, since these operations will release the original object in the same way as `kfree` function. We plan to incorporate kernel reference counting to help programmers manage their references of protection domain data objects. Moreover, adapting kernel programs to use SILVER requires certain understanding of security properties of their data and functions, and the entire procedure might be complex for converting very large programs. Hence, we also would like to explore automatic ways to transform an existing program to use SILVER given a security specification.

6 Related Work

In practice, protection domains are widely used for addressing security problems such as securing program extensions [11], privilege separation [29], implementing secure browsers [27], safely executing native code in a browser [11, 32] and mobile application deployment [1]. In this section, we review previous research efforts related to protection domains and OS kernel security, categorized by the approach to achieve their goals.

One major mechanism to achieve protection is through software fault isolation [7, 12, 26, 32], which rewrites binary code to restrict the control and data access of the target program. XFI [12] leverages SFI to enable a host program to safely execute extension modules in its address space by enforcing control flow integrity (CFI [5]) and data integrity requirements. While these approaches are efficient and effective for securing program extensions, they have difficulties for inferring and verifying system-wide resource and multi-principal access control rules in a static manner.

LXFI [19] is probably the closest related work with SILVER. It addresses the problem of data integrity and API integrity in SFI systems, using a completely different approach (compiler rewriting) than SILVER. Compared to LXFI, SILVER's run-time approach is more resilient to attacks that fully compromise an untrusted module and execute arbitrary code. Moreover, security enforcement of SILVER is more tamper-proof since the isolation and access control are carried out by the hypervisor.

Run-time protection approaches are mostly achieved by access control mechanisms to constrain the behavior of untrusted programs. Depending on the abstraction and granularity levels, these approaches mediate security-sensitive abstractions ranging from segmentation [10, 14, 32] and paging protection [25], to system call interposition [11, 15]. These events are regulated by a set of access control policies. Traditional mandatory access control systems such as SELinux [3] are inflexible and difficult to configure fine-grained policies because the internal state of an application is difficult to infer externally. In contrast, capability-based systems [23, 29] and DIFC systems [17, 33] delegate part of security decisions to application developers, which eases the burden of administrators for setting up complex system policies externally and allows applications to have its own control of data and communication security. Flume [17] provides DIFC-based protection domain to user applications in Linux at the granularity of system objects such as processes and files. SILVER's security model follows a similar spirit of these approaches, yet it enforces protection for kernel programs at data object granularity.

Many research efforts are focused on improving the reliability of operating system kernels. Micro-kernel OSes [8, 16, 18] removes device drivers from kernel space and execute them as userspace server applications. However, as discussed in Section 1, despite their elegant design, it is generally difficult to retrofit these approaches in commodity OSes. Mondrix [30] compartmentalizes Linux and provides fine-grained isolation, but it requires a specific designed hardware. Nooks [25] is a comprehensive protection layer that leverages hardware protection to isolate faulty device drivers within Linux kernel and recover them after failures. Since its primary focus is fault resistance rather than security, it does not address attacks such as manipulating architectural state. Also, Nooks does not provide the flexibility to specify security properties of individual data.

SILVER leverages a VMM as another layer of indirection to mediate cross-protection-domain activities. VMMs are also widely used for protection systems to enhance the security of applications and the OS kernel. Overshadow [9] and TrustVisor [20] protect the integrity and secrecy of an application even in case that the OS kernel is compromised. SIM [24] uses hardware virtualization for securely running an isolated and trusted monitor inside an untrusted guest. Secvisor [22] and NICKLE [21] are hypervisor-based systems which guarantee that any unauthorized code will not be executed in the operating system kernel. Hooksafe [28] protects kernel control data (i.e., hooks) from being tampered by kernel-level rootkits. In comparison, SILVER aims to provide a more compre-

hensive protection with the integrity guarantee of both code, data and control flows.

7 Conclusions

In this paper, we have described the design, implementation and evaluation of SILVER, a framework to achieve transparent protection primitives that provide fine-grained access control and secure interactions between OS kernel and untrusted extensions. We believe that SILVER is an effective approach towards controlled privilege separation, by which developers could protect their programs and mitigate the damage to OS kernel caused by attacks exploiting a vulnerability in untrusted extensions.

Acknowledgements

We would like to thank our paper shepherd Andrea Lanzi, the anonymous reviewers and Trent Jaeger, for their helpful comments on earlier versions of this paper. This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, AFOSR W911NF1210055, and ARO MURI project "Adversarial and Uncertain Reasoning for Adaptive Cyber Defense: Building the Scientific Foundation".

References

1. Android: Security and Permissions. http://developer.android.com/guide/topics/security/security.html.
2. Linux kernel can slub overflow. http://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/.
3. NSA. Security enhanced linux. http://www.nsa.gov/selinux/.
4. The SLUB allocator. http://lwn.net/Articles/229984/.
5. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *CCS '05*, 2005.
6. K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, The Mitre Corporation, 1977.
7. M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-granularity Software Fault Isolation. In *SOSP '09*, 2009.
8. J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Trans. Comput. Syst.*, 12:271–307, 1994.
9. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS '08*, 2008.
10. T.-c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *SOSP '99*, 1999.

11. J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *OSDI'08*, 2008.
12. U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI '06*, 2006.
13. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys '06*, 2006.
14. B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX ATC*, 2008.
15. T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS'04*, 2004.
16. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP '09*.
17. M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *SOSP '07*, 2007.
18. J. Liedtke. On Micro-kernel Construction. In *SOSP '95*, 1995.
19. Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP '11*, 2011.
20. J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.
21. R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08*.
22. A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP '07*, 2007.
23. J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a Fast Capability System. In *SOSP '99*, 1999.
24. M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *CCS '09*, pages 477–487, New York, NY, USA, 2009. ACM.
25. M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*.
26. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *SOSP '93*, 1993.
27. H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *USENIX Security '09*, 2009.
28. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *CCS '09*.
29. R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security'10*, 2010.
30. E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *SOSP '05*, 2005.
31. X. Xiong, D. Tian, and P. Liu. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. In *NDSS'11*, 2011.
32. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *IEEE Symposium on Security and Privacy*, 2009.
33. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI '06*, 2006.