Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege

Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens

iMinds-DistriNet - KU Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium
niels.avonds@student.kuleuven.be,
firstname.lastname@cs.kuleuven.be

Abstract. Consumer devices are increasingly being used to perform security and privacy critical tasks. The software used to perform these tasks is often vulnerable to attacks, due to bugs in the application itself or in included software libraries. Recent work proposes the isolation of security-sensitive parts of applications into protected modules, each of which can only be accessed through a predefined public interface. But most parts of an application can be considered security-sensitive at some level, and an attacker that is able to gain in-application level access may be able to abuse services from protected modules.

We propose Salus, a Linux kernel modification that provides a novel approach for partitioning processes into isolated compartments. By enabling compartments to restrict the system calls they are allowed to perform and to authenticate their callers and callees, the impact of unsafe interfaces and vulnerable compartments is significantly reduced. We describe the design of Salus, report on a prototype implementation and evaluate it in terms of security and performance. We show that Salus provides a significant security improvement with a low performance overhead, without relying on any non-standard hardware support.

Keywords: Privilege separation, principle of least privilege, modularization.

1 Introduction

Both desktop and mobile devices are increasingly being used to perform security and privacy critical tasks, such as online banking, online tax declarations and e-commerce in general. The software to perform these tasks either runs inside a web browser, or is written as a standalone application. In both cases, the software is often vulnerable to attacks, either due to bugs in the application itself or due to bugs in included software libraries or in the runtime environment used to execute the application (e.g. the browser).

Because of their widespread use and potentially high-impact nature, such applications form an interesting target for cybercriminals. A lot of research has focused on defending against specific attack vectors such as buffer overflows[1,2,3,4], format string vulnerabilities[5] and non-control-data attacks[6]. Even though

T. Zia et al. (Eds.): SecureComm 2013, LNICST 127, pp. 252–269, 2013.

many of these defense mechanisms are applied in practice, successful attacks against high-value applications are still common.

To provide stronger security guarantees, research efforts have shifted to the isolation of small, security-sensitive parts of applications such as cryptographic libraries. By relying on hardware support for trusted computing, state-of-the-art research prototypes are able to achieve such isolation with a very small trusted computing base, in some cases even excluding the running kernel[7,8,9,10] or even having a zero-software TCB[11]. Recent work[12] has proven that such platforms can effectively isolate sensitive information in protected modules from the rest of the system; an in-process or in-kernel attacker is only able to interact with a module through its predefined interface. Hence, an attacker that has compromized a non-security-sensitive part of the application can still only perform the actions explicitly allowed by the interface of a security-sensitive part of the application.

In practice however, isolating security-sensitive parts of an application is difficult as most program logic can be considered security-sensitive at some level[13]. A too coarse-grained approach will result in bloated modules that may contain vulnerabilities and that are too big to be formally verified[14]. Minimum-sized modules on the other hand, can provide strong and easily verifiable guarantees, but may need to expose insecure interfaces to interact with other modules. This is a common problem of module-isolating security platforms, both in software as in hardware. For instance, in the recent DigiNotar attack, the root CA's private cryptographic key was safely stored in a hardware security module (HSM), but its insecure interface enabled attackers to sign arbitrary certificates.

In order to improve upon these shortcomings, we acknowledge that almost every part of an application performs security-sensitive operations. To reduce chances of a successful attack, we propose to partition the *entire* application into compartments and implement a non-hierarchical access control mechanism between compartments. Compartments not only provide provable secure isolation of stored private data (as modules in related work do), but are also able to confine software vulnerabilities to the compartments they occur in by restricting the types of system calls that they are allowed to perform. In addition, caller/callee authentication is able to reduce the impact of insecure interfaces. By separating likely attack vectors from attack targets and placing them into different compartments, an attacker has to find a vulnerability in multiple compartments to reach her goal.

Consider, as an example, a certificate signing application consisting of a parser, a validator, a signer and a logging component (Figure 1). When run as a single monolithic application, a vulnerability in any one of these components can lead to the compromise of the entire application. When placing each of these components in a separate compartment under Salus, components can only call each other through their well-defined interfaces and each component can authenticate both its callers and its callees. This restricts the flow of data and control between compartments to predefined patterns, which significantly raises the bar for an attacker, since she would need to exploit multiple vulnerabilities in different

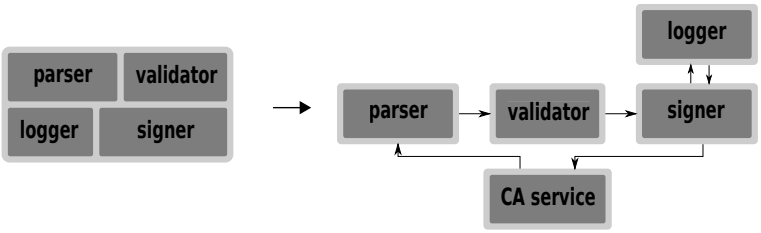

Fig. 1. Salus' compartmentalization enables strong isolation of security-sensitive data *and* contains possibly vulnerable code. Multiple vulnerable compartments need to be exploited to attack the system successfully.

components of the system in order to perform a successful attack. Furthermore, by restricting the types of system calls that can be made from each compartment, the impact of a successful attack is reduced.

Concretely, we make the following contributions in this paper:

- We present a novel approach for partitioning processes into compartments with support for strong isolation of sensitive data *and* containment of vulnerabilities. To the best of our knowledge, Salus is the first solution that simultaneously (1) reduces the impact of insecure compartment interfaces, (2) enables compartments to restrict the types of system calls they are allowed to perform and (3) executes compartments in same address space.
- We report on a prototype implementation of Salus in the Linux kernel.
- We evaluate the security of our approach and the performance of our prototype.

The remainder of this paper is structured as follows: in Section 2 we define our attacker model and describe our desired security properties. In Section 3 we provide a high-level overview of Salus, before presenting our prototype implementation in Section 4. Finally we evaluate our approach in Section 5, discuss related work in Section 6 and conclude in Section 7.

2 Attacker Model and Security Properties

We consider an attacker with the ability to inject and execute arbitrary code in a process, for instance by exploiting a buffer-overflow vulnerability. We assume the application under attack takes advantage of Salus' protection mechanism by authenticating caller and callee on each intercompartmental call and by restricting the possible system calls to those strictly required. Salus must protect against such an attacker in the following way:

- The exploitation of a compartment must not affect the security of compartments other than those that explicitly trust the compromised compartment, in the sense that an attacker should be able to interact with those trusted

compartments only through their public interface. A compartment trusts another compartment when it is a caller or callee. Although this objective does not protect against abuse of poorly designed interfaces, Salus provides application developers with the primitives required to create secure compartment interfaces.
– Attackers are explicitly allowed to create new compartments. There is thus no guarantee that compartments requesting protection can be trusted. Hence, Salus must isolate compartments from one stakeholder from those of another, possibly malicious, stakeholder.
– An attacker should not be able to execute system calls that have been revoked.

Kernel-level and physical attacks are considered out of scope. Regarding the cryptographic primitives used, we assume the standard Dolev-Yao model[15]: An attacker can observe, intercept and adapt any message. Moreover, an attacker can create messages, for example by duplicating observed data. However, the cryptographic primitives used cannot be broken.

3 Overview of the Approach

This section presents a high-level overview to Salus. Section 3.1 describes the memory access control mechanisms on which Salus is based. Section 3.2 presents the services Salus provides to protected applications. Section 3.3 shows how these services are used in a typical life cycle of a compartmentalized application. Finally, section 3.4 describes how two compartments can securely communicate with each other.

3.1 Compartments of Least Privilege

Structure of a Compartment The basic layout of a compartment, shown in Figure 2, is a virtual memory region divided into two sections: a public section and a private section. The *public* section contains the compartment's code and any data that should be read accessible by other compartments of the same application. This section can never be modified after initialization, which enables other compartments to authenticate the compartment based on a cryptographic hash of the public section (see Section 3.4). The start of the functions that make up the compartment's public interface are marked as entry points. Execution of the compartment can only be entered through these memory locations (see Table 1).

The *private* section contains the compartment's private data, which consists of application-specific security-sensitive data (e.g. cryptographic keys) as well as data relevant to the correct execution of the compartment, such as the runtime call stack. The data in the private section is read and write accessible from within the compartment, but completely inaccessible for code executing outside of the compartment. Note that since each compartment has its own private call

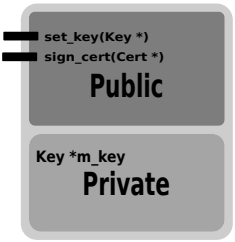

Fig. 2. Salus' memory access control model enables the creation of compartments that provide strong isolation guarantees to sensitive data. Secure communication primitives reduce the impact of an insecure interface.

Table 1. The enforced memory access control model enforces, for example, that a compartment's private section (4^{th} column) can only be read-write accessed from the public section of the same compartment (3^{rd} row)

from\to	Entry pnt.	Public section	Private section	Unprot. mem.
Entry pnt.	`---`	`--x`	`---`	`---`
Public section	`r-x`	`r-x`	`r-w`	`rwx`
Private section	`---`	`---`	`---`	`---`
Unprot. mem/ other compartment	`r-x`	`r--`	`---`	`rwx`

stack, intercompartmental function call arguments and return addresses must be passed via CPU registers (as opposed to passing them using the runtime stack).

Applications can still have a memory region that is not part of any compartment. This region is termed *unprotected memory* and is read/write accessible from any compartment. All compartments of the same application run in the same address space, which facilitates the compartmentalization of legacy applications. Nonetheless, fine-grained compartmentalization of a large code base can still require significant developer effort. Therefore, Salus enables applications to be compartmentalized incrementally by storing code and/or data in unprotected memory. While unprotected memory does not provide any of the security guarantees of compartments, it does provide an incremental upgrade path for legacy applications.

As an example of a compartment, consider a single compartment providing a certificate signing service (see Figure 2). The compartment provides two functions as part of its public interface. The first function, `set_key`, allows setting the cryptographic key used to sign certificates. This key is stored as the `m_key` variable in the private section. The second function, `sign_cert`, handles the actual signing requests. Salus' memory access control model ensures that only these two functions are executable; any attempt to jump to another memory location in the compartment will fail. Similarly, any attempt to directly read or write the cryptographic key in the private section from unprotected code or from another compartment will be prevented. Only after calling a valid entry point

will read and write access to the private section be enabled, making the cryptographic key only accessible while the compartment is being executed. When the function is terminated, execution returns to the caller and read/write access to the compartment's private section will again be disabled.

Special care is required when execution returns to a compartment after a call to another compartment. Execution must resume at the return location, which is the instruction right after the call instruction in the caller compartment. This location however does not typically correspond to an entry point and hence would cause a memory access violation according to Salus' memory access control model (see Table 1). Compartments can implement a *return entry point* to avoid this access violation. Right before calling another compartment, the return location is placed on the top of the calling compartment's private stack. When the intercompartmental call has finished, execution flow jumps to the return entry point where the return location is retrieved from the stack and jumped to. Note that a return entry point is a software implementation and follows the same access rights as any other entry points.

Restriction of Privileges. Salus provides two important primitives to limit the impact of a compromised compartment. The first primitive is caller and callee authentication. By authenticating callers and callees, a compartment can limit its interaction to trusted compartments only. Although this does not protect against trusted compartments that have been compromised, it does significantly limit the capabilities of an attacker after a successful exploit. Moreover, compartments can dynamically adjust their trust relations to other compartments. For instance, the certificate signing compartment introduced in the previous section (Figure 2) could restrict communication to the compartment that last set its cryptographic key. Secure communication between compartments is discussed in more detail in Section 3.4.

The second primitive allows compartments to disable specific system calls for any code executed from within their public section. Once a system call is disabled, it cannot be re-enabled. By carefully partitioning an application into compartments, each of which should disable any system call it doesn't need, the impact of the exploitation of a vulnerable compartment is minimized. Note that much more fine-grained solutions exist than restricting complete system calls[16]. However, we focus on providing strong compartmentalization primitives that can be used as a building blocks for finer-grained privilege restriction mechanisms.

3.2 Provided Services

To enable compartmentalization of applications, Salus provides runtime support of the following services:

Create. After code is loaded into memory, this service can be used to create a new compartment. Given a memory location and size for the compartment to create, Salus will enable memory protection for this region and will return a system-wide unique ID for the new compartment.

Destroy. A compartment can only be destroyed by the compartment itself. After destruction, the memory access protection is disabled. Hence, a compartment should overwrite any private data before destruction.

Request compartment ID. To support secure communication, Salus provides a service to request the ID and layout (i.e. the size and locations of the public and private sections and the available entry points) of a compartment covering a given memory location. If there is no compartment at the specified location, the service returns an error code. This service is used as a primitive in compartment authentication.

Request caller ID. To support caller authentication, Salus provides a service to request the ID and layout of the compartment that called an entry point of the current compartment.

Disable system call. To limit the impact of the exploitation of a compartment, unused system calls can be disabled. Once a system call is disabled, it cannot be re-enabled. To prevent an attacker from gaining system call privileges by creating a new compartment, compartments inherit system call privileges from their parent.

3.3 Life Cycle of a Compartmentalized Application

Compartmentalized applications can be started as any other application. After the (trusted) operating system or loader loads the application into memory and starts its execution, the application can create the required compartments. Finally, execution can jump to the compartment containing the application's main function. Compartments can be created at any point during the applications' execution, for example, at the time a new (compartmentalized) plugin is loaded.

Creation of Compartments. As the first step of setting up a new compartment, the application allocates (unprotected) memory and loads the compartment's code. Next, the application enables protection of this memory region, by calling Salus' creation service. Note that there is no guarantee that the new compartment's code has been loaded correctly into memory, since the creator might have been compromised already. However, any tampering with the code will be detected when the compartment tries to communicate with another compartment, as will be explained in Section 3.4.

When a new compartment is created, Salus clears the first byte of the private section. This serves as a flag to indicate to the compartment that it should initialize itself when its service is first requested. As part of its initialization, a compartment should clear the private memory locations it will use. This prevents an attacker from crafting a private section by setting it up in unprotected memory locations where a new compartment will later be created. Initialization code typically also disables the system calls that will not be used during further execution of the compartment.

Destruction of Compartments. A compartment can *only* be destructed by the compartment itself. This ensures that compartments can clear their private

section (which may contain sensitive data), before the memory protection is lifted. In addition, trusted communication endpoints could be notified of the compartments' imminent destruction. After destruction, the unprotected memory area of the destructed compartment can be freed.

3.4 Secure Communication

Salus' memory isolation mechanism provides strong guarantees that sensitive data in the private section can only be accessed by code in the public section[12]. Reconsidering our certificate signing as an example (see Figure 1), we can prove that the signing key will never leave its compartment. But an attacker with access to the compartment's interface is still able to sign arbitrary certificates. Salus limits the feasibility of such attacks by enforcing both caller as callee authentication. The signing compartment, for example, may enforce that it can only be accessed by the validator compartment. Likewise, the validator authenticates the signing compartment to verify that it hasn't been tampered with before its memory protection was enabled.

Security Report. Authenticating a compartment consists of verifying whether that compartment adheres to a trusted *security report* of that compartment. A security report of a compartment consists of:

The cryptographic hash of its public section This allows any code to verify that the public section of the compartment has not been tampered with: the cryptographic hash should be recalculated at runtime and be compared to the known-good value stored in the security report. This protects against an attacker who is able to modify the public section of a compartment during its creation, before memory protection is enabled (see Section 3.3).

The layout of the compartment When a creation request originates from unprotected memory, the request itself may have been tampered with. An attacker could, for instance, specify an incorrect layout for the compartment to create. This may result in the use of unprotected memory that should be under Salus' protection. By storing the known-good layout of the compartment in the security report, any code can verify that the layout was not tampered with during creation of the compartment.

A cryptographic signature In order to have integrity protection and authentication of the security report, it is digitally signed by its issuer. Each compartment can decide independently whether or not to trust a certain issuer, which opens up the opportunity to integrate compartments from different parties into a single application. Since the cryptographic signature provides integrity protection, security reports can be placed in unprotected memory.

Authentication of Called Compartments. When exchanging sensitive information between compartments, caller and callee must authenticate each other *before* sensitive data is exchanged.

To authenticate a compartment to be called, its ID must first be obtained using Salus' 'request compartment ID' service. Next, the callee's security report must be acquired. For this a central service where each compartment registers to on initialization, can be used. Given the callee's ID, the service should return the (location of the) corresponding security report. Note that this service need not be trusted, as any tampering with the information returned will be detected during the next steps. Once the security report has been obtained, it should be validated by checking the cryptographic signature and by checking that the issuer is trusted. Each compartment should contain a private list of trusted security report issuers. Next, the callee compartment's layout should be requested from Salus and a hash of the Public section should be calculated. The layout and the hash must be compared to the values listed in the security report. This completes the authentication and allows the caller to securely call one of the callee's public functions.

When calling a compartment that has already been authenticated in the past, a re-validation must occur because the callee may have been destroyed since the last interaction. A full authentication using the security report on every call would be very time consuming, so to reduce the performance impact, Salus allows compartments to be re-authenticated quickly based on their ID. Salus ensures each compartment has an ID that is unique on the system until the next reboot. Hence, a re-authentication can simply consist of requesting the ID of the compartment to be called (using the 'request compartment ID' service) and checking that it is the same as during the initial authentication. Using unique identifiers has the added benefit that code can distinguish between different instances of the same compartment.

Authentication of Calling Compartments. To enable compartments to limit use of their (possibly insecure) interface to trusted caller compartments, Salus provides primitives for caller authentication. For a compartment to authenticate its caller, it can first request the caller's ID and memory location (using the 'request caller ID' service) and proceed to authenticate the caller similarly as described above.

4 Implementation

Access rights to compartment sections depend on the value of the program counter. For instance, only if execution is in the public section of a compartment, will the private section of that compartment be read/write accessible. This program counter-based memory access scheme is at the core of Salus' protection mechanism. Enforcing this scheme purely in software would have a huge performance impact as every memory access has to be checked. A pure hardware implementation of the scheme is possible[11], but prohibits its use on commodity, off-the-shelf PC platforms. The approach taken for Salus combines the best of both alternatives, by using the key insight that memory access rights for compartments only need to change when execution crosses a compartment border.

This allows Salus to use the standard memory management unit (MMU) to enforce the memory protection scheme.

A prototype for Salus has been implemented as a Linux kernel modification. Section 4.1 describes how the program counter-based access control mechanism is implemented in this prototype. Section 4.2 describes the API Salus provides to processes and finally Section 4.3 lists the Linux system calls that had to be modified in order to provide a secure implementation of the protection mechanism.

4.1 Program Counter-Based Access Control

By aligning compartment sections to pages, the standard MMU found on any recent commodity computer can be applied to enforce the required memory protection scheme. After a compartment is created (e.g. from unprotected memory), the MMU access rights for the pages of the new compartment are set up according to Table 1: the public section is world-readable while the private section is isolated completely.

When execution tries to enter a compartment (e.g. because of a call instruction), a page fault is generated by the MMU. Based on the memory location addressed and the access type (read, write or execute), Salus determines whether a valid entry point was called and, if necessary, modifies the access rights of only the public and private sections, according to Table 1. This minimizes the number of page faults and access right modifications, thereby reducing the overall performance impact.

Because unprotected memory is always readable, writable and executable, no page fault is generated when execution returns from a compartment to unprotected memory. To restore the access rights of the exited compartment, the compartment itself must issue a system call to Salus.

The Linux page fault handler was modified to implement these access right modifications. To keep track of a process' compartments, the Linux process descriptor data structure was extended with a list of `comp_struct` structures. Each `comp_struct` describes a single compartment and contains:

- The (virtual) start address and length of the public and private sections
- The compartment's unique ID
- The compartment's saved stack pointer
- A list of the compartment's remaining system call privileges

4.2 System Call API

The following new system calls were implemented in the Linux kernel. These system calls represent the API Salus provides to processes.

`void salus_create(void* start, uint len_pub, uint len_priv)` Before a new compartment is created, the list of existing compartments is checked to ensure that the new compartment will not overlap with any existing ones.

New compartments must also not overlap with the kernel or have their memory pages mapped to files. When these checks succeed, a new compartment is created and added to the current process' compartment list. It receives the same system call privileges as its parent.

`void salus_destroy(void)` Since compartments can only be destroyed from within their own public section, this system call does not require any arguments. This system call restores the original memory access rights on memory region occupied by the executing compartment and then removes the compartment from the current process' compartment list.

`struct comp_layout* salus_layout(void* addr)` This system call returns the ID and memory layout of the compartment covering a given memory location. It can be implemented by simply iterating over the current process' compartment list until a matching compartment is found. A `null` pointer is returned when there is no compartment covering the given address.

`struct comp_layout* salus_caller(void)` This system call returns the ID and memory layout of the compartment that last called an entry point of the current compartment. A `null` pointer is returned when the current compartment was last called from unprotected memory.

`void salus_syscall_disable(uint syscall_id)` This system call disables further use of the specified system call, by removing it from the list of system call privileges in the `comp_struct` of the current compartment. Once a system call is revoked, it cannot be re-acquired.

`void salus_return(void* addr)` Before execution returns from a called compartment back to its caller (i.e. unprotected memory or another compartment), the access rights of the called compartment's pages need to be restored. This system call performs this access rights modification and then continues execution at the specified address.

4.3 Conflicting System Calls

Some existing system calls in the Linux kernel conflict with Salus' compartmentalization. Additional security checks had to be inserted for these conflicting system calls.

mprotect. The `mprotect` system call can be used to change the access rights of pages in memory. Additional checks were added to prevent this system call from modifying the access rights of compartments.

mmap. Existing system calls such as `mmap` or `mremap` modify the virtual address space of a process. An attacker could abuse these system calls to map a compartment's private section to a file, for instance. Additional checks were added to prevent this type of abuse.

personality. In Linux, each process has a *personality*, which defines the process' execution domain. The personality includes, among other settings, a `READ_IMPLIES_EXEC` bit, which indicates whether read rights to a memory region should automatically imply executable rights as well. For compartments this would result in world-executable public sections, nullifying the

use of designated compartment entry points. Therefore, Salus enforces that this bit is disabled for compartmentalized processes.

fork. The `fork`, `vfork` and `clone` system calls can be used to create a new process or thread. As these processes or threads share parts of their page tables, the elevated access rights of the private section of a called compartment, affects all processes/threads and enable its access from unprotected memory. While these system calls could be modified to create copies of the page tables, our prototype currently uses Linux' existing `CLONE_VM` and `VM_DONTCOPY` flags to prevent compartments being mapped in the new process or thread. Checks were also added to the `madvice` system call, since it can be used to modify the `VM_DONTCOPY` flag.

5 Evaluation

The effectiveness of Salus' protection mechanisms is evaluated in Section 5.1 and its performance impact is discussed in Section 5.2.

5.1 Security Evaluation

To evaluate Salus' security, we make a distinction between memory-safe and memory-unsafe compartments. A memory-unsafe compartment can be exploited by an attacker using low-level attack vectors such as buffer overflows[1,2,3,4], format string vulnerabilities[5] or non-control data attacks[6]. A memory-safe compartment does not contain such vulnerabilities, for instance because it was written in a memory-safe language or simply because the compartment doesn't contain any memory-safety bugs.

Since memory-safe compartments cannot be exploited directly, the only attack vector against them is through exploitation of another compartment in the same address space. However, recent research[12] has shown that memory protection mechanisms such as those offered by Salus, are able to provide full source code abstraction. This means that, even when other compartments have been successfully exploited, an attackers' capabilities are limited to interacting with the memory-safe compartment through its public interface. A carefully constructed interface can thus effectively limit the attack surface of a compartment. But in many cases, creating a secure interface is still a challenging problem[17]. Recall the example of a certificate signing compartment introduced in Section 3.1: even if the private cryptographic key is never exposed, an attacker could potentially still use the compartment's interface to sign arbitrary certificates[18]. By taking advantage of Salus' support for caller/callee authentication however, the risk of such an attack can be minimized by only servicing requests from compartments that would issue them as part of the normal operation of the application (e.g. in Figure 1, the signer compartment should only accept requests from the validator compartment).

Memory-unsafe compartments may still contain vulnerabilities that can be exploited by attackers. Even though Salus does not prevent such attacks, compartmentalization can still provide significant security benefits. Firstly, high-risk

components can be identified and be placed in separate compartments. Effective but high-overhead countermeasures[19,20] can be used to harden such compartments. By only applying these countermeasures to likely vulnerable compartments, their performance impact remains limited.

Secondly, compartmentalization can automatically thwart certain types of attacks. For instance, limiting entrance of compartments to valid entry points significantly reduces the chance of an attacker finding enough gadgets to successfully execute a return-oriented-programming (ROP) attack[21,22].

Thirdly, compartmentalization can be used as a building block for new countermeasures. For instance, a custom loader could be implemented that loads compartments at different locations in memory for every program execution. This is similar to address space layout randomization (ASLR)[23], but can be applied at a much finer-grained level.

Finally, even when a compartment has been successfully exploited, Salus can still limit the impact of the attack. Because Salus provides entry point enforcement, caller/callee authentication and system call privilege containment, an attacker will likely have to compromise multiple vulnerable compartments before reaching her intended target. This significantly increases the effort an attacker must take to successfully exploit the application. The ability to confine attackers to the exploited compartment even allows implementing a tightly controlled sandbox where user-provided machine code can be executed safely.

5.2 Performance Evaluation

To evaluate the performance of Salus, we performed micro- and macrobenchmarks. All tests were run on a Dell Latitude E6510. This laptop is equipped with an Intel Core i5 560M processor running at 2.67 GHz and contains 4 GiB of RAM. A Ubuntu Server 12.04 distribution with (modified) Linux 3.6.0-rc5 x86_64 kernel was used as the operating system.

System-Wide Impact. To show that legacy applications not using the modularization technique are not impacted by our changes to the Linux kernel, we ran the SPECint 2006 benchmark. All tests finished within ±0.4% compared to the vanilla kernel.

Microbenchmarks. To measure the overhead caused by switching the access rights, we created a microbenchmark that measures the cost of a call to a secure compartment and compare it to the cost of calling a regular function and calling a system call. The compartment used in the benchmark immediately returns to the caller. The system call and function behave similarly.

Table 2 displays the results of this microbenchmark. Calling a compartment is about 677 times slower compared to calling a regular function. This overhead is attributed to the need to modify the access rights of pages. Compared to calling a system call, the compartment is only 20 times slower. Due to these high costs, there is a trade-off to be made between a low number of compartment transitions and small compartments with additional security guarantees.

Table 2. Compartment access overhead

Type	CPU cycles	Relative
Function Call	5,944	1
System Call	193,970	32.63
Compartment Call	4,024,227	677.02

Secure Web Server. As a macrobenchmark, we compartmentalized an SSL-enabled web server. For every new connection a new compartment is created, securing session keys even in the event that an attacker is able to inject shellcode in the compartment providing its own SSL session.

The secure compartment was built using the PolarSSL cryptographic library and a subset of the diet libc library. A simple static 74-byte page is returned to the clients over an SSL-connection protected by a 1024-bit RSA encryption key.

Table 3. Requests per second of an SSL-enabled webserver where every SSL session is protected in its own compartment, for an increasing number of clients

Concurrency	Vanilla kernel	Salus kernel	Relative perf.
1	109.11	96.54	-11.52 %
2	165.56	153.62	-7.21 %
4	184.31	164.78	-10.60 %
8	199.98	175.35	-12.32 %
16	206.82	181.00	-12.48 %
32	207.78	181.50	-12.65 %
64	206.64	180.35	-12.72 %
128	206.49	180.97	-12.36 %

We used the Apache Benchmark to benchmark this web server for an increasing number of clients that are concurrently requesting pages. The results are shown in Table 3. The performance overhead tops at 12.72% and is mainly attributed to the many compartment boundaries crosses during the SSL negotiation phase.

Compartmentalized Parser. As input files are often under the control of an attacker and sanitation of their content can be difficult, parsers are a likely attack vector for many applications. As a second benchmark, we isolated the decompressing function of gzip (GNU zip). While disabling unused system calls for the entire process would result in similar security guarantees, we are interested in the impact of repeated compartment crossings in a parser setting. Applications that place their parser and the rest of the application in different compartments, would incur a similar overhead as only one additional compartment boundary needs to be crossed.

To benchmark the application, we created input files with randomized content, ranging from 16 KiB to 64 MiB in size, compressed them and measured the time taken to decompress the files with the hardened application. The application was run 100 times on each file. File I/O used a buffer of 32 KiB and the output was redirected to the null device. Figure 3 displays the results.

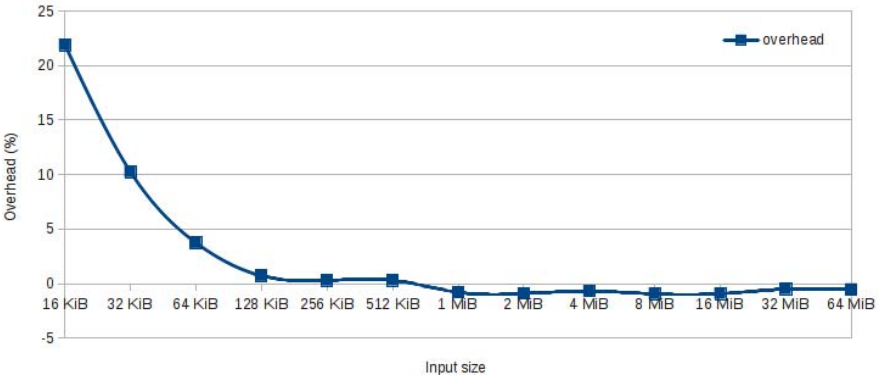

Fig. 3. Salus' performance overhead on the gzip macro benchmark drops significantly as the input file size increases

Given the relatively high overhead of a call to a compartment and the low computation cost of the decompressing function, it is unsurprising that for small input files the overhead can be as high as 21.9%. When the input size is increased however, the overhead drops steadily to -0.5% for 64 MiB input files, even though also the number of compartment-border crossings increases from 8 to 8200. We attribute this significant drop in overhead to the increased amount of slow disk I/O that needs to be performed as the input file size gets bigger, an effect that we predict to see in most parser-like compartments. The small performance gain of 0.5% can be attributed to cache effects.

The way an application is partitioned will have a significant impact on performance. Applications should be compartmentalized in logical blocks where each compartment has direct access to most of its required data. Once a logical block has finished, control and all data should be passed to the next compartment, reducing the number of inter-compartment calls. Smaller, heavily protected compartments such as an SSL compartment, provide strong security but may impact performance more significantly when called repeatedly. This makes the performance impact of compartmentalization difficult to predict. Therefore we advocate for automatic partitioning tools that reduce the number of compartment crosses and help the programmer decide which compartments should be hardened most thoroughly.

6 Related Work

Various security measures have been proposed to harden applications. Many of them aim to protect against very specific vulnerabilities such as buffer overflows

[1,2,3,4], format string vulnerabilities[5] or non-control data attacks[6]. While these countermeasures make it significantly more difficult for an attacker to compromise software applications, they cannot offer complete protection. Static verification of source code[24], in contrast, is able to provide such hard security guarantees, but typically comes at a significant economic cost in terms of programming and verification effort.

Singaravelu et al.[13] proposed to isolate security-sensitive parts of applications in complete isolation from the rest of the system. Many research proposals have since been filed based on this principle. Each of them provides some way of executing modules in isolation, relying on a trusted code base ranging from only a few thousands of lines of code[8,10] to only the protected modules themselves and a small runtime library[7,9]. While these research prototypes offer provable security to the sensitive data that they protect[12], they do not attempt to reduce the impact of a vulnerability elsewhere in the code by executing modules with the least amount of privileges possible[25]. An attacker who successfully gains control over the platform is still able to interact with protected modules unrestrictedly.

Other work focuses on confining possible software vulnerabilities. Early work focused on reducing the size of the kernel itself[26], where process privileges are managed by capabilities. Recently Watson et al.[16] proposed applying a similar idea to partition applications themselves, where capabilities can be granted to each created partition. As partitions live in their own process, interaction takes place through remote procedure calls and pointers cannot to passed directly. Salus avoids these drawbacks by executing compartments in the same address space and unprotected memory can be used to gradually partition legacy applications. While fine-grained privilege containment is out of scope for this paper, Salus can easily be extended with a capability mechanism.

Native Client (NaCl)[27,28], which builds upon the concepts of software fault isolation[29], takes another approach and attempts to completely sandbox x86 code. Accesses to the environment from within a sandbox are tightly controlled by runtime facilities. While NaCl focuses on downloaded, untrusted binary code, it could be used to partition entire applications. Interaction between two NaCl partitions is provided through a service similar to Unix domain sockets, making porting existing legacy applications a challenging undertaking. Salus on the other hand can provide a similar tightly controlled sandbox by placing such partitions in one compartment while the remaining legacy application is placed in another. A specially created wrapper can ensure that all system call privileges are revoked before execution control is given to the sandboxed code. There are however two major differences compared to NaCl. First, Salus only impacts performance when compartment boundaries are crossed. NaCl on the other hand places constraints on the binary code itself, resulting in a varying performance impact. Second, Salus employs a non-hierarchical separation of privilege, allowing compartments to be completely isolated from other compartments (possibly provided by other vendors) while compartments of the same vendor can co-operate easily.

Finally, our earlier work[30,10] is the most related to Salus. It also employs a program-counter based access control mechanism, but assumes a safe interface. Therefore it has the same limitation as other research prototypes[9,7,8] that provide strong isolation of sensitive data: it does not reduce the possible impact of exploited vulnerabilities.

7 Conclusion

Recent module-isolation security architectures provide strong security guarantees of sensitive data stored in small pieces of applications. In practice, however, it is hard to isolate such security-sensitive parts, as most code in an application is sensitive up to some level. As a result, modules of such platforms may need to provide unsafe interfaces. We presented Salus, a new security architecture that can not only provide strong isolation guarantees of sensitive data, but its mutual authentication support also reduces the impact of insecure interfaces. By placing likely attack vectors and targets into different compartments, multiple compartments need to be attacked successfully before an attack target is reached.

Acknowledgement. This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT). Pieter Agten holds a PhD fellowship of the Research Foundation - Flanders (FWO).

References

1. One, A.: Smashing the stack for fun and profit. Phrack Magazine 7(49) (1996)
2. Erlingsson, Ú.: Low-level software security: Attacks and defenses. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2006/2007. LNCS, vol. 4677, pp. 92–134. Springer, Heidelberg (2007)
3. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: EuroSec 2009, pp. 1–8. ACM (2009)
4. Younan, Y., Joosen, W., Piessens, F.: Code injection in c and c++: A survey of vulnerabilities and countermeasures. Technical report, KULeuven (2004)
5. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: Formatguard: automatic protection from printf format string vulnerabilities. In: SSYM 2001, p. 15. USENIX Association, Berkeley (2001)
6. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX 2005, vol. 14, p. 12 (2005)
7. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: EuroSys 2008. ACM (April 2008)
8. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: SP 2010 (May 2010)

9. Azab, A., Ning, P., Zhang, X.: Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: CCS 2011, pp. 375–388. ACM (2011)
10. Strackx, R., Piessens, F.: Fides: Selectively hardening software application components against kernel-level or process-level malware. In: CCS 2012 (October 2012)
11. Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A.V., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F.: Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: Proceedings of the 22nd USENIX Security Symposium (2013)
12. Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: CSF 2012, pp. 171–185. IEEE Computer Society (2012)
13. Singaravelu, L., Pu, C., Härtig, H., Helmuth, C.: Reducing tcb complexity for security-sensitive applications: three case studies. In: EuroSys 2006 (2006)
14. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. ACM SIGOPS 37(5) (2003)
15. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
16. Watson, R.N., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: practical capabilities for unix. In: USENIX Security (2010)
17. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers & Security 11(1), 75–89 (1992)
18. Hoogstraten, H., Prins, R., Niggebrugge, D., Heppener, D., Groenewegen, F., Wettinck, J., Strooy, K., Arends, P., Pols, P., Kouprie, R., Moorrees, S., van Pelt, X., Hu, Y.Z.: Black tulip - report of the investigation into the diginotar certificate authority breach. Technical report, FoxIT (2012)
19. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: Paricheck: an efficient pointer arithmetic checker for c programs. In: ASIACCS 2010, pp. 145–156. ACM, New York (2010)
20. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: USENIX 2009 (2009)
21. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS 2007, pp. 552–561. ACM (2007)
22. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: CCS 2010, pp. 559–572. ACM, New York (2010)
23. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: USENIX 2003 (2003)
24. Jacobs, B., Piessens, F.: The VeriFast program verifier (2008)
25. Saltzer, J., Schroeder, M.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
26. Liedtke, J.: Toward Real Microkernels. Communications of the ACM 39(9) (1996)
27. Yee, B., et al.: Native client: A sandbox for portable, untrusted x86 native code. In: SP 2009, pp. 79–93. IEEE (2009)
28. Sehr, D., et al.: Adapting software fault isolation to contemporary cpu architectures. In: USENIX Security Symposium (2010)
29. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP 1993, pp. 203–216. ACM, New York (1993)
30. Strackx, R., Piessens, F., Preneel, B.: Efficient Isolation of Trusted Subsystems in Embedded Systems. In: Jajodia, S., Zhou, J. (eds.) SecureComm 2010. LNICST, vol. 50, pp. 344–361. Springer, Heidelberg (2010)