Stronger Secrecy for Network-Facing Applications through Privilege Reduction

Petr Marchenko

A dissertation submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at

University College London.

September 2013

I, Petr Marchenko, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Despite significant effort in improving software quality, vulnerabilities and bugs persist in applications. Attackers remotely exploit vulnerabilities in network-facing applications and then disclose and corrupt users' sensitive information that these applications process.

Reducing privilege of application components helps to limit the harm that an attacker may cause if she exploits an application. Privilege reduction, *i.e.*, the Principle of Least Privilege, is a fundamental technique that allows one to contain possible exploits of error-prone software components: it entails granting a software component the minimal privilege that it needs to operate. Applying this principle ensures that sensitive data is given only to those software components that indeed require processing such data.

This thesis explores how to reduce the privilege of network-facing applications to provide stronger confidentiality and integrity guarantees for sensitive data. First, we look into applying privilege reduction to cryptographic protocol implementations. We address the vital and largely unexamined problem of how to structure implementations of cryptographic protocols to protect sensitive data even in the case when an attacker compromises untrusted components of a protocol implementation. As evidence that the problem is poorly understood, we identified two attacks which succeed in disclosing of sensitive data in two state-of-the-art, exploit-resistant cryptographic protocol implementations: the privilege-separated OpenSSH server and the HiStar/DStar DIFC-based SSL web server. We propose practical, general, system-independent principles for structuring protocol implementations to defend against these two attacks. We apply our principles to protect sensitive data from disclosure in the implementations of both the server and client sides of OpenSSH and of the OpenSSL library.

Next, we explore how to reduce the privilege of language runtimes, *e.g.*, the JavaScript language runtime, so as to minimize the risk of their compromise, and thus of the disclosure and corruption of sensitive information. Modern language runtimes are complex software involving such advanced techniques as just-in-time compilation, native-code support routines, garbage collection, and dynamic runtime optimizations. This complexity makes it hard to guarantee the safety of language runtimes, as evidenced by the frequency of the discovery of vulnerabilities in them. We provide new mechanisms that allow sandboxing language runtimes using Software-based Fault Isolation (SFI). In particular, we enable sandboxing of runtime code modification, which modern language runtimes depend on heavily for achieving high performance. We have applied our sandboxing techniques to the V8 Javascript engine on both the x86-32 and x86-64 architectures, and found that the techniques incur only moderate performance

overhead.

Finally, we apply privilege reduction within the web browser to secure sensitive data within web applications. Web browsers have become an attractive target for attackers because of their widespread use. There are two principal threats to a user's sensitive data in the browser environment: untrusted third-party extensions and untrusted web pages. Extensions execute with elevated privilege which allows them to read content within all web applications. Thus, a malicious extension author may write extension code that reads sensitive page content and sends it to a remote server he controls. Alternatively, a malicious page author may exploit an honest but buggy extension, thus leveraging its elevated privilege to disclose sensitive information from other origins. We propose enforcing privilege reduction policies on extension JavaScript code to protect web applications' sensitive data from malicious extensions and malicious pages. We designed ScriptPolice, a policy system for the Chrome browser's V8 JavaScript language runtime, to enforce flexible security policies on JavaScript execution. We restrict the privileges of a variety of extensions and contain any malicious activity whether introduced by design or injected by a malicious page. The overhead ScriptPolice incurs on extension execution is acceptable: the added page load latency caused by ScriptPolice is so short as to be virtually indistinguishable by users.

Acknowledgments

I would like to express my heartfelt gratitude to all of the people who have helped me, not only throughout my research, but to help me become the person I am today.

I have immensely enjoyed the opportunity to work under the supervision of Brad Karp, whose advice and mentoring have had a tremendous impact, both on my academic and personal lives. At every juncture, he has always helped to steer in the right direction and has advised on how to excel in my research. I am also immeasurably grateful to my second supervisor, Mark Handley, for his significant contribution to my work.

It was my great fortune to work at Google as an intern alongside Brad Chen and Úlfar Erlingsson. The experience I have gained from their expertise on all levels has contributed greatly towards my dissertation. I would like to thank them both for this unforgettable experience.

I would also like to offer special thanks to my colleagues and friends. Andrea Bittau introduced me to Bash, Vim, and Screen. Costin Raiciu, Socrates Varakliotis, Georgios Nikolaidis, Astrit Zhushi, Nikola Gvozdiev and Lynne Salameh all played their part in helping to ensure that my student life was a very enjoyable and ultimately successful one.

There are no words to express the extent of my gratitude to my parents, Andrey and Irina. Although they were thousands of miles away, I have always been blessed with their unconditional love and support throughout the course of my study. Along with everyone I have worked with on this project, they have always believed in me and have been instrumental in my success.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Users do much of their computing today with network-based applications. These applications are provided with access to the network and use it to convey information between server and client parts. Today, various desktop applications are replicated as online services [10, 12, 16, 32, 34, 43], and this trend is likely to continue. The advantage of the web model over the desktop one is that a user installs only one application–the web browser–and can use it to access all desired services. When a developer updates or modifies its web application, all the changes happen on the server side, and users are not required to download and install new binaries. In contrast, a desktop application demands pushing new binaries to the users at every update. The convenience of web applications motivated Google to introduce a new operating system, ChromeOS [115], that focuses entirely on web applications rather than native ones.

Users entrust network-based applications with their personal data, including sensitive information such as bank account balances, email, and pharmacy records. Often, network-based applications fail to provide the confidentiality and integrity guarantees for sensitive information as evidenced by a wide range of successful exploits of vulnerabilities [50, 52, 58, 95, 112, 116, 130]. Recently, Sony Corporation announced that an unauthorized person broke into Sony's PlayStation Network through its online service and stole sensitive information of more than *24.6 million users* [50]. This information includes customers' names, addresses, email addresses, birth dates, login names, hashed passwords, and phone numbers. Moreover, Sony admitted a theft of 10,700 direct debit records which include bank account numbers, customer names, account names, and customer addresses. Three months before the attack, security experts warned that Sony PlayStation Network was running on an outdated version of the Apache web server [87]. Thus, vulnerabilities patched in Apache's latest version still persisted in the old one and are very likely to have contributed to this severe compromise of sensitive information in the PlayStation Network.

Our research targets providing end-to-end confidentiality and integrity guarantees for sensitive data in network-facing applications. We explore how to improve the confidentiality and integrity of sensitive data in three domains of network-facing applications. The *native domain* targets native network- based applications written in "unsafe" programming languages, *e.g.*, C and C++, and suffering from low level vulnerabilities such as buffer overflows [56] and format string vulnerabilities [123]. The low level vulnerabilities relate to lack of bounds checking and checks on user input in these languages and may allow an

attacker to inject arbitrary code by suppling malicious input to an application and hence disclose/corrupt sensitive data from the application. The *language runtime domain* describes browsers' language runtimes that execute client-side web applications. The language runtimes are also implemented in unsafe languages and subject to low level attacks. Their safety depends on large amounts of possibly flawed, native code—implementing extensive libraries, as well as advanced mechanisms like runtime code generation and garbage collection [60, 108]. Thus, a client-side web application supplied from a third-party server over an untrusted network may attempt to exploit vulnerabilities in a language runtime and disclose sensitive information. Finally, the last domain is the *web application domain* which defines client-side web applications written in "safe" languages, *e.g.*, JavaScript, Flash, Silverlight, immune to low level attacks but still permitting code-injection attacks, *e.g.*, cross-site scripting, and sensitive information disclosure.

Instead of designing a narrow prevention mechanism that only targets a single attack vector, *e.g.*, buffer overflow vulnerabilities, we focus on a more general defense agnostic to attack vectors and articulated in the *Principle of Least Privilege* (PoLP) [122]:

> *Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.*

In software, the PoLP dictates that a programmer should partition his code into compartments, each of which executes a portion of the program with minimal privilege necessary to carry out its function. When applied to code, privilege corresponds to access rights for data in a system: to read or write memory, file system, or network, to invoke system calls, *&c*. In the context of exploitable vulnerabilities, the PoLP does not prevent a compromise of a compartment but limits the harm the exploit may cause by restricting the actions that it may take.

We explore adhering as closely as possible to the PoLP and partition network- facing applications into multiple privileged and reduced-privilege compartments. The partitioning is based on the requirement of a compartment to process sensitive data. The *privileged compartments* are those that perform operations on sensitive data and thus are entrusted with access to the sensitive data. *Reduced-privilege* or *unprivileged* compartments are the ones that do not process sensitive data and are restricted from accessing it. Privileged compartments should be small and thus possible to audit in order to eliminate vulnerabilities: in case of their compromise, the sensitive information can be disclosed and corrupted. However, it is acceptable if unprivileged compartments contain vulnerabilities: if an attacker exploits such a compartment, there is no risk of sensitive data disclosure or corruption because the compartment lacks access rights for the sensitive data. When a compartment is compromised, we assume that an

attacker may inject arbitrary code and execute it with the compartment's privilege.

In the native domain, most of today's network-facing applications are monolithic and run in a single compartment (under UNIX or Linux, a process), where all instructions execute with full privilege and may access all data. Thus, an exploit of a vulnerability may result in disclosure of sensitive data, and more generally, grants the full privilege held by the application to any code injected by the attacker.

To mitigate the threat of unauthorized disclosure and corruption of sensitive information in native, monolithic applications, we propose to enforce partitioning and privilege reduction in applications with Wedge—a system for splitting legacy, monolithic applications into fine-grained, reduced-privilege compartments, discussed in Chapter 2. Wedge includes two parts: default-deny OS primitives that form isolated compartments requiring a programmer explicitly state the compartments' privilege, and a runtime analysis tool that assists the programmer in identifying the memory-access privileges required by specific code. Wedge's default-deny isolation primitives better suit the PoLP and force a programmer to make considered decisions when granting privilege to compartments. In Chapter 3, we demonstrate how to apply Wedge for splitting legacy applications into reduced-privilege compartments by producing a fine-grained partitioning of the SSL-enabled Apache web server. The Wedge-partitioned server protects users' sensitive data against disclosure and corruption when an exploitable vulnerability is present in code that processes network input.

Network-based applications attempt to safely handle sensitive data by employing cryptographic protocols such as Transport Layer Security (TLS [74], formerly known as SSL [82]) to preserve the integrity and secrecy of the sensitive data as it traverses a network. While these protocols incorporate strong mechanisms to defend against in-network eavesdropping and modification of data in transit, such protocols function in today's distributed systems only as imperfect human-written applications and protocol implementations. Remotely exploitable vulnerabilities in cryptographic protocol implementations [22,23,31,53] prevent the cryptographic protocols from achieving *end-to-end* integrity and secrecy for sensitive data even though the protocols' abstract designs are correct.

As well as in the case of any other network-facing application, privilege- separating an application employing a cryptographic protocol mitigates the threat of sensitive information corruption and disclosure. However, one must partition the application first, and we found partitioning of cryptographic protocol implementations to be a non-trivial problem. In order to partition a protocol implementation, a programmer must correctly identify and label the protocol's sensitive data (*e.g.,* a session key or a server's private key) so as to restrict unprivileged compartments from accessing this sensitive data. Mislabeling of a protocol's sensitive data allows an attacker to disclose or corrupt the mislabeled data which may lead to disclosure of a user's sensitive information transferred within the protocol, disclosure of authentication credentials, and impersonation of peers. We confirm the severity of the protocol partitioning problem with two classes of novel, powerful attacks on cryptographic protocol implementations within state-of- the-art, reduced-privilege applications: the privilege-separated OpenSSH server [118] and the HiStar-labeled Apache SSL web server [137]. Our *session key disclosure attack* allows disclosure and corruption of a user's sensitive data in individual sessions by compromising her session keys. And the

oracle attack allows an attacker to impersonate a server and compromise confidentiality and integrity of users sensitive data in many sessions. These attacks exploit inadequate partitioning of protocols' implementations and thus succeed prior attempts at adhering to the principle of least privilege. In Chapter 4, we propose protocol-agnostic principles for structuring cryptographic protocol implementations to protect sensitive data against these powerful attacks. We demonstrate that the principles are general by applying them to partition the OpenSSH client and server implementations as well as client and server sides of the OpenSSL library.

In the language runtime domain, language runtimes are compiled into browsers or linked into their address spaces as binary plugins. They usually hold the same privileges as the browsers: may process sensitive information in web pages and web applications, and may access user files through the system call interface. Language runtimes execute untrusted client-side code within web pages and thus are at risk of compromise. In particular, an attacker may supply maliciously constructed client-side code; while a browser executes it, the code exploits a vulnerability in the language runtime and injects malicious code into the runtime. Then the injected code runs with the language runtime's privilege and discloses sensitive information from web pages and the client system. Thus, language runtimes add new potential means of attack, as evidenced by the frequent exploits of web-based languages [41, 67, 68].

In Chapter 5, we show how to reduce the privilege of language runtimes with the Native Client (NaCl) SFI sandbox—a sandbox for untrusted x86 code [125, 132]. Reducing privilege of a language runtime makes it pointless for an attacker to exploit it with malicious client-side code as she gains no privileges on top of the ones already given to the client-side code. We chose the NaCl SFI sandbox as a privilege reduction mechanism for language runtimes over Wedge or any other mechanism because of NaCl's portability and rapidly spreading deployment. It is ported into all popular operating systems: Microsoft Windows, MacOS, Linux, and it supports variety of architectures: x86-32, x86-64, ARM. NaCl is a part of the Google Chrome browser, and available as a plugin for Mozilla Firefox. Unlike Wedge, it is implemented in userspace and thus requires no kernel modules or drivers. Note that while a user-space implementation of Wedge exists [65], it requires root privilege and relies on the Security-Enhanced Linux (SELinux) kernel feature, which is available only on Linux OSes.

The novelty of sandboxing language runtimes lies in enabling support for self- modifying code in NaCl. JIT-enabled language runtimes, *e.g.*, the V8 JavsaScript engine, rely heavily on runtime machine-code modification to achieve high performance. We implement a set of machine code constraints and mechanisms to support self-modifying code in NaCl on commodity hardware, the x86-32 and x86-64 architectures, despite the simultaneous execution of that code on other processors by untrusted user threads. With the NaCl sandbox, we lower privilege of language runtimes to the privilege of code they execute: language runtimes have no access to a browser's memory and system call interface. We evaluate the practicality of our system by sandboxing the V8 JavaScript engine in Chapter 5.

Within the web application domain, we focus on the third-party extensions— privileged JavaScript code installed by users to extend browser functionality —allowed by modern browsers. Unlike code in web applications, extensions hold elevated privilege and may manipulate the content of every page and

access the user's sensitive data stored within the browser, *e.g.,* a user's browsing history and bookmarks. This elevated privilege is necessary to enable extensions to enhance browser functionality; however, it presents potential security threats from malicious extensions and malicious pages. A malicious extension may appeal to a user with useful functionality but secretly disclose sensitive data from the browser and web applications using its elevated privilege. A page developer may design a page that exploits a vulnerable extension and injects malicious code into the extension's execution environment. The browser executes the injected code with the extension's privilege, and thus the code may disclose sensitive data from other web applications, *e.g.*, an online banking web application, as well as the browser's sensitive data. The threat of malicious pages exploiting vulnerable extensions is real: we identified a number of extensions from the Google Chrome store which contain code injection vulnerabilities that we manage to successfully exploit by designing malicious pages.

In Chapter 6, we propose simple extension-agnostic and application-agnostic policies that browsers can apply to thwart the threats of malicious extensions and pages. Our containment policies prevent sensitive information disclosure by malicious extensions and vulnerable extensions that allow code injection attacks by malicious pages. They restrict extensions from releasing pages' sensitive data to unauthorized origins. We present a classification of extensions based on their use of page data and design a tailored containment policy for each extension class to allow correct functioning of a wide range of legacy extensions. Our prevention policy blocks code injection into vulnerable extensions and thus prevents malicious pages from elevating their privilege and disclosing sensitive data from other pages or corrupting extensions' execution. To enforce these policies, we built ScriptPolice—a system for enforcing flexible policies on execution of JavaScript applications and browser extensions, implemented in the Google Chrome browser's V8 just-in- time JavaScript compiler and interpreter.

In this thesis, we make the following contributions:

- Experience applying Wedge to partition native, monolithic, legacy applications and suggesting primitives and tools to ease partitioning with Wedge.
- Fine-grained partitioning with Wedge of the SSL-enabled Apache web server that protects users' sensitive data against attacks on the server and the SSL protocol implementation. The partitioning incurs acceptable performance overhead.
- Two novel classes of attacks on cryptographic protocol implementations, session key disclosure and oracle attacks, that allow disclosure and corruption of users' sensitive data even in state-of-the-art reduced-privilege applications, such as the OpenSSH server and the HiStar-labeled Apache web server.
- Protocol-agnostic principles for partitioning protocol implementations to protect against session key disclosure and oracle attacks.
- Implementation of the OpenSSH server and client restructured in accordance with our partitioning principles that is thus immune to session key disclosure and oracle attacks. Restructured implementation of the OpenSSL library (client and server sides) that provides similar guarantees, and

can act as a drop-in replacement for the stock OpenSSL library, bringing robustness against these attacks to a wide range of SSL-enabled applications.

- A set of locally verifiable machine code constraints and mechanisms that allow safe, SFI-based sandboxing of machine code that requires runtime code modification on commodity hardware, despite the simultaneous execution of that code on other processors by untrusted user threads.

- Implementation of our SFI extension in Native Client for the x86-32 and x86-64 architectures, as well as ports of the V8 JavaScript runtime allowing software to embed languages like JavaScript as untrusted components with modest overhead.

- Novel vulnerabilities in popular extensions that allow disclosure of sensitive information from web pages of any origin by maliciously constructed pages.

- Flexible discretionary access control policies tailored to the canonical behaviors of legacy browser extensions. We characterize three simple canonical behaviors of legacy browser extensions, and a containment policy tailored for each that applies DAC either upon reads of page data or writes to the network, as appropriate to an extension's behavior. These policies prevent leakage of sensitive data while preserving extension functionality.

- Design and implementation of a fast just-in-time-compiled information flow control (IFC) system for JavaScript and an IFC-based scheme for prevention of code-injection attacks. While others have applied IFC to the JavaScript interpreter, interpreted IFC offers an average 37x per-operation execution time overhead *vs.* without IFC, whereas JIT-compiled IFC offers an average 1.25x overhead.

- Dynamic taint specialization, an optimization that leverages JIT compilation to omit label propagation instructions (and their overhead) from code that does not process tainted data, dynamically recompiling to include such instructions only when code encounters tainted data at run time. This IFC design is the first to do full taint tracking with execution overhead *only in proportion to the volume of tainted data processed.*

- ScriptPolice, an efficient and effective browser policy system. Implemented for the Google Chrome browser's V8 JavaScript engine, ScriptPolice incorporates all the above techniques. It allows a wide range of legacy extensions to function correctly, and prevents sensitive data exfiltration and script injection, all while typically increasing page load times by 5% or less.

Chapter 2

Privilege Separation Mechanisms

In this chapter, we discuss existing isolation mechanisms that state-of-the-art privilege-separated applications apply to enforce the PoLP. We refer to these mechanisms and applications when presenting work on Wedge in Chapter 3 and on partitioning cryptographic protocol implementations in Chapter 4.

In monolithic applications, all code may accesses all data, and an exploit may result in immediate disclosure and corruption of sensitive data. The defense mechanisms discussed in this chapter are applied for privilege separation and mitigate the threat of sensitive data disclosure and corruption.

The general approach for privilege separation requires splitting an application into several compartments with different privilege levels. *Privilege* refers to the ability to read and write sensitive data in memory and files, and to execute system calls that may provide access to sensitive data. Then, a defense mechanism enforces the compartments' privilege restrictions. Unprivileged compartments are denied access to sensitive data, and may execute possibly vulnerable code without a risk of sensitive data disclosure if the code gets compromised. For example, code that parses data is very hard to analyze for security vulnerabilities [47–49, 51]; however, the parsing code does not normally need access to sensitive data to perform its function. When an attacker compromises a compartment, we assume the attacker may inject arbitrary code and execute it with the compartment's privilege. Privileged compartments perform operations on sensitive data; thus, they are privileged to access the data. If an attacker compromises a privileged compartment, she may disclose the sensitive data. However, applications require only a limited set of operations on sensitive data, and thus their privileged code base is normally small. The small privileged code base makes it possible for security experts to audit privileged code for vulnerabilities–that is, to inspect such code manually and attempt to validate that it does not contain vulnerabilities.

2.1 Privilege Separation with Processes

Early privilege-separated applications use UNIX processes to enforce isolation and privilege reduction of compartments [42, 64, 93, 118]. Operating systems isolate the virtual address spaces of processes from one another to prevent one process from tampering with the execution of another process. However, data isolation by itself is not sufficient for containment of a compromised compartment running an attacker's malicious code. The attacker-controlled process may issue system calls and use inter-process

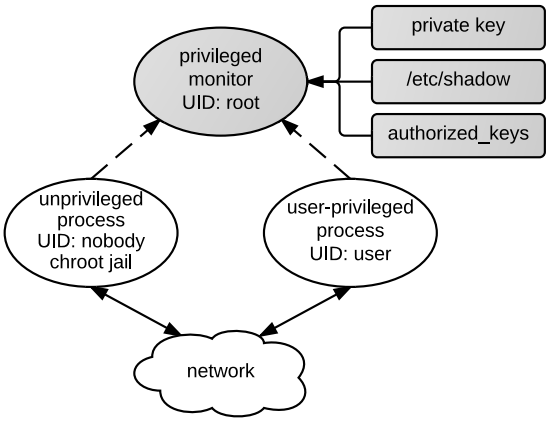


Figure 2.1: The Privilege-Separated OpenSSH Server.

communication mechanisms, *e.g.*, shared memory, signals, locks, *&c.*, to tamper with other processes running under the same user ID (UID). The most notorious system call supported by UNIX-like OSes is `ptrace` which allows debugging a process with the same UID. A process may attach to another process with the `ptrace` system call and control its execution by reading and writing memory as well as by intercepting system calls and signals. To restrict possible interference via system calls, privilege-separated applications apply an isolation mechanism known as a *setuid jail* through which an application can change the user and group IDs of unprivileged compartments to user and group IDs of an unused user, *e.g.*, *nobody*, with the `setuid` and `setgid` system calls. Because, privileged and unprivileged compartments run in processes with UIDs, they cannot tamper with one another.

Privilege-separated systems must also restrict unprivileged processes from accessing the file system to prevent tampering with other processes via shared and temporary files, as well as to prevent compromised compartments from leaking information about the system (*e.g.*, available programs, content of configuration files and other files readable by the unprivileged compartments). A *chroot jail* restricts access to the filesystem in accordance with the needs of privilege-separated systems. An unprivileged process must issue a `chroot` system call, which changes the root of the filesystem as seen by the process, and switch its root to an empty directory. Afterwards, the unprivileged process is not able to access any files.

The `setuid` and `chroot` system calls are only available to processes with root privilege. Thus, an unprivileged process must start as `root` and then reduce its privilege by enforcing chroot and setuid jails. Dropping root privilege is essential as an application may escape its chroot jail if it keeps the root privilege [127]. The process may perform its privilege reduction after initialization but strictly before processing any input that may carry an attacker's exploit. Note that unprivileged processes may tamper with each other using system calls if they run under the same UID.

Privilege-Separated OpenSSH Server

The OpenSSH server [118] is a state-of-the-art privilege-separated application; it employs chroot and setuid jails to contain its unprivileged compartments. Figure 2.1 shows the architecture of the OpenSSH

server, where ovals represent code: shaded ovals are privileged compartments, while white ovals are unprivileged compartments. A dashed arrow between compartments A and B indicates that A may invoke an operation in B with arguments and retrieve the result. Boxes represent sensitive data. A solid arrow from data to a compartment denotes that the compartment may read that data; an arrow in the reverse direction denotes write access.

The OpenSSH server establishes a user connection within an unprivileged process which is chroot'ed to an empty directory to restrict its access to files and has a user ID of nobody to prevent it from tampering with users' processes. The unprivileged compartment performs a key exchange protocol resulting in a shared symmetric key which is used in an SSH session between client and server to protect confidentiality and integrity of data in transit. The unprivileged compartment also authenticates the server to the client by signing a session ID with the server's private key. The server's private key is highly sensitive information as it may be used to impersonate the server and compromise sensitive data in multiple sessions of different users. Therefore, the unprivileged process is restricted from accessing the server's private key. Instead, it invokes a signing operation within a *privileged monitor* which is a separate process running as root and thus provided with access to the sensitive information (here, the private key). The monitor isolates sensitive data and exports privileged operations on this data to unprivileged compartments. Finally, the unprivileged process performs user authentication which may involve password authentication, public key authentication, and other authentication methods. Again, authentication requires access to sensitive data such as files containing passwords and authorized users' keys; the monitor performs these operations on behalf of the unprivileged process. As soon as the user authenticates to the server, the monitor kills the unprivileged process and creates a new one with the user ID of the authenticated user. The user-privileged process runs a shell for the remote user and is allowed to access the user's local files.

The partitioning scheme in Figure 2.1 is designed to deny an attacker execution of code with root privilege on the server and to restrict access to the server's private key. Provos *et al.* state that "programming errors occurring in the unprivileged parts can no longer be abused to gain unauthorized privileges" [118]. This claim holds because an attacker only interacts directly with the unprivileged process, and the unprivileged process executes with restricted file system access (enforced with a chroot jail) and with user and group IDs of nobody. The chroot and setuid jails prevent the process from tampering with other processes: the privileged monitor and user-privileged processes.

2.2 Security-Enhanced Linux

The chroot and setuid jails are coarse-grained isolation mechanisms and do not suffice to deny certain behavior. For example, in the OpenSSH server, a compromised unprivileged process serving an attacker's connection may ptrace and disclose sensitive information, *e.g.*, a user password, of another unprivileged process that handles an SSH connection of a different user. This is possible because both unprivileged processes run under the same user ID. In Chapter 4, we show an attack that employs a `ptrace` system call to disclose sensitive data of a user from the OpenSSH server. The attack succeeds because the setuid jail provides insufficient isolation for the unprivileged processes. Thus, privilege-separated systems

require tighter isolation for unprivileged compartments.

Security-Enhanced Linux (SELinux) [98] is a security extension to Linux that post-dates Provos *et al.*'s work; it allows enforcement of flexible mandatory access control policies specified by a system administrator. These policies support more fine-grained restriction of a process's privilege than under stock Linux, primarily by checking system call invocations in the kernel against a per-process access control list.

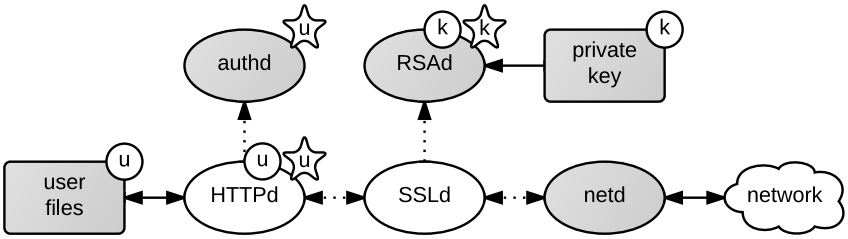
SELinux supports three types of policies: type enforcement, role-based access control, and multi-level security. Under the type enforcement mechanism, an application executes under a policy which assigns types to system resources such as files, sockets, *&c.*, and domains to processes. A policy may specify rules that allow a process running in a certain domain to access resources tagged with certain types at a system call granularity, *e.g.*, "this domain may invoke the `read` system call on files of this type only". The policy is pre-loaded into the kernel at application installation, and the kernel enforces it at application runtime. At every system call invocation, the kernel checks whether there is a policy rule associated with the process's domain that allows the requested system call on the requested SELinux type. We do not use role-based access control and multi-level security in our work, and therefore we omit the details of these mechanisms for brevity.

2.3 Decentralized Information Flow Control

Decentralized Information Flow Control (DIFC), as implemented in the research prototype operating systems Asbestos [77] and HiStar [136], and retrofitted to Linux in Flume [94], offers a different approach to limiting privilege within applications. Instead of restricting code from accessing sensitive data, the system tracks data flow within an application and makes sure that the sensitive data are not disclosed without authorization. This requires a programmer to specify an information flow policy by labeling sensitive data with sensitivity levels. When a compartment accesses labeled data, it gets tainted with the data's sensitivity level, and the system prevents the tainted compartment from communicating with compartments with less-sensitive levels at risk of disclosing the information to untainted compartments. The DIFC system controls all applications' outputs, such as the console, files, network, *&c.*, which makes it possible to guarantee that none of the sensitive information is leaked out of the system by restricting tainted compartments from communicating with such destinations for output. To allow developer-controlled output of sensitive information, DIFC introduces privileged compartments that possess an ownership over some sensitivity levels that allows to declassify sensitive data (to strip the data's sensitivity labels) and release it to less tainted compartments or out of the system. This mechanism is used to perform privileged operations over sensitive data and share the results of the operations with unprivileged compartments.

HiStar-Labeled SSL Web Server

Zeldovich *et al.* present a state-of-the-art SSL web server privilege-separated with DIFC [137], shown in slightly simplified form in Figure 2.2. Circles annotating data items and code compartments indicate labels; in the latter case, a compartment is tainted with the label in question. A label within a star denotes



In the interest of brevity, we present only data confidentiality labels and omit data integrity ones. We also omit SSLd and netd's labels.

Figure 2.2: The HiStar-Labeled SSL Web Server.

that a compartment owns that label (and may declassify data labeled with it).

The HiStar-labeled SSL web server is partitioned into several unprivileged compartments to limit the effect of a compromise of any single one. The major compartments are per-connection *SSLd*, per-connection *HTTPd*, and shared *RSAd* daemons. SSLd handles a client's SSL connection and performs key exchange, server authentication, encryption and decryption. HTTPd processes cleartext HTTP requests; it uses SSLd to decrypt requests and encrypt replies. HTTPd can obtain ownership of a user's label by authenticating with the privileged *authd* daemon. Label ownership allows HTTPd to read the user's data and declassify it for transferring over the network. The privileged *netd* compartment serves as a barrier between the application and the network. It passes only declassified data (with no label) to the network. The goal of the server is to provide data isolation among users which is achieved with per-connection compartments; even if they are exploited, they cannot disclose one user's data to another, as each is tainted with a particular user's label.

The privileged RSAd compartment isolates the server's RSA private key, which is essential for server authentication; the server signs an ephemeral RSA public key or a public Diffie-Helman component with the private key (depending on the negotiated ciphers) to prove its identity. SSLd and HTTPd are unprivileged; while they may read the private key, neither owns its label. Thus, if they do read the private key, they will become tainted with the key's highly sensitive label and refused communication with the network by netd, which is tainted with a less sensitive label than that of the RSA key.

2.4 Wedge

Wedge is a system for splitting applications into reduced-privilege compartments. Wedge suggests new OS primitives built in the spirit of the PoLP and tools that help to partition legacy applications. My colleague Andrea Bittau designed and implemented Wedge, and this work is the core of his PhD thesis [65]. We contributed to the work by applying Wedge to legacy applications and suggesting tools and techniques that ease the applications' partitioning. Our work on Wedge described in Chapter 3 identified the challenges associated with partitioning cryptographic protocol implementations and initiated the work on protocol partitioning rules discussed in Chapter 4.

Applying privilege reduction to applications significantly mitigates the threats of sensitive information disclosure and corruption by attackers. The applications are structured in such a way that a compromise of unprivileged compartments does not allow an attacker to corrupt and disclose sensitive

information.

It appears that today's primitives used for splitting applications into reduced-privilege compartments are not designed for this purpose. In particular, processes, which privilege-separate the OpenSSH server, as discussed in Section 2.1, and partition the HiStar- labeled SSL web server, as described in Section 2.3, grant privilege by default, contradicting the PoLP. When a process forks a child, the child inherits the parent's data and opened file descriptors which may contain sensitive information. To prevent default-sharing of data, a child must explicitly scrub sensitive data and close file descriptors that the parent would like to keep private immediately after it is forked. However, scrubbing is an error-prone process as a programmer may simply forget to scrub some data, or she may not even be aware of the existence of sensitive data allocated and/or copied by a third-party library.

2.4.1 Design

Wedge is a system that allows splitting complex, legacy applications into reduced-privilege compartments under a *default-deny* model. Wedge consists of partitioning primitives and a pair of tools to help programmers to partition their legacy, monolithic applications. The partitioning primitives include *sthreads*—a primitive that defines a default-deny compartment with a security policy attached at its creation, *tagged memory* for expressing memory privilege, and *callgates*—a mechanism that defines an interface to privileged code and sensitive data for unprivileged compartments.

Sthreads

An sthread is a combination of a thread of control and an associated security policy. The security policy specifies:

- The memory that the sthread may access expressed in memory tags, and the access permissions for each tag (read, read-write, copy-on-write).
- The file descriptors that the sthread may access, and the permissions for each file descriptor (read, write, read-write).
- The callgates that the sthread may invoke.
- The SELinux policy [98] which defines the system calls that the sthread is allowed to use.

Unlike forked processes, an sthread has no access to a parent's resources by default, nether memory nor file descriptors. In order to share a file descriptor or a memory region, a programmer must explicitly state this in a security policy that is attached to the sthread at its creation time. This memory sharing is facilitated with memory tags, described next, which require a programmer explicitly tag data shared with the sthread and thus reason about whether it is safe to share a particular data item.

An sthread can only create a child with the same level of privilege, stated in the sthread's policy, or reduce the child's privilege by attaching a more restrictive policy. Note that changes in SELinux policy must be explicitly allowed as domain transitions, subject to the application's SELinux policy.

Tagged Memory

A programmer specifies memory permissions in terms of memory tags. A memory tag is a label that the programmer assigns to an empty memory region; we call such assignment tagging a memory region. Later in the program, a data buffer can be allocated within the tag (or more precisely, in the memory region associated with this tag) by using our allocator *smalloc*. smalloc is a library call that is similar to malloc, but it takes two parameters, the tag where a memory buffer needs to be allocated and the size of the buffer. The programmer may allow access to a memory tag and all data allocated with it to another sthread by specifying the tag and its access permission in an sthread's policy. Non-smalloc'ed data, such as malloc-allocated data and stack-allocated data, cannot be referenced by any tag which makes their sharing impossible and enforces the default-deny model. Wedge implements tagged memory and its sharing with the standard hardware page protection mechanism [89].

Privilege-separating a legacy application with sthreads may require to tag dynamically allocated objects within third-party libraries. If a library is closed source and distributed as a binary, the smalloc allocator cannot be applied. For this specific case, we provide two library calls *smalloc_on* and *smalloc_off*. Every malloc call between smalloc_on and smalloc_off is converted to an smalloc call with the tag specified at the initial smalloc_on call. This mechanism significantly simplifies tagging malloced data within an application; however, a programmer must avoid it if possible or use it with caution as there is a risk to unintentionally tag and share sensitive information that is manipulated by the closed-source library.

For tagging statically allocated global data, one may convert it to heap-allocated data and apply the smalloc mechanism. However, this technique requires significant porting effort when a legacy application makes heavy use of static globals. Wedge provides a more convenient mechanism for tagging the static globals by adding annotations in source code against variables to be tagged. All annotated variables are placed in a memory region referenced by a tag which a programmer can use in an sthread's policy for sharing these globals with the sthread.

The details on implementation of the smallic_on and smalloc_off mechanism as well as on tagging statically allocated data can be found in our paper [66].

Callgates

A callgate is a mechanism that allows an sthread to perform an operation that requires elevating privilege, *e.g.*, to compute on sensitive data. A callgate can be created by an sthread which must specify an execution entry point and a set of permissions equal to or less than its current privilege level. Then the sthread may create an unprivileged child sthread with a more restrictive policy than the callgate's one, but allow the child sthread to invoke the callgate with the child-specified parameters when a controlled privilege escalation is required. Thus, the child sthread may call into the callgate to perform privileged operations; the callgate invocation results in creation of a new sthread with the callgate's permissions that starts execution at the entry point specified by the callgate creator. The child blocks until the callgate finishes its execution and returns the result back to the caller. Then the callgate's sthread is terminated.

Creating and destroying an sthread for a short-lived callgate introduces significant performance

overhead especially in the case of server applications such as the Apache web server [66]. In order to mitigate this performance overhead, we provide long-lived callgates which we name *recycled callgates*. A recycled callgate amortizes the cost of sthread creation over multiple invocations as we do not destroy it after each invocation but block until the next call [66].

Crowbar: Partitioning Tools

Privilege separation of legacy applications with sthreads may require substitution of processes by sthreads. Within the default-deny model, a programmer would need to identify all memory objects that an sthread (formerly a process) accesses, explicitly tag them, and share with the sthread. Identifying these memory objects and their tagging are non-trivial problems. For example, the user session-handling code in the Apache web server makes use of over 600 distinct memory objects, scattered throughout the heap and globals [66]. Just identifying these is a burden, and tagging them requires non-trivial effort since a programmer must find the objects' allocation places in the application's source code. To ease the identifying and tagging of an sthread's memory dependencies, we provide *Crowbar*, a pair of Linux tools that assist a programmer in applying the default-deny primitives in applications.

A programmer uses *cb-log*, the first Crowbar tool, to log memory accesses and allocations made by an application at runtime. For each memory read and write, cb-log stores a complete backtrace of the program execution where this memory access happened. The backtrace is augmented with function names, file names, and line numbers, and it provides the execution context of each access. cb-log can identify the following memory objects. For a global object, it provides the object's name using the application's debugging information. For each malloc use, cb-log keeps a corresponding backtrace which is used to identify a heap object when referenced. For stack references, cb-log provides the backtrace of the current execution stack frame and the reference offset from the frame base. This information helps a programmer to identify which local variable on the stack is accessed.

The second Crowbar tool, *cb-analyze*, is used to query a log file produced by cb-log. The programmer can issue the following queries that help in tagging memory objects.

- Given a procedure, what memory items and with what permissions do it and all its descendants in the execution call graph access? This query is crucial for identifying which memory objects must be tagged and sheared for a procedure to run in a reduced-privilege sthread.

- Given a list of data items, which procedures use any of them? If a programmer wishes to provide a callgate for operations on some data objects, this query helps to identify which operations must be implemented by the callgate.

- Given a procedure known to generate sensitive data, where do it and its descendants in the execution graph write data? A programmer can use this query to identify data that may contain some sensitive information produced as a side-effect of an operation on known sensitive data.

Crowbar is a runtime analysis tool implemented in Pin [101], a dynamic binary instrumentation tool; therefore, it is trace driven and analyzes a single execution path at a time. A programmer may need

to do multiple runs of an application with diverse inputs to generate a trace that ensures coverage of a broad portion of the application.

Chapter 3

Contribution to Wedge

Our colleague Andrea Bittau implemented Wedge, which constitutes his PhD thesis [65]. We contributed to Wedge by investigating how difficult it is to apply default-deny primitives, sthreads, when privilege-separating legacy, monolithic applications.

We partitioned the SSL-enabled Apache web server 1.3.19 [4] and the OpenSSL 0.9.6 library [30] to protect users' sensitive information transferred within SSL sessions and processed by the server from unauthorized disclosure and corruption by a remote attacker. Our privilege-separated Apache web server is designed to withstand the attacks described in Section **??**. To justify the need for our partitioning, we show how the baseline Apache web server fails to defend against these attacks and thus may allow disclosure and corruption of user sensitive data, as described in Section 3.3.

The work on partitioning of the Apache web server identified the need for the Crowbar tools and motivated their design as well as the design of additional primitives: the smalloc_on/off and tagging globals, Section 2.4. The performance evaluation of the privilege-separated Apache web server revealed the significant overhead imposed by callgates and motivated the design of recycled callgates, Section 2.4.1. Below, we summarize our contributions to Wedge.

- Experience of applying Wedge to partition native, monolithic, legacy applications and suggesting primitives and tools to ease the Wedge-partitioning.
- Fine-grain Wedge-partitioning of the SSL-enabled Apache web server that protects users' sensitive data against attacks on the server and the SSL protocol implementation. The partitioning incurs acceptable performance overhead.

3.1 Background: SSL Protocol

In order to understand some of our attacks described in the next section, it is important to know how the SSL protocol operates. This section presents an overview of the SSL protocol [74, 82] and the RSA key exchange protocol (used within SSL) and highlights their key features.

The Apache web server employs the SSL protocol to securely transfer data between the server and a client. Confidentiality and integrity of data transferred with an SSL-enabled channel rely on the protocol's sensitive data, *e.g.*, symmetric keys, a pre-master secret, a server private key. Two *symmetric keys* provide confidentiality of the data in transit: one is used for decrypting inbound traffic and the other

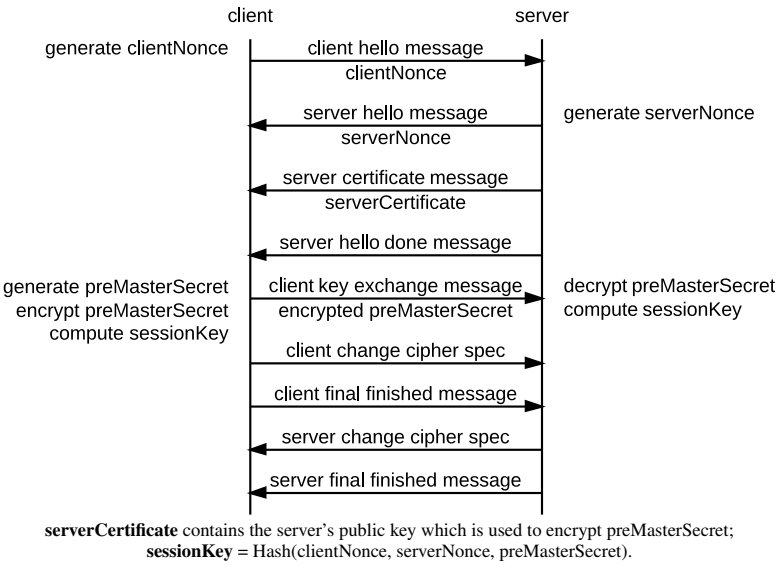


Figure 3.1: The RSA Key Exchange Protocol within the SSL Protocol.

one for encrypting outbound traffic. The SSL protocol enables data integrity guarantees with Message Authentication Code (MAC) [74, 82] which requires another two symmetric keys, similarly, for inbound and outbound traffic. Disclosure of these keys allows a man-in-the-middle attacker to eavesdrop cleartext messages and forge validly-encrypted messages within a single SSL-protected communication channel.

The SSL protocol derives all four symmetric keys from a single secret *session key*. A client and a server perform a key exchange protocol and negotiate the shared session key. SSL supports the RSA key exchange protocol, the Diffie-Hellman key exchange protocol, and some variations of these two protocols [74, 82]. In this section, we discuss the RSA key exchange protocol since it is the only key exchange protocol supported by our Wedge-partitioned Apache web server.

Figure 3.1 shows SSL messages when the RSA key exchange protocol is in use. A client starts by sending its random nonce within a client hello message. A server replies with a random nonce and a certificate signed by a trusted authority (server hello and server certificate messages). The client verifies the certificate's signature and ensures that the certificate was not modified in transit. Then, it obtains a server public key included in the certificate. The client generates a random pre-master secret, encrypts it with the server's public key, and sends the encrypted pre-master secret to the server in client key exchange message. Only the true server possessing the corresponding private key may decrypt the pre-master secret. Thus, public key cryptography allows a client and server confidentially share a pre-master secret and compute a shared session key by hashing three components: a client nonce, server nonce, and pre-master secret. Then the parties derive symmetric keys from the computed session key.

The random nonces are not sensitive and transferred in cleartext. They are included in a session key to ensure that the generated key is different each session and that a man-in-the-middle attacker cannot force the key to match any previously used one by modifying nonces. If the attacker modifies one of the nonces, the server's session key and the client's one will not match and the key exchange fails.

The secrecy of a session key[1] depends on the secrecy of a pre-master secret and the server's private key. If an attacker learns the pre-master secret or the session key, she may compute corresponding symmetric keys and disclose/corrupt sensitive data within a single SSL session.

The server's private key ensures secrecy of pre-master secrets, and thus it protects confidentiality and integrity of multiple SSL sessions of different users. A man-in-the-middle attacker leaking a server private key may disclose sensitive data of eavesdropped SSL sessions by decrypting pre-master secrets and computing session keys. In addition, the server authenticates to a client by proving possession of the private key. Within the RSA key exchange, the server proves that it has the private key by successfully decrypting pre-master secrets presented by clients. Thus, if an attacker obtains the server's private key, she will be able to decrypt the pre-master secretes and impersonate the server.

A *change cipher spec message* signals that a client or server enables MAC and encryption on the communication channel and therefore all the following messages will be MACed and encrypted. *Final finished messages* are there to test if the server's computed symmetric keys match the client's ones, and that the SSL handshake messages were not tampered with. The final finished messages are MACed and encrypted and contain hash of all previously exchanged SSL handshake messages and a session key. SSL/TLS uses the SHA [45] and MD5 [119] algorithms to produce the hash. There are two final finished messages in order to check inbound and outbound symmetric keys of both parties.

3.2 Threat Model

Our goal is privilege separation of the Apache web server with Wedge to protect it from the three attack models described below. We consider a powerful attacker originating from the network who may compromise multiple unprivileged compartments, shown as white ovals in our diagrams, within the privilege-separated Apache web server. The attacker may inject arbitrary code and execute it with the compartments' privilege. Code in privileged compartments, represented as shaded ovals in our diagrams, must be audited against vulnerabilities by security experts so to prevent their compromise. Within our threat model, an attacker cannot compromise privileged compartments as well as escape privilege restrictions imposed on unprivileged compartments. Thus, the attacker cannot exploit the system enforcing compartment privileges, the operating system, and any other software running on the same machine as the Apache server.

Our Apache partitioning is designed to thwart three attacks: *user data disclosure*, *private key disclosure*, and *session key disclosure*. All of them aim at disclosure and corruption of users' sensitive data processed by the server but use different means to achieve this. The first one denotes an attacker that establishes a connection to the server, compromises the server, and discloses *other* users' sensitive information if such information is within the compromised compartment's reach. The private key disclosure attack involves an attacker compromising the server and disclosing the server's private key. Then, the attacker uses the key to impersonate the server and obtain sensitive information of users that connect to a forged server instead of the real one.

The session key disclosure attack is shown in Figure 3.2 where an attacker tricks a client to connect

[1] Throughout this document, the session key term refers to a session key and corresponding symmetric keys.

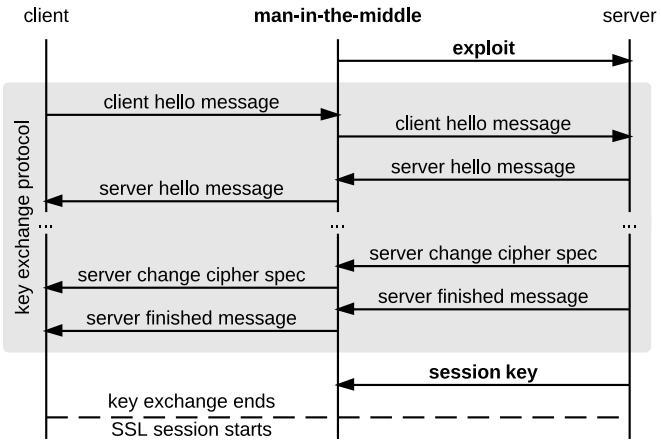


Figure 3.2: The Session Key Disclosure Attack within the SSL Protocol.

to an attacker-controlled server impersonating the real server, or she places herself on a communication path between the client and the legitimate server. When the client connects to the attacker's server, the attacker initiates her own connection with the real server and compromises it. Thus, the attacker keeps two active connections: client-attacker and attacker-server. Note, all above happen before key exchange protocols are initiated in both connections. The client starts a key exchange protocol with the attacker's server, the attacker relays all client messages to the real server and all server replies to the client. Thus, the client and server perform the key exchange protocol with messages *unmodified* by the attacker. In the end of the protocol, both parties, the client and server, compute a shared session key which the attacker discloses from the compromised server. The client and server are unaware that the attacker has obtained the session key and continue exchanging sensitive information within the compromised SSL session. The attacker discloses the sensitive information using the leaked session key.

Our threat model defines only the attacks presented above and does not address other attacks such as denial-of-service attacks where an attacker exhausts a system's resources to disrupt the operation of the web server.

3.3 Baseline SSL-Enabled Apache

We discuss the architecture of the baseline Apache web server and show that it fails to protect user sensitive data from the attacks defined above.

In the baseline Apache web server, a master process performs the server's initialization and loads a server private key into its memory. Then it forks a pool of worker processes to serve remote connections. Each worker process obtains a copy of the private-key-initialized memory snapshot from the master. When a client connects to the server, one of the worker processes is scheduled to handle the client's connection. The process enables the SSL protocol, authenticates the server to the client, negotiates a session key, and sets symmetric keys for the SSL connection. Then the worker process serves the client's HTTP requests. When the client terminates the connection, the worker process joins a queue of available workers and waits to be scheduled to serve a next connection.

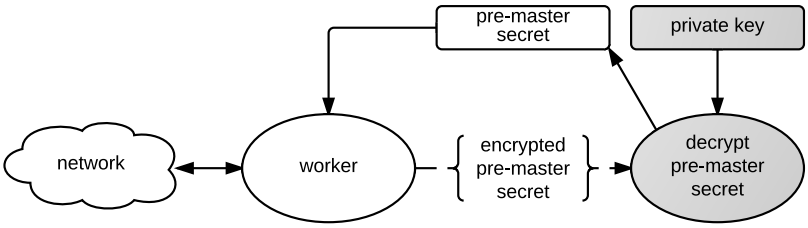


Figure 3.3: SSL-Enabled Apache: Oracle-Exposed Partitioning.

The baseline Apache web server is vulnerable to the user data disclosure attack because it reuses worker processes across multiple sessions of different users. An attacker may compromise a worker process and wait while it is scheduled to serve a connection of a legitimate client. Then, the compromised worker process may disclose sensitive data that the user sends to the server. Alternatively, the compromised worker process may disclose the server private key which renders the server vulnerable to the private key disclosure attack. Finally, the server's architecture does not attempt to safeguard a session key of a legitimate client. Thus, the session key disclosure attack may succeed in the baseline Apache web server.

The reason the baseline Apache web server fails to defend against our attacks is because its workers have a monolithic design, where all code has access to all data, and the workers are reused to serve different sessions.

3.4 Wedge-Partitioned SSL-Enabled Apache

We designed a partitioning scheme that thwarts the user data disclosure, private key disclosure, and session key disclosure attacks. The Crowbar tools helped us to compartmentalize Apache's worker processes, and Wedge's isolation primitives define compartments and restrict their privileges according to the PoLP. Next, we discuss how the Wedge-partitioned Apache web server addresses each attack.

3.4.1 User Data Disclosure Threat

In order to prevent the user data disclosure attack, we modified the architecture of the baseline Apache web server. We substituted worker processes with worker sthreads and disabled reusing the worker sthreads across multiple connections. Thus, the master process creates a distinct sthread to serve a user connection, and when the connection terminates, the sthread is destroyed. The per-connection sthread architecture prevents an attacker compromising a worker sthread from disclosing user sensitive information as the compromised sthread obtains data supplied only from attacker's connection. Note that the policy attached to the sthread restricts it from tampering with other sthreads and accessing unauthorized files.

3.4.2 Private Key Disclosure Threat

In the baseline Apache web server, a compromise of a worker process may result in disclosure of the server private key which resides in the worker's address space. To deny the disclosure of the private key in the privilege-separated web server, our master process allocates the key in a memory region labeled with the "private key" tag. The security policy attached to worker sthreads lacks the permission

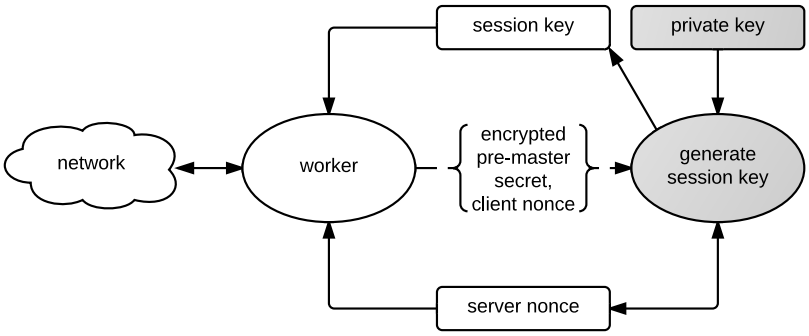


Figure 3.4: SSL-Enabled Apache: Oracle-Resistant Partitioning.

to access private-key-tagged memory and thus denies the sthreads access to the private key. However, worker sthreads must perform some operations with the private key, *e.g.*, decryption of pre-master secrets to perform the RSA key exchange protocol. To allow these privileged operations, the master process allows the worker sthreads to invoke a privileged callgate which is granted access to the private key to perform the required operations on behalf of a worker sthread.

Figure 3.3 shows the described partitioning scheme that thwarts private key disclosure. However, does the scheme prevent a compromised sthread from using the private key to impersonate the server? The *decrypt pre-master secret* callgate acts as a decryption *oracle* for the private key: a compromised worker sthread may supply any encrypted data and retrieve a corresponding cleartext message. Recall, in the RSA key exchange protocol, a server authenticates to a client by decrypting a pre-master secret presented by the client. Thus, in order for an attacker to impersonate the server, she must decrypt a user's pre-master secret. To decrypt the pre-master secret, the attacker may exploit a worker sthread of the server and employ the privileged callgate as a decryption oracle.

We designed an oracle-resistant partitioning scheme for the Wedge-partitioned Apache web server that isolates the server's private key from worker sthreads, allows them to perform the RSA key exchange protocol, and restricts the private key decryption oracle. Figure 3.4 shows this partitioning scheme. The key difference between the previous scheme and the current one is that the privileged callgate does not release a decrypted pre-master secret to a worker sthread; it releases a computed session key instead. When an unprivileged worker sthread calls into the *generate session key* callgate, it supplies an encrypted pre-master secret and a client's random nonce. The callgate decrypts the pre-master secret and generates the session key using the supplied client nonce and a server nonce generated earlier. The callgate generates the random server nonce on the worker's request; the nonce is shared with the worker so that the worker can pass it on to the client, but the callgate keeps its private copy of the nonce for generating the session key. Thus, the callgate shares the server nonce read-only.

The current partitioning structure restricts an attacker from decrypting arbitrary pre-master secrets. However, the attacker may attempt to supply a client nonce and encrypted pre-master secret from an eavesdropped user session to the callgate, and produce the session key that matches the one in the eavesdropped session. This will not work because the callgate generates a different server nonce each time

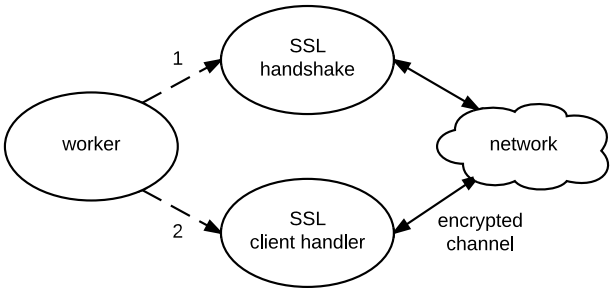


Figure 3.5: SSL-Enabled Apache: Top-Level Partitioning.

it produces a session key, and the nonce makes the session key unique to the current session. Recall, a session key is a cryptographic hash of three inputs, Section 3.1, and if the attacker does not control one of them, she cannot influence the resulting session key.

3.4.3 Session Key Disclosure Threat

There are two phases to an SSL session. The first phase is an *SSL handshake* which performs a key exchange protocol, shares a session key, and authenticates a server. The shared session key is used to enable encryption and MAC on the SSL channel to provide confidentiality and integrity for user data transferred in the second phase, an *SSL client handler* phase. Note that in the first phase, the server and a client exchange cleartext messages, and in the second phase, they process *only* encrypted and MACed messages.

A session key disclosure attacker may compromise the server only during the SSL handshake phase but not in the SSL client handler phase. The latter phase requires the attacker to validly encrypt and MAC (the latter is more important) an exploit which is not possible without a knowledge of the session key. The server rejects any SSL messages that are not properly MACed with the shared session key at decryption and MAC verification stage. The only way the attacker can exploit the SSL client handler phase is by compromising the decryption and message authentication code. However, we rely on this code to be audited and vulnerability free. The SSL client handler phase is not exposed to the session key disclosure attacker as soon as the SSL handshake phase preserves confidentiality of the session key.

Because the two SSL phases have different threat models, we treat them separately and dedicate an sthread to each phase. Figure 3.5 shows the high-level partitioning needed to implement the two-phase approach. The *worker* sthread does no work, but it is there to invoke *SSL handshake* and *SSL client handler* sthreads and to make sure that they only execute sequentially. The worker sthread invokes the SSL handshake sthread first, and waits while it terminates. After SSL handshake terminates, worker invokes the SSL client handler sthread. The two-sthread model guarantees that even if the session key disclosure attacker compromises the SSL handshake sthread, a new attacker-free sthread starts afresh for the SSL client handler phase. Thus, we make sure that the attacker does not propagate to the SSL client handler phase.

In the SSL handshake phase, a session key and data that may allow deriving the key, *e.g.*, a private key and pre-master secret, are sensitive and must not be shared with compartments that are at risk of

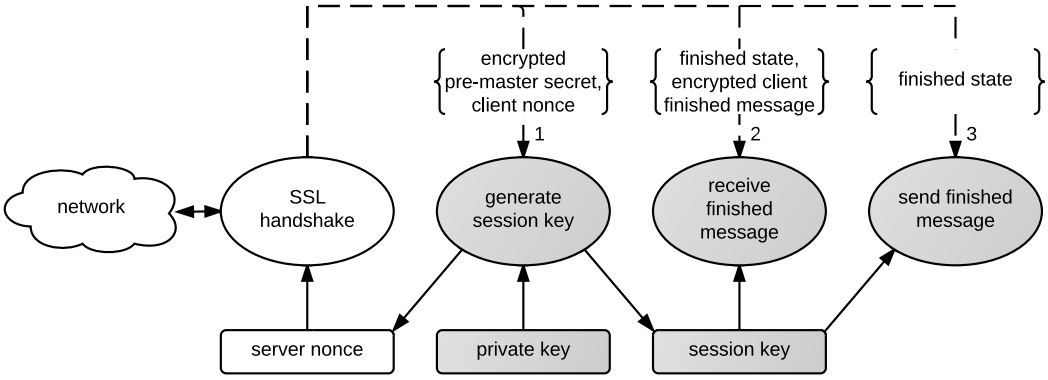


Figure 3.6: SSL-Enabled Apache: SSL Handshake Partitioning.

compromise. Otherwise, an attacker may succeed with the session key disclosure. Figure 3.6 shows the partitioning of the SSL handshake phase. The operations of the *generate session key* callgate are the same as described above in Figure 3.4 with the difference that the unprivileged *SSL handshake* sthread gets no access to the computed session key.

The SSL handshake phase must use a session key twice, to process a client's final finished message and to produce a server's final finished message, Figure 3.1. A final finished message is an encrypted hash of all previously exchanged SSL handshake messages and the session key. The server decrypts a client's finished message and verifies that it matches the locally computed one. In this way the server tests that the inbound encryption and MAC keys are correct and that no one has tampered with the messages during the handshake. Finally, the server must produce its own final finished message for the client.

Encryption and decryption operations require access to a session key; therefore, the unprivileged SSL handshake sthread is provided with callgates to perform these operations. However, callgates that simply encrypt and decrypt data with the session key result in encryption and decryption oracles. With these oracles, an attacker cannot disclose the session key, but she may use them to decrypt user sensitive data and encrypt arbitrary messages. Our callgates avoid oracles by performing more sophisticated operations than simple encryption and decryption, and the operations fall within correct functioning of the SSL protocol.

To process a client's final finished message, the *receive finished message* callgate takes the encrypted final finished message and the current server finished state from the sthread, hashes the finished state and the session key producing a server final finished state, decrypts client final finished message and compares it with the server final finished state. The callgate returns a binary result of the comparison, true or false, to the sthread.

The receive final finished callgate avoids a decryption oracle by not returning cleartext messages to its caller; however, one may suggest that this compartment is vulnerable to dictionary-like attack. In particular, an attacker may supply an encrypted message instead of an encrypted final finished message and a cleartext guess of the message's content as a server finished state. The callgate reveals if the guess

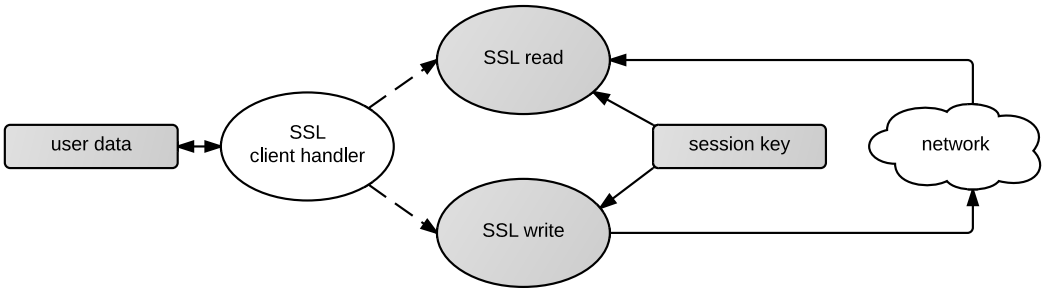


Figure 3.7: SSL-Enabled Apache: SSL Client Handler Partitioning.

matches the cleartext message. Such a dictionary attack is not possible because the callgate hashes the server finished state and the session key before the comparison operation, and the hashing converts the attacker guess into a random value.

To produce a server final finished message, the SSL handshake sthread employs the *send finished message* callgate. The callgate takes a single argument, a server finished state, hashes the state and the session key thus computing the server final finished message, encrypts the message, and returns the result to the sthread. Again, this callgate cannot be used as an encryption oracle since the compartment hashes supplied data before the encryption.

After the SSL handshake sthread computes the session key and processes final finished messages, it exits, and the SSL client handler phase starts with a new sthread. Figure 3.7 shows the partitioning of the SSL client handler phase. The SSL client handler sthread uses two callgates, *SSL read* to decrypt incoming messages and *SSL write* to encrypt outgoing messages. SSL client handler may manipulate user sensitive data although it is untrusted and may contain vulnerabilities. This is safe as the compartment is protected with SSL's MAC and encryption. For an attacker to compromise SSL client handler, she must pass a validly encrypted and MACed exploit through the SSL read callgate. It is unlikely that the attacker succeeds in encrypting and MACing the exploit without the knowledge of the session key. Even if an attacker compromises the privileged SSL read compartment (although it is against our threat model), the attacker cannot disclose the session key directly as the compartment lacks write access to the network, and it is restricted from opening new connections by Wedge. Additionally exploiting the SSL client handler sthread is no use for the attacker as any information sent out of the server is encrypted. Thus, the attacker must compromise SSL write in order to leak the session key.

An attacker may compromise the SSL client handler sthread that serving her own connection, but the compromised sthread is limited to access only the attacker's data. An SSL server may be allowed to retrieve data from a database which stores sensitive data of multiple users. In this case, access to the database must be safeguard with further partitioning of the server.

3.5 Evaluation of Wedge-Partitioned SSL-Enabled Apache.

3.5.1 Security Analysis

In our privilege-separated Apache web server, we isolate unprivileged compartments with sthreads and insure that they are restricted from accessing sensitive data. Consider the SSL handshake compartment in Figure 3.6; the compartment is denied access to the server's private key and the computed session key. The keys are stored in tagged memory regions which are not mapped into the untrusted sthread's address space. The sthread may still use these keys through the trusted callgates.

We confine untrusted compartments with restrictive SELinux policies which prevent them from issuing system calls such as to open a socket or unauthorized file and to ptrace a process. In the case of the SSL handshake compartment, the file system access is completely denied, and the SSL client handler may open only files hosted in the `/var/www` directory. Moreover, similar to the baseline Apache web server, the untrusted sthreads run under unprivileged UID and GID which further restrict them from opening users' files and tempering with the users' processes.

Wedge's privilege reduction mechanisms make it impossible for an attacker compromising an untrusted compartment to disclose and corrupt sensitive information in trusted compartments/callgates and stored on the system.

3.5.2 Porting Effort

Initial porting of the Apache web server without an aid of Crowbar tools required significant effort. In particular, substitution of worker processes with worker sthreads required identifying 222 heap objects and 389 globals. Missing even one of them results in a protection violation and crash under Wedge's default-deny model. To identify these objects, we applied run-and-fail technique where each run of the server leads to a crash, and each crash identifies a single memory object that must be shared with the failing sthread. In order to tag the identified object, we had to manually search for a place in the source code where the object is allocated. These porting challenges motivated the design of Crowbar tools, which greatly ease identifying these objects and their allocation sites, as well as smalloc_on/off and globals' tagging mechanisms.

3.5.3 Trusted Code Base

Our fine-grained partitioning of the Apache web server includes nine compartments, unprivileged sthreads and privileged callgates. The partitioning ensures safety of user sensitive data and the SSL protocol sensitive data, *i.e.*, a private key, pre-master secret, and session key, from corruption and disclosure. In the privilege-separated Apache web server, $\approx$16K lines of C code execute in privileged callgates, and $\approx$45K lines of C code in unprivileged sthreads (including comments and empty lines). In the baseline Apache web server, all code runs as privileged; thus, our partitioning reduced the trusted code base just by under two-thirds. To implement our partitioning, we made changes to $\approx$1700 lines of code, which is only 0.5% of the total Apache and OpenSSL code base.

Experiment	Baseline	Wedge	Recycled
all sessions cached (req/s)	1238	238 (5.20x)	339 (3.65x)
no sessions cached (req/s)	247	132 (1.87x)	170 (1.45x)

Slowdowns relative to the baseline server are in parenthesis.

Table 3.1: Performance of Wedge-Partitioned SSL-Enabled Apache.

3.5.4 Performance Evaluation

We compared the performance of the privilege-separated Apache web server against the performance of the baseline Apache web server by measuring the maximum load in requests per second a server can sustain when stressed from multiple clients requesting a small static page over the SSL protocol. We looked at two implementations of the privilege-separated web server, one with conventional callgates where each callgate invocation requires creating a new sthread, and the other one with recycled callgates where single sthread creation is amortized among multiple callgate invocations. The detailed description of recycled callgates is in Section 2.4.1.

Performance of the Apache web server heavily depends on user workload, in particular, whether a client enables SSL session caching. When terminating an SSL session, a client and the server may cache the symmetric keys used for the session. Later, if the same client initiates a new SSL session, it may request the server to reuse the cached symmetric keys for the new session. Thus, SSL caching allows to omit computationally expensive cryptographic operations of the RSA key exchange protocol and significantly improves the server's throughput. In our evaluation, we looked at two experiments: clients requesting a static page with SSL caching disabled and clients requesting a static page *only* via cached SSL sessions. In the former test case, every HTTP request results in establishing a full SSL session (with the expensive cryptographic operations). In the latter test case, the server does not perform RSA cryptography at all.

Table 3.1 shows the maximum throughput in requests per second that our implementations of the privilege-separated web server achieve. *Wedge* is the implementation with the conventional callgates, and *Recycled* shows the performance of the implementation with the recycled callgates. The *Baseline* column refers to the unmodified Apache web server. When all SSL sessions are cached, Apache's partitioning based on conventional callgates introduces 5.20x slowdown, and the partitioning with recycled callgates shows 3.65x slowdown. When no SSL sessions are cached, privilege-separated Apache reduces the server's throughput by 1.87x and 1.45x, respectively. Note that the experiment with cached sessions is the worst workload for us which best shows the cost of our partitioning; however, it is an artificial workload and does not correspond to any real one. A real workload should have a mix of cached and non-cached SSL sessions. Moreover, servers usually dispatch dynamic content over SSL; thus, they spend significantly more time generating a page than it takes to retrieve a static page. The cost of generating the dynamic content and the cost of RSA operations in the SSL handshake should amortize the overhead of our partitioning in the real-life workloads.

The cost of partitioning is proportional to the number of sthreads and callgates created and invoked per request. In our implementations of the privilege-separated Apache web server, each request creates

two sthreads and invokes eight callgates (nine, for non-session-cached clients), a few callgates are invoked more than once per request. Baseline Apache instead uses a pool of (reused) workers, so it does not pay process creation overhead on each request. The cost of partitioning can be reduced with recycled callgates as demonstrated by our evaluation of Apache's implementation with recycled callgates. In the experiment with cached sessions, the server's slowdown drops from 5.20x to 3.65x when conventional callgates are replaced with recycled callgates.

3.6 Conclusion

Wedge-partitioned Apache is our contribution to Wedge. Splitting the Apache web server revealed the need for Wedge's partitioning tools: the Crowbar tools, smalloc_on/off, and tagging globals. The work on Apache identified a sophisticated attack, the session key disclosure attack, which allows a remote attacker to disclose users' sensitive data from the server. We demonstrate that simple isolation of sensitive data from unprivileged compartments may not be sufficient to prevent subtle attacks such as in the case of the private key oracle. Our Wedge-partitioned Apache web server is resistant to user data disclosure, private key disclosure, and session key disclosure attacks which aim at disclosure and corruption of users' sensitive data processed by the server. Performance evaluation of the privilege-separated Apache web server shows that it is possible to achieve strong security guarantees for user sensitive data with Wedge at acceptable performance cost.

Wedge-partitioned Apache was a starting point for the work on privilege-separating cryptographic protocol implementations and designing general, protocol-agnostic partitioning guidelines that suggest defenses against subtle attacks on protocol implementations.

Chapter 4

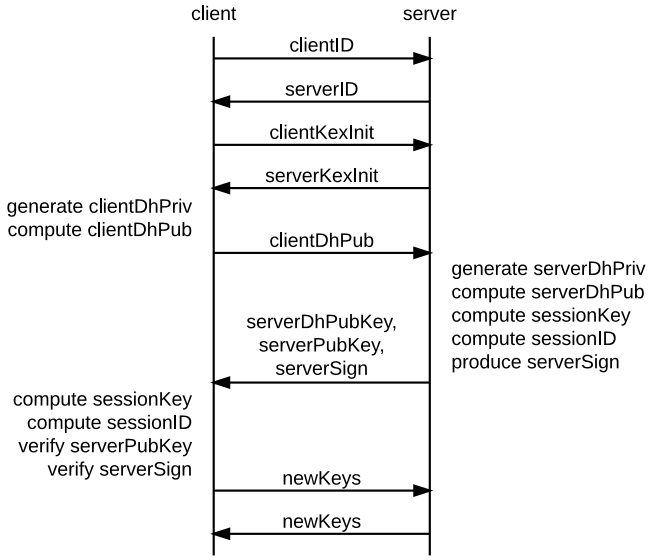
Structuring Protocol Implementations

Our work on applying Wedge to partition the SSL-enabled Apache web server showed that privilege-separating the SSL protocol is a nontrivial task. In this chapter, we further investigate the problem of how to structure implementations of cryptographic protocols to protect sensitive data even in the case when the implementations are compromised.

Applications that deal with highly sensitive data employ cryptographic protocols to protect this data from unauthorized disclosure and corruption when transferring it over an untrusted network. The protocols are designed to thwart eavesdropping and modification of data in transit, and impersonation of communicating parties; however, the protocol defenses are as secure as human-written protocol implementations. Frequent bugs and vulnerabilities [22, 23, 31, 53] in widely used cryptographic protocol implementations are a serious cause for an alarm. An exploit targeting a vulnerability in a protocol implementation may disclose and corrupt sensitive data despite the fact that the protocol's abstract design is correct.

There is a number of approaches that aim at preventing disclosure and corruption of sensitive information from remote attacks caused by vulnerabilities and exploits in network applications including those that use cryptographic protocols. The state-of-the-art systems, the privilege-separated OpenSSH server [118] and the HiStar-labeled SSL web server [137], discussed in Chapter 2 enforce the PoLP and partition the applications into multiple reduced-privilege compartments where a compromise of an unprivileged compartment should not allow sensitive information disclosure and corruption.

Our findings show that none of the state-of-the-art systems adequately partition protocol implementations, and the inadequate partitioning exposes them to novel sophisticated attacks, the session key disclosure and oracle attacks, that we have discovered [106]. As a solution to the partitioning problem, we propose protocol-agnostic principles that can guide programmers in splitting their protocol implementations to build robust protocol structures and thwart our novel attacks. As evidence of the practicality and generality of these principles, we present restructured implementations of the OpenSSH server and client and of the OpenSSL library that limit privilege so as to protect users' sensitive data from an adversary who can remotely exploit the implementation. This restructured OpenSSL library is of particular interest because it can act as a drop-in replacement for the stock library and secure a range of security-conscious applications. The OpenSSL library is an example of how the partitioning effort can be amortized across



p – a large safe prime; g – a generator for a subgroup of $GF(p)$; q – the order of the subgroup $GF(p)$; **clientDhPriv** – x where x is a random number in a range $1 < x < q$; **serverDhPriv** – y where y is a random number in a range $1 < y < q$; **clientDhPub** – $e = g^x mod\, p$; **serverDhPub** – $f = g^y mod\, p$; **sessionKey** – $y^e mod\, p = e^y mod\, p$; **serverPubKey** – a server's public key; **serverPrivKey** – the server's private key; **sessionID** – Hash(clientID || serverID || clientKexInit || serverKexInit || serverPubKey || clientDhPub || serverDhPub || sessionKey); **serverSign** is a signature produced on sessionID with serverPrivKey.

Figure 4.1: Diffie-Hellman Key Exchange within the SSH Protocol.

multiple applications. This chapter includes the following contributions:

- Two novel classes of attacks on cryptographic protocol implementations, session key disclosure and oracle attacks, that allow disclosure and corruption users' sensitive data even in the state-of-the-art reduced-privilege applications, the OpenSSH server and the HiStar-labeled Apache web server.

- Protocol-agnostic principles for partitioning protocol implementations to protect against session key disclosure and oracle attacks.

- Implementation of the OpenSSH server and client restructured in accordance with our partitioning principles and thus immune to session key disclosure and oracle attacks. Restructured implementation of the OpenSSL library (client and server sides) that provides similar guarantees, and can act as a drop-in replacement for the stock OpenSSL library bringing robustness against these attacks to a wide-range of SSL-enabled applications.

4.1 Background: SSH Protocol

This section presents details of the SSH protocol [134] which we use to demonstrate challenges encountered when partitioning cryptographic protocols and our proposed solutions.

There are three stages in the SSH protocol. The first one is performing a key exchange protocol that secretly negotiates a shared session key between a client and server and authenticates the server. Then the client and server enable message authentication (MAC) and encryption on the communication channel and start the second stage, client authentication. The final stage is an SSH session where the

server provides a user shell to the client.

SSH uses the Diffie-Hellman (DH) key exchange protocol [135]. Figure 4.1 shows the messages of the Diffie-Hellman key exchange protocol within SSH. A client initiates key exchange by sending its *clientID* message. The client and server identification messages notify the parties of SSH versions and software versions. The *clientKexInit* and *serverKeyInit* messages are issued to negotiate various SSH parameters, *e.g.*, a key exchange protocol, an encryption algorithm, a message authentication algorithm, a compression algorithm, *&c*. After both parties agree on algorithms, the client generates a private DH component, *clientDhPriv*. The private component is essential for computing a session key, and the secrecy of the session key depends on the the secrecy of the component. Moreover, since this private component is generated afresh for each SSH session, it plays a role of a client random nonce similar to the client nonce in the RSA key exchange protocol, Section 3.1. Using the private component, the client computes the public DH component, *clientDhPub*, and shares it with the server. clientDhPub is not sensitive information and transferred in clear-text. Note that it is not possible to recover the private DH component from the public DH component due to noninvertibility of the cryptographic operations that produce the latter component.

Similarly to the client, the server generates its own private DH component and computes its public counterpart. Again, the server's private component is responsible for the secrecy of the session key and introduces the server randomness to the key exchange protocol. A combination of a private DH component and another party's public DH component allows to compute a shared session key. Thus, the server computes the session key using its *serverDhPriv* and *clientDhPub*, and the client uses clientDhPriv and serverDhPub components to compute the same session key.

In the Diffie-Hellman key exchange protocol, a server authenticates itself to a client by proving possession of a private key. With the private key, the server signs a session ID which is a hash of following components: client and server IDs, algorithm negotiation messages, client and server public DH components, a server public key, and a session key. The server forwards its public DH component (serverDhPub), public key (serverPubKey), and signature over a session ID (serverSign) to the client.

The client computes a session key from its private DH component and a server public DH component, then it computes its own session ID that must match the server's one. To authenticate the server, the client checks if the signature on the session ID is valid. Before that, the client must verify that the provided public key indeed belongs to the server. There are two methods of public key verification: certificate verification and local database verification. If a certificate is provided, it is signed by a trusted authority and incorporates the server public key. The client must validate the signature on the certificate and extract the public key. In the case of the local database, the client keeps a list of trusted (previously approved by a user) public keys in a file. To verify a public key, the client checks if the key is present in the local database. When the public key (serverPubKey) is verified, the client may validate the signature on the session ID, serverSign. If the validation passes, the server is authenticated successfully.

A session ID incorporates server and client per-session randomness to prevent an attacker from bypassing server authentication by replaying previously eavesdropped signatures. The randomness guar-

antees that a session ID and its signature are unique for every SSH session. Server and client private DH components provide this per-session randomness; they are generated afresh for every SSH session and included in public DH components. The public DH components are part of the session ID's hash, Figure 4.1. The session ID includes other components to guarantee that they were not modified while in transit.

The final *newKeys* messages signal to parties that they must enable message authentication and encryption on the channel. Thus, all the following messages are MACed and encrypted with symmetric keys derived from the session key.

Following the Diffie-Hellman key exchange protocol, a client authenticates to a server with one of the following authentication methods: password authentication, public key authentication, and host based authentication [133]. The password authentication requires a client sending its username and password to the server within a MACed and encrypted message. Note that it is safe for the client to share its password with the server as the latter party has already been authenticated. The server completes the authentication by checking if the system has a user with the provided username, and the password is correct.

The public key authentication requires a client to sign a session ID and some additional items: a username, a public key algorithm, and the client's public key. The client produces the signature with its private key, and the server validates it with the provided public key and a locally computed session ID. Before signature validation, the server must verify the client's public key using a certificate or a local database. The key verification process is the same as the one discussed above.

The host based authentication is not suitable for high-security parties as the authentication is based on the host a user originates from. We omit the host based authentication in this work and do not discuss it here.

The final stage involves a server providing a service for a client, *e.g.*, a remote shell, port forwarding, X11 forwarding, *&c.*, over a MACed and encrypted channel.

4.2 Threat Model

Our goal is to privilege-separate network-facing applications that employ cryptographic protocols to protect their sensitive data: users' sensitive data, session keys, and applications' private keys. We also aim at preventing the session key disclosure and oracle attacks, described below.

Our threat model assumes a remote attacker who can compromise unprivileged compartments of a protocol implementation within an application and inject arbitrary code. The injected code executes with privileges allowed by the compromised compartments and may disclose and corrupt any information available to these compartments. In our threat model, privileged compartments must be manually checked against vulnerabilities by security experts so to prevent their compromise and disclosure of sensitive information. The attacker cannot escape privilege restrictions imposed on compartments; thus, she cannot exploit the system enforcing compartment privileges, the operating system, and any other software running on the same machine as the target application employing the cryptographic protocol.

We consider two specific classes of attacks: the first is that of the *Session Key Disclosure* (SKD)

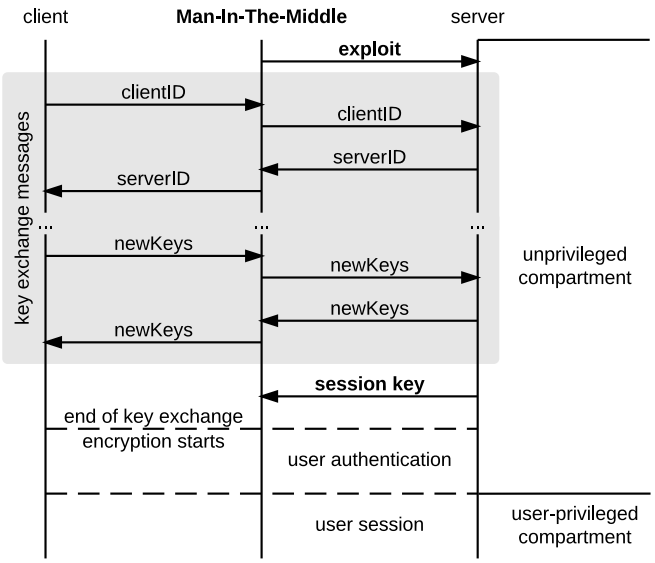


Figure 4.2: The Session Key Disclosure Attack against the Privilege-Separated OpenSSH Server.

attack where a man in the middle exploits an unprivileged compartment and compromises a session key, the second is *oracle attacks* where an attacker exploits an unprivileged compartment and misuses operations on sensitive data which a privileged compartment (isolating the sensitive data) exposes to the compromised compartment. Abusing these operations may allow the attacker to impersonate a server or a client, learn a session key, encrypt and decrypt arbitrary messages. Our partitioning schemes protect from attacks targeting disclosure of users' sensitive data and private keys described in Section 3.2. However, we do not address other attacks such as denial-of-service attacks.

4.2.1 Session Key Disclosure Attack

In Section 2.1, we discussed the partitioning structure of the OpenSSH server that prevents an attacker from obtaining root privilege in case of compromise of unprivileged compartments. Achieving this goal allows to protect a system running the OpenSSH server form compromise, but it is not sufficient to protect user sensitive data, the primary goal of the SSH protocol, as we demonstrate next with our session key disclosure attack. We discovered this attack when privilege-separating the Apache web server with Wedge, Section 3.2, where the attack targets the SSL protocol. We realized that the SKD attack is general and may target any cryptographic protocol that involves negotiating of shared symmetric keys. In this section, we show how the attack applies to the SSH protocol within the OpenSSH server.

The SKD attack involves an active man in the middle residing on the communication path between a client and a server. Figure 4.2 demonstrates the SKD attack on the SSH protocol in the privilege-separated OpenSSH server, Section 2.1. The attacker begins by exploiting the server's unprivileged compartment, Figure 2.1. When the client and server perform key exchange protocol, the attacker relays all their messages allowing the legitimate parties to establish a shared session key. After the key negotiation, the attacker's injected code sends the session key to the attacker from the compromised unprivileged process.

The client and server use the compromised session key to encrypt messages following the key exchange protocol. Thus, the attacker may reveal a user password if the server performs password authentication after the key exchange. The attacker may decrypt user messages and disclose clear-text data as well as validly encrypt forged messages. The session key disclosure attack succeeds because the unprivileged process of the OpenSSH server is allowed access to the session key.

The state-of-the-art HiStar-labeled SSL web server discussed in Section 2.3 states its gaol as protecting user sensitive data from unauthorized disclosure. However, authors of the server did not consider our SKD attack. A session key is computed within SSLd, an unprivileged compartment that can be compromised by a man-in-the-middle attacker, Figure 2.2; thus, the SSL server is vulnerable to the SKD attack in the same way as the privilege-separated OpenSSH server.

4.2.2 Oracle Attack

The HiStar-labeled SSL web server, shown in Figure 2.2, isolates the server's private key by labeling it with sensitivity level K. If an unprivileged compartment gets compromised and accesses the private key, the compartment will be tainted with the key's label, and DIFC will prohibit disclosing the private key to the network and less tainted compartments. The privileged RSAd compartment performs operations with the private key; it possesses an ownership over the key's sensitivity level, Figure 2.2, so it can perform the operations and declassify the results. The server must authenticate itself to a client, and RSAd uses the private key to decrypt a pre-master secret or sign an ephemeral RSA key or sign a public Diffie-Hellman component [74, 82]. The data to decrypt or sign depends on the negotiated key exchange protocol (RSA, ephemeral RSA, Diffie-Hellman) and is supplied by the unprivileged SSLd compartment.

Despite the isolated private key, it is possible to abuse the decryption and signing operations. An attacker controlling the unprivileged SSLd compartment may supply arbitrary data for decryption and signing. SSLd is intended to decrypt or sign authentication data that corresponds to its current SSL session. However, the attacker can force it to supply authentication data that corresponds to other SSL sessions where he wants to impersonate the server. By abusing the decrypt and sign operations in the HiStar-labeled web server, the attacker can impersonate the server and trick users to send sensitive data to him. This example demonstrates that simply putting sensitive data beyond direct reach of unprivileged code does not provide sufficient protection.

We name such attacks against cryptographic protocol partitioning oracle attacks. Any privileged compartment or sequence of privileged compartments isolating sensitive data and exporting privileged operations to unprivileged code can be an oracle. By abusing privileged operations, an attacker can obtain sensitive information. In our example above, a pair of clear-text authentication data and corresponding signature matching a benign SSL session can be considered sensitive information as it allows server impersonation.

In fact, the privilege-separated OpenSSH server [118] suffers exactly from the same oracle because the monitor exporting a sign operation to the unprivileged process signs arbitrary data supplied by the process. An attacker controlling the unprivileged process can impersonate the OpenSSH server by exploiting this oracle.

Oracle	Description	Attack	Applicability
Private key decryption	decrypts arbitrary data with a private key	server impersonation	SSL
Deterministic session key	generates a chosen session key	session key disclosure, replay messages from a past session	SSL
Signing	signs arbitrary data	server/client impersonation	SSL, SSH
Signature verification	bypasses signature verification	server/client impersonation	SSL, SSH
Session key encryption	encrypts arbitrary data with a session key	forge encrypted messages within a session	SSL, SSH
Session key decryption	decrypts arbitrary data with a session key	disclose sensitive data within a session	SSL, SSH

Table 4.1: Oracles in Protocol Implementations.

In our work on structuring protocol implementations, we identified other oracle attacks. Digital signatures suffer not only from signing oracles, but also from verification oracles, in which an attacker can force successful signature verification by supplying chosen inputs to a privileged compartment performing the signature verification. There also exists an oracle where an attacker forces a set of privileged compartments generating a session key to produce a key used in a previous user session; we name this oracle a *deterministic session key oracle*. Forcing reuse of a session key allows an attacker to replay messages from a past session. This particular threat exists in SSL's RSA key exchange protocol. Finally, encryption and decryption oracles may allow an attacker to encrypt arbitrary data and decrypt confidential messages. Table 4.1 summarizes the different kinds of oracles that may appear in cryptographic protocol implementations.

4.2.3 Generality of Attacks and Partitioning Principles

Oracle and SKD attacks are independent of isolation primitives used to limit privilege in privilege-separated applications. As shown above, they are successful against applications privilege-separated with processes and applications confined with DIFC. The reason these attacks are successful is because of weak structuring of protocol implementations.

The attacks are protocol-agnostic, and they are equally a threat for SSL, SSH, IPSec, *&c.* The SKD attack targets cryptographic protocols that employ key exchange techniques to secretly negotiate a shared symmetric key, and we believe many cryptographic protocols use symmetric keys. Oracle attacks target cryptographic building blocks such as signing, signature verification, encryption, decryption, and message authentication rather than the protocols themselves. Thus, if a protocol performs a signing operation with a private key, it may be vulnerable to the singing oracle attack.

Our guiding principles, presented next, are designed to thwart oracle and SKD attacks which makes the principles protocol-agnostic and relevant for any defense mechanism applied.

4.2.4 Completeness of Attacks

Here, we attempt to answer whether we have identified all disclosure and oracle attacks on cryptographic protocol implementations. There are typically three classes of sensitive data in a system employing a cryptographic protocol: a private key mostly used for authentication, session key used for protecting

confidentiality and integrity of a user's sensitive data, and the user's sensitive data. We identified three attacks (user data disclosure, session key disclosure, and private key disclosure) that target disclosure of each class of sensitive data. The user data disclosure and private key disclosure attacks are described in Section 3.2, and the session key disclosure attack is presented above. We are confident that we have identified all of the disclosure attacks unless a protocol introduces a new class of sensitive data.

Oracle attacks target cryptographic primitives such as encryption, decryption, signing, signature verification, message authentication, *&c.* We looked at all of these operations and identify threats that can arise when these operations are isolated within privileged compartments. We are certain that we have identified all oracles, Table 4.1. However, our reasoning is based on analytical study, and we cannot rule out a possibility that we may have missed an oracle attack.

4.3 Principles for Protocol Partitioning

This section defines principles for protocol structuring that aim at guiding programmers in privilege-separating cryptographic protocol implementations. Following these principles makes the implementations robust against the protocol attacks presented above event when unprivileged compartments are compromised. These principles are valid within any privilege enforcement mechanism such as privilege separation with processes [118], sthreads [66], and DIFC [77, 94, 136]. The principles are equally apply to client and server applications.

4.3.1 Two-Barrier, Three-Stage Partitioning

Cryptographic protocols must share a symmetric session key for encryption, decryption, and message authentication (MAC) which is necessary to provide confidentiality and integrity guarantees for transfered data. In addition, the protocols must ensure that the parties at the end points are the ones they claim. The protocols perform client and server authentication where each party proves its identity by demonstrating possession of a secret. There are multiple schemes for authentication, some of them depend on transferring confidential data such as passwords and thus require a session-key-encrypted channel to be established in advance, others may perform authentication in clear-text, *e.g.*, public key authentication and challenge-response authentication. Client authentication in SSH [134] fits the first group as the protocol is designed to support different client authentication schemes including the password authentication. SSL/TLS [74, 82] supports only public key authentication and therefore belongs to the second group.

To prevent SKD and impersonation attacks, an application employing a cryptographic protocol must implement structures that we name a *session key barrier* and a *user privilege barrier*. These barriers divide protocol implementation into three stages, Figure 4.3. The *session key negotiation* stage performs a session key exchange protocol and results in a symmetric session key. The following stage, the *pre-authenticated stage*, is dedicated for peers' authentication. Finally, after the authentication, the application transitions into the last stage, the *post-authenticated stage*, where it processes user requests. A single unprivileged compartment represents each stage, the compartment is restricted from accessing sensitive information. A number of privileged compartments isolating sensitive data may export privi-

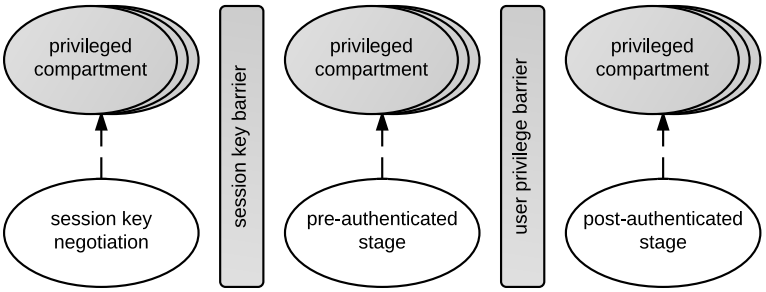


Figure 4.3: Barriers and Stages in Protocol Partitioning.

leged operations on the sensitive data for the unprivileged compartments. Threat models are different for each stage as well as the scope of sensitive data—the same piece of data may be sensitive in one stage and nonsensitive in another.

The Session Key Barrier

The session key barrier denotes killing of an unprivileged compartment performing the session key negotiation, spawning a new unprivileged compartment to operate in the following pre-authenticated stage, and enforcing MAC on the channel. The killing is necessary to ensure that the newly-spawned compartment is not under an attacker's control when it starts to operate, and the MAC protects the compartment from the attacker's exploits.

The unprivileged compartment of the session key negotiation stage is the only compartment that processes clear-text messages which are not MACed. Although these messages must be issued by a client, it is not possible to guarantee their origin and integrity. It is quite possible that a clear-text message is issued or modified by an attacker and carries an exploit for the unprivileged compartment of the session key negotiation stage. This unprivileged compartment must not have access to sensitive information including the computed session key. As we demonstrated with the SKD attack, the session key is sensitive information in the context of the session key negotiation stage. In addition, any data that allows deriving the session key such as a private Diffie-Hellman component (in the case of the Diffie-Hellman key exchange [134]) or a pre-master secret (in the case of RSA-based session key establishment in SSL [82]) must be isolated from the unprivileged compartment. In fact, the unprivileged compartment does not need access to the session key as it deals only with clear-text messages. Any operations that require access to sensitive data must be provided through privileged compartments.

> **Principle 1:** A network-facing compartment performing session key negotiation should not have access to a session key, nor any data that allows deriving the session key.

Because the unprivileged compartment of the session key negotiation stage can be compromised, we do not allow it to run in the pre-authenticated stage where it would have access to the session key, we kill it (*i.e.*, kill the Linux process).

An unprivileged compartment in the pre-authenticated stage processes only encrypted and MACed messages with the session key. Because of the MAC, the SKD attacker cannot compromise this stage

as he must validly MAC an exploit for this stage which is not possible unless the attacker knows the session key. The access to the session key was restricted in the session key negotiation stage; therefore, the attacker has no way to MAC the exploit. We rely on validity of the MAC function because only this code may deal with the attacker's input. We require this code to be carefully audited and secured from vulnerabilities.

Both the killing of the unprivileged compartment in the session key negotiation stage and the MAC on the channel compose the session key barrier which prevents SKD attacks. In order to guarantee integrity of the pre-authenticated and post-authenticated stages against the SKD attacker, there should be no unauthenticated messages beyond the session key barrier.

> **Principle 2:** When enabling the MAC, a network-facing compartment performing session key negotiation should be killed, and a new one created with privilege to access the session key.

> **Principle 3:** After enabling the MAC, there should be no unMAC'ed messages processed by the unprivileged compartment.

The session key barrier eliminates the SKD threat; therefore, the session key is not sensitive in the pre-authenticated and post-authenticated stages. We allow unprivileged compartments of these stages to access the session key directly.

The Diffie-Hellman key exchange in SSL and TLS requires a server to present a signature over a server public DH component to a client for authentication. Using the provided server public key, the client validates the signature and thus authenticates the server. The signed data must include client per-session randomness to guarantee that the data and its signature are produced specifically for the current session but not extracted from some other session.

In SSL 3.0, TLS 1.0, and TLS 1.1, the signed data does *not* include client randomness which renders these protocols to the following weakness. An attacker may establish an SSL/TLS session with a server and collect a server public DH component and corresponding signature when the server authenticates to the attacker. Later, the attacker may use this pair of data items to impersonate the server. At client authentication, the attacker may present the public DH component and the signature obtained earlier from the server. The authentication will pass as the signature is indeed produced with the real server private key. What stops the attacker from continuing the SSL/TLS session is that he is unable to compute the session key as he does not possess the private DH component corresponding to the signed public DH component.

If an attacker obtains a private DH component, public DH component, and a signature on the latter component, she may impersonate arbitrary SSL/TLS sessions which is equivalent of obtaining a server's private key. Note that the server sends the last two data items to the attacker when she initiates an SSL/TLS session with the server.

Within our protocol partitioning model, the private DH component is isolated from untrusted code in the key exchange stage. However, this data item is nonsensitive in the pre-authenticated and post-authenticated stages. Thus, an attacker may disclose the private DH component from the latter two

stages and render the impersonation attack described above.

To restrict the impersonation attack, a privileged compartment of the session key negotiation stage must scrub all sensitive key material, *e.g.*, a pre-master secret and a private DH component, after generating the session key.

> **Principle 4:** A privileged compartment computing a session key must scrub secret session key material after producing the session key.

TLS 1.2[1] requires a server to authenticate with signature on its public DH component, a client random nonce, and a server random nonce. The random nonces make the signature valid only within a single TLS session and rid the protocol of the weakness described above. Although Principle 4 is specific to the SSL 3.0, TLS 1.0, and TLS 1.1 cryptographic protocols, it does not harm other protocols to implement this principle.

Transitioning to the pre-authenticated stage may require moving a state from an unprivileged compartment of the session key negotiation stage to an unprivileged compartment of the pre-authenticated stage. Although both compartments are unprivileged, the threat models are different, in particular, the latter stage is not exposed to an SKD attacker because of the session key barrier. Therefore, we require the state to be validated to eliminate any exploits that the SKD attacker may planted into this data. This applies to any data passed from untrusted sources to privileged compartments, so to prevent exploitation of these compartments by an attacker.

> **Principle 5:** Any state exported from a compartment performing session key negotiation and any untrusted arguments passed to privileged compartments should be validated.

We strongly discourage protocol designers to alternate clear-text messages and MACed messages because this significantly complicates establishing of the session key barrier. However, the SSL/TLS protocol does this [74, 82], in particular it alternates the change cipher message (signaling that all the following messages will be MACed and encrypted) with the final finished message (carrying MACed and encrypted hash of all previous messages sent/received by a peer), Section 3.1. In case of such alternation, we require the session key barrier to be established after the last clear-text message. However, this presents a problem as a MACed and encrypted message needs to be processed within the session key negotiation stage where the unprivileged code has no access to the session key. A privileged compartment needs to be created to process this message, and it can result in a session key encryption/decryption oracle. We demonstrate how to avoid such oracles with Principle 8.

> **Principle 6:** A cryptographic protocol should not alternate between clear-text messages and MAC'ed messages.

User Privilege Barrier

The user privilege barrier represents any authentication method that can be used to authenticate a remote peer before granting it access to the sensitive data owned by a particular user. The pre-authenticated stage

[1] TLS 1.2 is rarely used: Google Chrome browser does not support TLS 1.2; Chase Bank, Bank of America, and Barclays Bank use TLS 1.0.

is responsible for performing such authentication. This stage is protected with the session key barrier, and there is no threat of an SKD attacker with assumption that this stage does not process unMACed messages. The unprivileged compartment of the pre-authenticated stage can access the session key directly as there is no an SKD attacker. The pre-authenticated stage can still be compromised by an attacker that aims at impersonating a user by subverting the authentication process. Therefore, the user authentication should be performed by an unprivileged compartment that does not have access to sensitive information. If the compartment requires some operations on the sensitive data, they should be provided by privileged compartments. Note that the impersonator can leak the session key, but it is his own session key that does not correspond to any other session (if the defense from a deterministic session key oracle is in place). The successful authentication transfers the application into the next stage, post-authentication stage. In this case, the killing of an unprivileged compartment is not required as the authentication ensures the identity of the peer, and it is unlikely that someone will exploit the pre-authentication stage to get access to its own data that is provided in the post-authenticated stage anyway.

Today, most privilege-separated applications implement the user privilege barrier as required. The OpenSSH server [118] performs user authentication in an unprivileged process which is restricted form accessing sensitive information such as the server's private key. The HiStar-labeled web server [137] supports only password authentication which is performed by the unprivileged HTTPd compartment with help of privileged authd. When the authentication is completed, authd grants ownership over the label of the authenticated user to HTTPd.

There are different authentication schemes; some of them can be performed in clear-text such as the public key authentication and challenge-response authentication. In case of these authentication schemes, the encrypted channel is not required, and authentication can be performed in the session key negotiation stage omitting the pre-authenticated stage. For example, SSL supports only the public key authentication; an SSL server and an SSL client can have only two stages in their implementations. Such optimization is encouraged as it reduces the number of required compartments and improves performance of an application.

The post-authenticated stage has access to sensitive data of an authenticated user and serves her requests. We note that this stage may require further partitioning. For example, when users have access to a centralized database. A compartment acting for one user should not have access to data of another user and vise versa. In this case, a central privileged monitor may be required to process requests for the database.

4.3.2 Oracle Prevention Techniques

As shown in Figure 4.3, each stage in a protocol implementation relies on a number of privileged compartments exporting operations to unprivileged compartments. With such structure, there is a risk of oracles which an attacker may abuse to subvert the protocol implementation's correct functioning and obtain sensitive data. In the session key negotiation and pre-authentication stages, an attacker may look for a private key decryption oracle, signing oracle, signature verification oracle, deterministic session key oracle, session key decryption and encryption oracles, Table 4.1. In Section 4.2.2, we show how to

exploit signing and decryption oracles in the HiStar-labeled SSL web server and the privilege-separated OpenSSH server. Clearly, we need some techniques to mitigate oracles in protocol implementations.

Entangle Output Strongly with Per-Session Known-Random Input

Network protocols employ random nonces to protect against authentication replay attacks where an active man in the middle replays authentication data, *e.g.*, an authentication signature eavesdropped in a past connection, to bypass authentication. The random nonces guarantee that the authentication data used in one session is invalid within another session as the other session uses different nonces. Authentication employs two nonces: one comes from a client and provides freshness guarantees for the client, and the second is provided by a server. In the Diffie-Hellman key exchange, a server and a client authenticate to each other by signing a session ID which incorporates the server and client's randomness (supplied from private DH components via public DH components). Thus, their signatures are valid only within this particular session, and an attacker cannot replay them within other sessions which will have different session IDs entangled with different client and server nonces.

Similarly to authentication, session key generation must include server and client random nonces to guarantee that a generated session key does not match any other key from a different session even if an attacker replays eavesdropped key exchange messages. Thus, the Diffie-Hellman key exchange uses randomness of client and server private DH components, Section 4.1, and in the RSA key exchange, a session key is a hash of a pre-master secret, server random nonce, and client random nonce, Section 3.1.

We employ the per-session, random nonce mechanism to defend against oracle attacks. If output of a privileged compartment does not completely depend on the untrusted input but incorporates a trusted, per-session random nonce, an attacker cannot replay eavesdropped inputs for this compartment and get a deterministic result of a privileged operation as the operation will use a different nonce supplied by a trusted source. This is what we name entangling the output of a privileged compartment with a trusted per-session random nonce. Entangling should involve a strong hash algorithm such as SHA [45] and MD5 [119] which guarantees that a fresh random nonce produces a nondeterministic result.

> **Principle 7:** To prevent oracles, entangle output strongly with per-session, known-random input.

Now we discuss oracles which may lead to impersonation of a party or compromise of a session. We start by demonstrating the private key decryption oracle and our defense against this oracle. We do not consider the session key disclosure attack here, and therefore our diagrams do not implement principles related to the session key barrier.

Private Key Decryption Oracle

The SSL/TLS protocol with the RSA key exchange involves a client generating a random pre-master secret, the only confidential part of a session key, which is hashed with server and client nonces to produce the session key. In order to securely share the pre-master secret with a server, the client encrypts it with the server's public key so that only the server possessing the private key can decrypt the secret. Thus, the server must perform a decryption operation to obtain the clear-text pre-master secret. Note that decrypting the pre-master secret implicitly authenticates the server to the client.

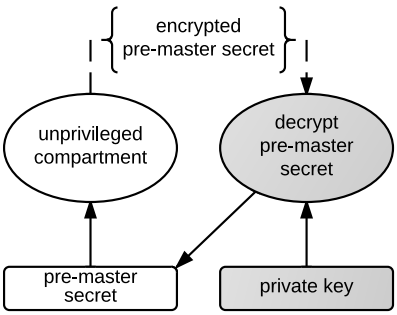


Figure 4.4: The SSL/TLS Private Key Decryption Oracle.

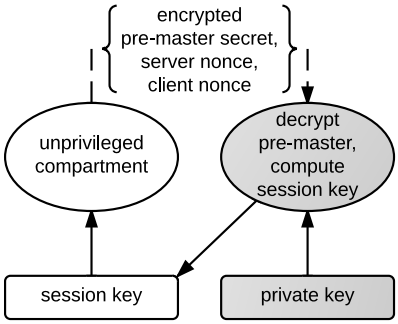


Figure 4.5: The SSL/TLS Deterministic Session Key Oracle.

Partitioning of the server-side SSL/TLS protocol may expose a private key decryption oracle; Figure 4.4 shows such naive partitioning. The private key oracle may allow an attacker to impersonate the server by misusing the decryption operation. For example, an attacker may establish a fake server and trick a client into connecting to it. Then, the client sends encrypted pre-master secret to the attacker. In order to impersonate the server, the attacker must decrypt the pre-master secret; however, she does not have the server's private key. The attacker may exploit an unprivileged compartment in the real server and invoke the private key oracle to decrypt the client's pre-master secret. Then, the attacker computes the session key and continues the SSL/TLS session on her fake server.

We restrict private key decryption oracles with per-session, known-random input. First, we show an incomplete solution where the random input supplied by an untrusted source, and the resulting scheme restricts the private key oracle but exposes the deterministic session key oracle. In the following section, we present a partitioning scheme with *known*-random input that restricts both oracles. Known-random input implies that the randomness comes from a trusted source, supplied by a privileged compartment.

Figure 4.5 shows the incomplete solution. The privileged compartment does not release a decrypted pre-master secret to the unprivileged compartment but returns a generated session key which is a hash of the pre-master secret and random nonces. Note that the encrypted pre-master secret and two nonces are supplied by the unprivileged compartment. This partitioning scheme restricts an attacker from decrypting arbitrary pre-master secrets and solves the private key decryption oracle; however, it still allows server impersonation.

Deterministic Session Key Oracle

The partitioning in Figure 4.5 restricts the private key decryption oracle, but it exposes the deterministic session key oracle which allows an attacker to impersonate a server or to force the server to generate a *chosen* session key. The impersonation requires an attacker to establish a connection with a client and collect an encrypted pre-master secret, client nonce, and the attacker's server nonce; these components are passed to the compromised unprivileged compartment of the real server, Figure 4.5. The attacker supplies the collected data to the privileged compartment and retrieves a session key that matches the key in the impersonated session. Thus, the attacker may use the generated key to continue SSL/TLS

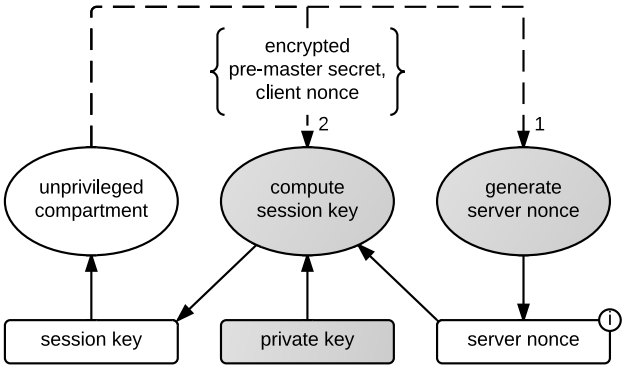


Figure 4.6: The SSL/TLS Private Key Decryption Oracle and Deterministic Session Key Oracle Defenses.

session with the client. If the partitioning implements the session key barrier, the attacker will be able to disclose the session key from untrusted compartments of the pre-authenticated or post-authenticated stages.

The deterministic session key oracle also renders the partitioning to the following *session replay attack*. An attacker eavesdrops a benign session between a client and the server and records a client nonce, server nonce, encrypted pre-master secret, and all the following SSL/TLS messages Then, the attacker establishes her own connection with the server, compromises the untrusted compartment, and supplies the three eavesdropped components to the trusted compartment that generates a session key. The trusted compartment will generate the same session key as in the eavesdropped session. The same session key allows the attacker to replay eavesdropped SSL/TLS messages.

The presented partitioning scheme is faulty because the randomness is supplied by an unprivileged compartment, but we require it to come from a trusted source. Figure 4.6 shows the correct partitioning scheme that restricts the private key decryption oracle and the deterministic session key oracle. Numbers next to privileged compartments specify the order of the compartments' invocation, and this order should be enforced by the application. An *i*-label next to a data box denotes integrity sensitive data meaning that an unprivileged compartment may read the data but not modify it. We use i-label notation instead of a read arrow from the labeled data box to the unprivileged compartment to improve figure readability.

The defense against the deterministic session key oracle forces the generated session key to be unique in each session despite the attacker-chosen inputs to the privileged compartment. The session key incorporates a known, per-session, random nonce which is generated afresh for every SSL/TLS connection by the privileged *generate server nonce* compartment. This nonce makes the session key produced by the *compute session key* compartment nondeterministic and unusable in any other SSL/TLS connection apart the one it is generated for.

Signing Oracle

In Figure 4.7 and Figure 4.8, we demonstrate the private key signing oracles for server-side implementations of the SSH and SSL/TLS protocols (excluding TLS 1.2) that are present in the privilege-separated OpenSSH server and the HiStar-labeled SSL web server, respectively. In both examples, the oracles exist

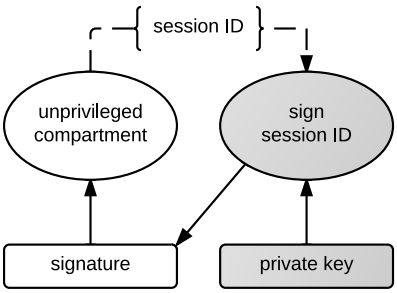


Figure 4.7: The SSH Signing Oracle.

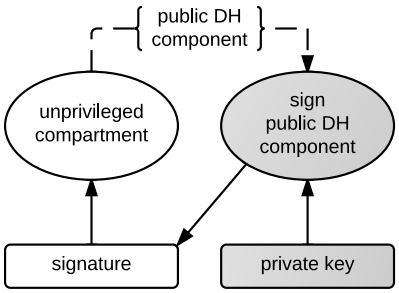


Figure 4.8: The SSL/TLS Signing Oracle.

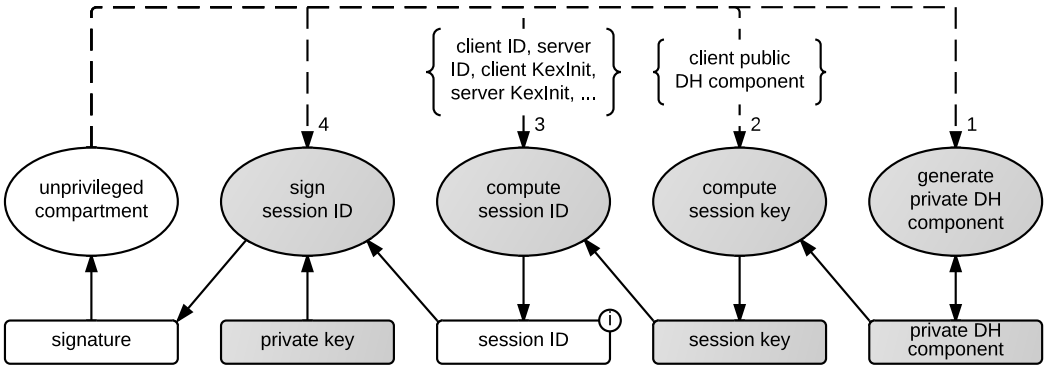


Figure 4.9: The SSH Signing Oracle Defense.

because the application partitioning schemes allow to sign arbitrary session IDs and public DH components. Thus, an attacker may produce valid signatures on authentication data, present them to clients, and impersonate the server. Similarly to the server-side implementations, signing oracles also appear in client-side implementations of cryptographic protocols where they allow client impersonation.

Figure 4.9 shows how to prevent the signing private key oracle in the server-side SSH protocol implementation. Our defense is based on restricting a privileged compartment performing signing operation to sign only data that is entangled with per-session, known-random input. The random input comes from the private DH component freshly generated by a server for every SSH session, the *generate private DH component* compartment. With a number of privileged compartments (compartments 1–3), we ensure that the session ID to be signed indeed includes the private DH component. The privileged *compute session key* compartment makes this component a part of the session key, and the session key is incorporated into the session ID by the privileged *compute session ID* compartment. Privileged *sign session ID* signs only session IDs entangled with per-session private DH components which make the produced signatures invalid in other SSH sessions.

In Figure 4.10, we present a similar partitioning scheme to protect the server-side SSL and TLS implementations form the private key signing oracle. Privileged *sign public DH component* signs a component that is entangled with per-session, random input. The *generate private DH component* compartment generates known-random input in a form of a private DH component, and the *compute public*

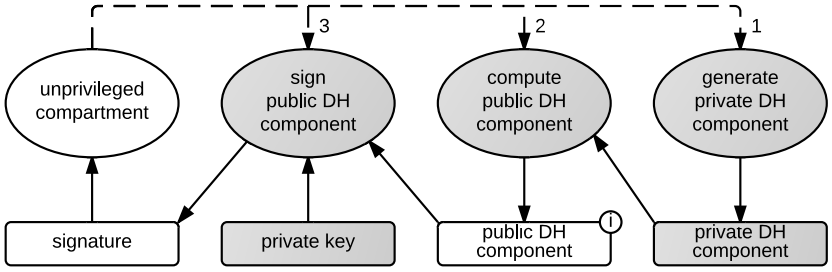


Figure 4.10: The SSL/TLS Signing Oracle Defense.

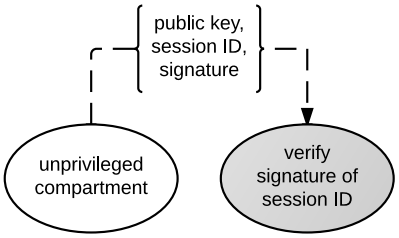


Figure 4.11: The SSH Signature Verification Oracle.

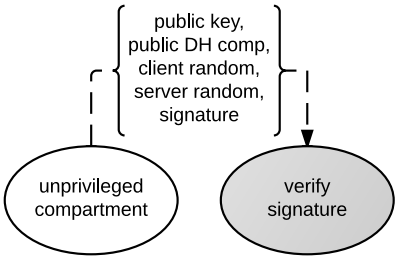


Figure 4.12: The TLS 1.2 Signature Verification Oracle.

DH component compartment entangles the signed public DH component with the randomness of the private DH component. The unique public DH components make the generated signatures valid only within current SSL sessions. TLS 1.2 produces signatures over a client nonce, server nonce, and public DH component. It is sufficient to guarantee known-random input from either the DH component or server nonce. A similar defense is applied to client-side signing oracles to restrict client impersonation attacks.

Signature Verification Oracle

Per-session random input is similarly used to prevent signature verification oracles where an attacker tries to bypass authentication by supplying chosen inputs. Figure 4.11 and Figure 4.12 show the signature verification oracles in the client-side SSH and TLS 1.2 protocol implementations. We use TLS 1.2 to demonstrate our defenses against the signature verification oracle as well as to highlight how it differs from other versions of TLS. Note, SSL and the other versions of TLS are also exposed to the signature verification oracle. An SSH client verifies a server's signature on a session ID that corresponds to their SSH session, and a TLS client uses hash of a server's public DH component with client and server nonces as data for signature verification.

There are two approaches an attacker may take to exploit a signature verification oracle and bypass authentication. The first approach proceeds in the following way. The attacker eavesdrops a signature and signature verification data (a session ID in the case of SSH, and a public DH component with nonces in the case of TLS) from a legitimate user's connection. Alternatively, the attacker may connect to the server as a client, and the server will send this pair of data to the attacker in order to authenticate itself.

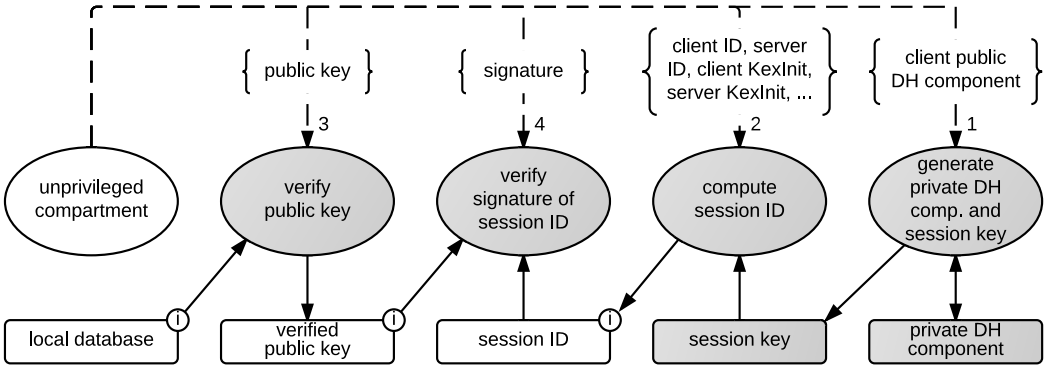


Figure 4.13: The SSH Signature Verification Oracle Defense.

Then the attacker compromises a client-side protocol implementation, Figure 4.11 and Figure 4.12, and supplies the real server's public key, the collected signature and authentication data to the privileged compartment performing signature verification. The signature verification passes as the supplied signature is indeed produced with the server's private key. The second approach is when an attacker compromises an protocol implementation and supplies her own public key, arbitrary verification data, and a signature produced over this verification data with the attacker's private key (which corresponds to the supplied public key). To restrict the signature verification oracle, it is crucial for the privileged compartment to use a verified public key and verification data entangled with per-session randomness; both inputs must come from a trusted source.

In Figure 4.13, we present partitioning that eliminates the signature verification oracle in the client-side SSH protocol implementation. The *generate private DH component and session key* compartment is a combination of compartments 1 and 2 in Figure 4.9. It produces per-session, random private DH component that gets incorporated in a session key. The *compute session ID* compartment propagates the randomness to a session ID by incorporating the session key in the ID. Thus, the session ID is known to contain per-session randomness and valid only within the current session. The privileged *verify public key* compartment checks whether a supplied public key is present in the local database and thus valid for authentication. The compartment stores a copy of the verified key in memory non-writable by the unprivileged compartment. Finally, the privileged *verify signature of session ID* compartment performs signature verification using the trusted public key, trusted verification data, and signature supplied by the unprivileged compartment. Because the verification data contains per-session randomness, and the verified public key is supplied by a trusted source, there is no signature verification oracle in this partitioning scheme.

Figure 4.14 shows a partitioning scheme of signature verification in the client-side TLS 1.2 protocol implementation that is resistant to the signature verification oracle. The server signs its public DH component hashed with client and server nonces. When a client validates the signature, the validation data is entangled with the client's per-session randomness which is the client nonce generated by the privileged *generate client nonce* compartment. Before the signature verification, privileged *verify public key* checks if the server's public key is in a certificate signed by a trusted authority and thus valid for

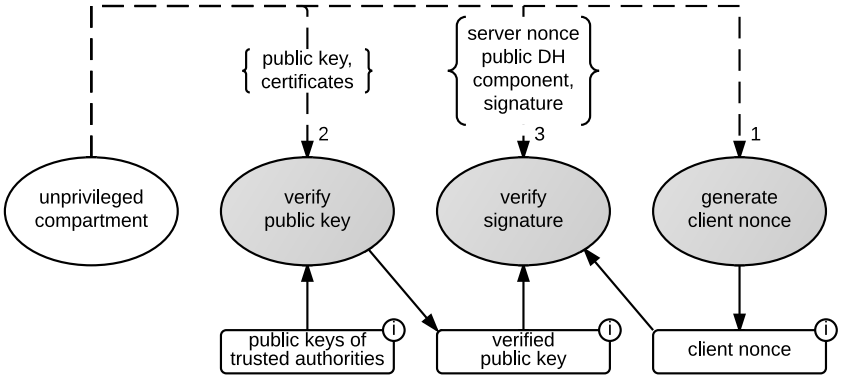


Figure 4.14: The TLS 1.2 Signature Verification Oracle Defense.

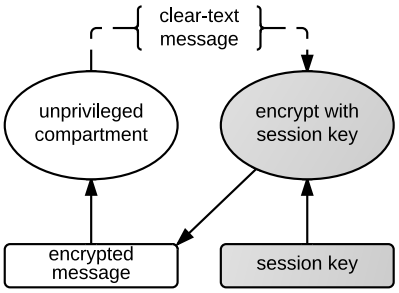


Figure 4.15: The Session Key Encryption Oracle.

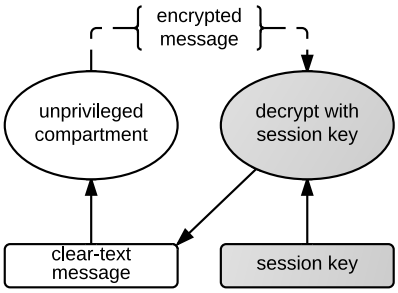


Figure 4.16: The Session Key Decryption Oracle.

authentication. The privileged *verify signature* performs signature verification with the trusted public key and client nonce, and the rest of the data, the server nonce, public DH component and signature, supplied by an unprivileged compartment.

Server-side verification oracles may exist in the SSH and SSL/TLS protocol implementations as both protocols support public key client authentication. The defense mechanisms described above apply to client-side and server side protocol implementations equally.

Obfuscate Untrusted Input by Hashing

The SSL/TLS protocol alternates clear-text messages and encrypted messages [74, 82], and according to Principle 3, there should be no unauthenticated messages beyond the session key barrier; therefore, we must enable the barrier after the last clear-text message is processed by the server's session key negotiation stage. Because of the alternation, we need to process a client message encrypted with the session key and produce another encrypted message within the session key negotiation stage. These messages are the final finish messages each of which contains a hash of all messages previously exchanged between the server and the client, Section 3.1. Note, message authentication is performed within the encryption and decryption operations.

Encryption and Decryption Oracles

Simply providing privileged compartments that perform encryption and decryption operations with the

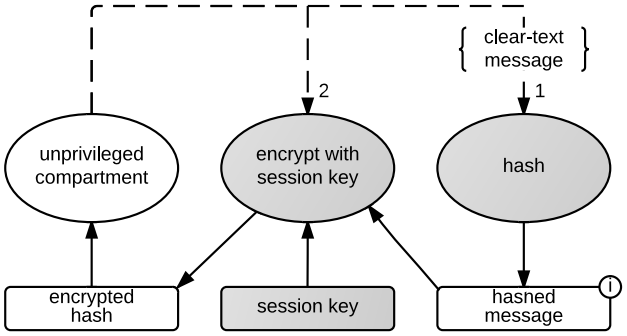


Figure 4.17: The SSL/TLS Session Key Encryption Oracle Defense.

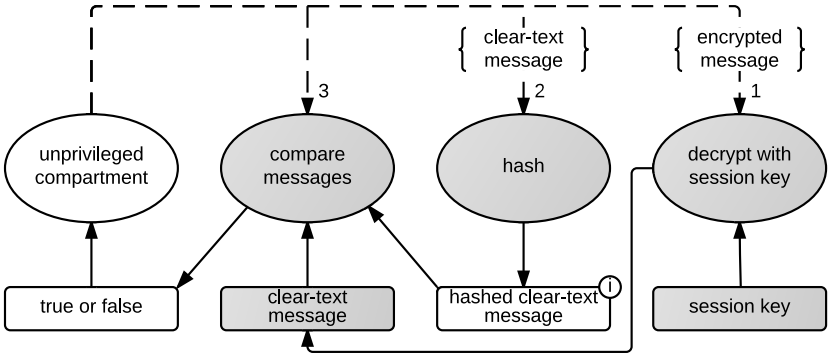


Figure 4.18: The SSL/TLS Session Key Decryption Oracle Defense.

session key results in encryption and decryption oracles that an attacker can use to obtain sensitive data and encrypt its exploits, Figure 4.15 and Figure 4.16. Our oracle mitigation technique allows the required privileged operations and avoids the session key oracles by obfuscating untrusted input with hashing. By hash we imply a strong hash algorithm such as SHA [45] and MD5 [119].

When an encrypted final finished message is produced, the privileged compartment hashes untrusted input and then encrypts it with the session key, Figure 4.17. A privileged operation that hashes data and then encrypts it is not a particularly useful for an attacker as any exploits for the pre-authenticated and post-authenticated stages will be viewed as hashes.

As for the decryption oracle, the privileged compartment performing decryption does not return the clear-text to unprivileged code. Instead, the privileged compartment takes verification data from an unprivileged compartment and performs verification by itself, the result of the verification is returned to the unprivileged compartment. However, this mechanism allows dictionary attacks, where an attacker can guess a content of an encrypted message by supplying his cleartext guesses as verification data. Again, obfuscating the untrusted validation data by hashing before comparing it with the clear-text finished message solves this problem, Figure 4.18. If an attacker attempts to guess a cleartext request, his guess will be hashed first, then compared with the original message. Hashing in the session key encryption and decryption oracle defenses fits the SSL/TLS protocol because the finished message happens to be a hash of all previous handshake messages.

Principle 8: To prevent oracles, obfuscate untrusted input by hashing.

Last Resort: More Privileged Code

Previous oracle mitigation techniques require curtain conditions enabled by a protocol such as presence of a random nonce or a hash function. Such requirements limit the applicability of these principles. In cases where there are no such conditions, we offer our last resort technique of adding more privileged code. Note that for an oracle to exist, the privileged compartment should return output to an unprivileged compartment. If a privileged compartment exposes an oracle, and none of the above techniques can be applied, the result of the privileged compartment should be treated as sensitive and processed by another privileged compartment. Thus, the output is not released to an unprivileged compartment eliminating the threat of an oracle. The drawback of the last resort technique is increased privileged code base as the new privileged compartment is required to process the output. Potentially, it can lead to a chain of privileged compartments processing their outputs, and the chain can be terminated by one of the above principles or a condition where a privileged compartment produces no output.

Principle 9: To prevent oracles, as a last resort, add more privileged code.

4.3.3 Degrees of Sensitivity

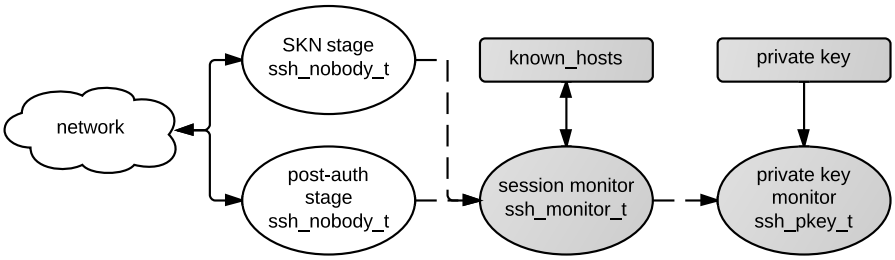
Cryptographic protocols operate with data of different sensitivity classes. Some sensitive data ensures confidentiality and integrity of user data within a single session, *e.g.*, a session key, a pre-master secret in the RSA key exchange, a private DH component in the DH key exchange, *&c*. The other sensitive data is responsible for confidentiality and integrity of many sessions, *e.g.*, a server private key, a client private key and passwords that are reused to access multiple servers. The integrity of the latter sensitive data is more important. Consider deviation from our threat model and assume that privileged compartments may be compromised. If an attacker obtains a session key, he can disclose sensitive data withing a single session; however, disclosing a server's private key allows the attacker to impersonate the server and obtain sensitive data of many sessions.

We require a privilege-separated application to mange sensitive data of these two classes in different compartments to mitigate consequences of a compromise of a privileged compartment operating sensitive data of a single session. However, operations performed on sensitive data of the same class should be manged by a single privileged compartment thus reducing a number of required compartments and improving the application's performance.

Principle 10: A privilege-separated application should manage a session with two separate privileged compartments—one to operate with data related to secrecy and integrity of the current session, and one to manage data that preserves secrecy and integrity of many sessions.

4.4 Hardened Protocol Implementations

We demonstrate the applicability of our partitioning principles by privilege-separating implementation of the SSH protocol in the OpenSSH client and server and the SSL/TLS protocol in the OpenSSL library.



Shaded ovals denote privileged compartments. Unshaded ovals denote unprivileged compartments. The last line in each oval denotes the SELinux policy enforced. Dashed lines denote process communication via UNIX sockets.

Figure 4.19: Partitioning of the Privilege-Separated OpenSSH Client.

The OpenSSL library is particularly interesting because we produced a drop-in replacement. In order to secure an application that uses the OpenSSL library, a programmer must link our privilege-separated implementation instead the baseline implementation without modifying the application.

Recent privilege separation and DIFC work focuses on server applications, as they accept connections and can thus be attacked at will. Often, client-side applications get less attention from research community, but the rise of web browser exploits demonstrates that client code is equally at risk. We start by partitioning the OpenSSH client which is a monolithic application up-to-date.

4.4.1 Hardened OpenSSH Client

The OpenSSH client executes within a single process with full privilege. Therefore, an attacker can set up a public service and provide access to it via SSH. Then, by exploiting vulnerabilities in the SSH client implementation, the attacker can collect user passwords and private keys used to authenticate users to other SSH servers. Users often use the same passwords and private keys to authenticate to different SSH server. Therefore, the attacker may disclose user sensitive data from other servers using the collected passwords and private keys. In addition, the SKD attack is equally valid on both sides, server and client, protection against it is equally needed on the two sides.

The OpenSSH client runs under the invoking user's user and group IDs. Because enforcing setuid and chroot jails requires root privilege, these defenses cannot be applied here. Instead, we limit the privilege of trusted and untrusted compartments of the OpenSSH client with SELinux policies [98], and the SELinux type enforcement mechanism in particular, Section 2.2. SELinux policies allow us to restrict untrusted processes from issuing unwanted system calls such as ptrace, open, connect, *&c.* Our prototype supports only password and public key authentication mechanisms, and does not yet implement advanced SSH functionality (tunneling, X11 forwarding, or support for authentication agents).

Our hardened OpenSSH client starts in the *ssh_t* domain, defined as a standard policy in the SELinux package for the original monolithic SSH client. This policy provides the union of all privileges required by all code in the SSH client; *i.e.*, an application in the *ssh_t* domain may open SSH configuration files, access files in the `/tmp` directory, connect to a server using a network socket, *&c*. We use this domain to initialize the client application and connect to the requested SSH server. At this point, the client has not yet processed any data from the server. Before exchanging any SSH protocol messages, the client creates two new processes (compartments): a privileged session monitor that performs privileged operations on

Session monitor

```
1 client_DH_private_key = generate_DH_private_key()
2 client_DH_public_key = compute_DH_public_key(client_DH_private_key)
3 session_key = compute_session_key(client_DH_private_key,
  server_DH_public_key)
4 session_ID^i = compute_session_id(session_key, server_version,
  client_version, server_kexinit, client_kexinit, ...)
5 symmetric_keys = derive_symmetric_keys(session_ID^i, session_key)
6 server_public_key^i = verify_public_key(server_public_key, username,
  known_hosts)
7 verify_signature(session_ID^i, server_public_key^i, signature)
```

Private key monitor

```
1 signature = sign(client_private_key, session_ID^i, user_name,
  service, auth_mode, ...)
```

Sensitive data appear in bold, and are accessible only by the corresponding monitor. Untrusted parameters provided by unprivileged compartments are not in bold. $\mathbf{x}^i$ denotes that sensitive data **x** is exported to an unprivileged compartment read-only.

Figure 4.20: Privileged Operations Implemented by the Session Monitor and Private Key Monitor in the OpenSSH Client.

sensitive data that can compromise only a single SSH session, and a private key monitor that performs authentication operations with the client's private keys. This ensemble of three compartments appears in Figure 4.19. The use of two distinct monitors is motivated by Principle 10.

The session monitor runs in the *ssh_monitor_t* domain, a domain we have defined that confines the process to access only the `known_hosts` file which contains public keys for approved servers, and to read/write UNIX sockets for communicating with the private key monitor and an unprivileged processes running untrusted code in session key negotiation, pre-authentication, and post-authentication stages. The session monitor cannot create or access any files apart from `known_hosts`, nor may it create new sockets. The private key monitor runs in the *ssh_pkey_t* domain, a domain we have defined with a similarly tight policy, allowing it only to read the user's private key(s), with no access to other files, nor privilege to create any sockets. The private key monitor shares a UNIX socket with the session monitor and only processes requests from the latter. After creating these two monitor processes, the original SSH client process drops privilege to the *ssh_nobody_t* domain. Untrusted code of session key negotiation and pre-authentication stages runs in this unprivileged process and domain during the rest of the SSH client's execution. The *ssh_nobody_t* domain allows the unprivileged process to communicate with the session monitor and remote server via previously opened sockets, but prevents it from opening any new ones, and it further denies all access to the file system. The OpenSSH client's SELinux policies allow both monitors and unprivileged processes access to the terminal device via standard input, output, and error file descriptors.

The session monitor compartment isolates all sensitive data that can be used to compromise the current remote login session, and performs all privileged operations with these data, enumerated in Figure 4.20, that are essential for key exchange and prevention of a private-key oracle. When a privileged operation takes non-sensitive data as input, the non-sensitive input is supplied by the unprivileged compartment. Symmetric keys are the keys derived from the session key for the MAC and encryp-

tion/decryption. The session monitor enforces the order in which an untrusted compartment may invoke its privileged operations.

The private key monitor isolates the client's private key and performs a signing operation with the key as shown in Figure 4.20. Only the session monitor may invoke this signing operation in the private key monitor (over a UNIX-domain socket), and it provides the session ID to be signed as an argument. We give a more detailed explanation of the private key signing operation below.

Session Key Negotiation Stage

We now consider the first stage of the hardened OpenSSH client, the session key negotiation stage, designed to thwart SKD attacks described in Section 4.2.1. Although we described an SKD attack against the OpenSSH server, this attack succeeds in the same way when directed against the client. In the session key negotiation stage, an unprivileged compartment (with the help of the session monitor) performs Diffie-Hellman key exchange to negotiate a session key and authenticate the server. In accordance with Principle 1, we restrict the session key negotiation stage to run in an unprivileged compartment that cannot access sensitive data—not the DH private key, nor the session key, nor the symmetric keys as shown in Figure 4.20. Keeping the session key secret (and thus thwarting an SKD attack) requires in turn keeping this data secret. The session monitor implements Principle 4 and scrubs the private DH component after computing the session key.

We must also prevent a signature verification oracle described in Section 4.3.2 that may present at this point in the handshake. Suppose the attacker wants to impersonate a server to the client, and can trick the client into connecting to a server he controls, instead of to the bona fide server intended by the client. Suppose further that the attacker exploits the client. To authenticate the server, the client must verify the server's public key against the list of trusted public keys in the `known_hosts` file, and then validate the server's signature on the session ID. Once the attacker exploits the client, if the exploited compartment of the client implementation allows invocation of signature verification operation with the session ID or server's public key provided by this compartment then the attacker may be able to force signature verification to succeed, and thus spoof the bona fide server to the client. To see why, note the arguments to the signature verification routine *verify_signature()* in the session monitor in Figure 4.20. If the attacker controls the values of the signature argument and *either* the session ID argument *or* the server public key argument, he can provide inputs that will cause the signature to verify. That is, he can either sign a benign session ID with his *own* private key and supply his *own* corresponding public key, or supply a bogus session ID signed by the bona fide server (readily obtained from the attacker's own connection to the bona fide server), along with the bona fide server's true public key.

To prevent this verification oracle, we must not allow an unprivileged compartment (at risk for exploit) to provide either server public key or session ID to verify_signature(). We thus perform signature verification in the session monitor, and isolate session ID and server public key within the monitor. Actuality, the untrusted compartment provides server public key to the session monitor, but the session monitor validates it against the content of the `known_hosts` file before verifying the signature. Note that session ID is entangled with trusted random bits generated by the client every new session,

originating from the client's DH private key via compute_session_key() and compute_session_ID(). This construction, specified by the OpenSSH protocol, implicitly applies Principle 7, which further prevents an attacker from forcing session ID to match that from a past eavesdropped session.

We now turn our attention to the next step taken by the client. In the OpenSSH protocol, session key negotiation and server authentication, which establishes the user privilege barrier, are intertwined. Therefore, our partitioning of OpenSSH needs no distinct pre-authenticated stage, and the SKN stage proceeds immediately to the post-authenticated stage.

Post-authenticated Stage

After computing symmetric keys and authenticating the server, the client kills the untrusted compartment from the SKN stage and creates a new untrusted compartment, also confined to the *ssh_nobody_t* domain, to execute operations in the post-authenticated stage as required by Principle 2. This new compartment is granted access to the session's symmetric keys so that it can perform encryption and decryption operations. It may invoke privileged operations in the session monitor, and the session monitor can invoke privileged operations on the client's private keys by the private key monitor. To do so, the private key monitor executes with the privilege to read private key files.

In the post-authenticated stage, the server authenticates the client. Our prototype supports password and public key authentication. Password authentication does not require any further partitioning of the client to protect against a malicious server, as the SSH protocol requires that the client sends the password to the server. However, we can apply fine-grained privilege separation to deny the server access to the client's private key(s). There is no need for the untrusted compartment to have direct access to the keys, and if it does, a malicious server that the user logs in may exploit the client and obtain its private keys, and thus obtain sensitive information from other SSH servers where the user authenticates himself using the same private keys. Therefore, we isolate the client private keys from the post-authentication stage's untrusted compartment by placing them in a privileged private key monitor.

To prevent a private key signing oracle, described in Section 4.3.2, we do not allow the untrusted compartment to directly sign data of its own choice using the private key. The untrusted compartment passes untrusted input (user name, service name, authentication mode, *&c.*) via the session key monitor. Note that we rely on session key monitor to supply the trusted session ID computed earlier in the key exchange protocol to the private key monitor as shown in Figure 4.20. Recall that the session ID has been entangled with trusted random bits generated by the client for the current session in compliance with Principle 7. Thus, the signature produced by the private key monitor will not be valid in any session but the current one, and a private key oracle has been disseminated.

To support session key rekeying, the unprivileged process of the post-authenticated stage is permitted to invoke privileged rekeying operations implemented by the session monitor.

4.4.2 Hardened OpenSSH Server

The baseline privilege-separated OpenSSH server uses `chroot` and `setuid` system calls to restrict unprivileged compartments. As discussed in Section 2.1, the chroot jail restricts file system access, and the setuid jail prevents a process from tampering with user processes. However, more fine-grained

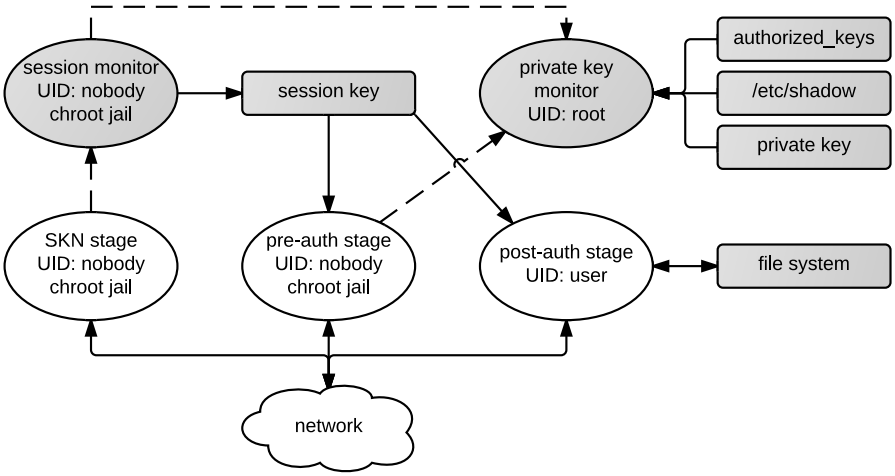


Figure 4.21: Partitioning of the Hardened OpenSSH Server.

control over a process's privilege is required to restrict dangerous system calls. For example, `ptrace` system call can be used by an unprivileged process to control and read address space of another unprivileged process that handles a session of a legitimate user. Thus, an attacker may establish a connection with a server and compromise the server's unprivileged process, Figure 2.1. Then, by ptracing other unprivileged processes that handle concurrent sessions of legitimate users, the attacker may discloses passwords and computed session keys of the users. Apart from this attack, the baseline implementation of the OpenSSH server is vulnerable to the session key disclosure attack and oracle attacks, Section 4.2. In particular, the baseline server exposes the signing oracle, signature verification oracle, and deterministic session key oracle.

Figure 4.21 shows our hardened OpenSSH server that implements our protocol partitioning rules. We split an unprivileged process in the baseline OpenSSH server, Figure 2.1, into two compartments one to act as an unprivileged compartment in the session key negotiation stage, and another to form the pre-authenticated stage, Principle 2. The session key negotiation stage has no access to a session key or any sensitive information that allows deriving the session key as required by Principle 1.

The monitor process in the baseline OpenSSH server is split into two privileged compartments: a per-connection *session monitor* that isolates sensitive data crucial for integrity of a current session and exports operations on this data, and a per-connection *private key monitor* that isolates the server's private key responsible for integrity of many SSH sessions and performs operations that require root privilege. Principle 10 motivates this division. Note that the session monitor does not require access to file system and root privilege to perform its tasks, it only isolates sensitive data produced at session key negotiation; therefore, it is confined with the same restrictions as the unprivileged processes (setuid jail, chroot jail, SELinux policy). In addition, the session monitor scrubs sensitive session key material, the server's private DH component, after computing the session key as required by Principle 4.

In our hardened implementations of the OpenSSH server and client, we employ SELinux policies [98] to tighten system call privilege of compartments. Thus, unprivileged processes of the session

Session monitor

```
1 server_DH_private_key = generate_DH_private_key()
2 server_DH_public_key = compute_DH_public_key(server_DH_private_key)
3 session_key = compute_session_key(server_DH_private_key,
  client_DH_public_key)
4 session_ID^i = compute_session_ID(session_key, server_version,
  client_version, server_kexinit, client_kexinit, ...)
5 symmetric_keys = derive_symmetric_keys(session_ID^i, session_key)
```

Private key monitor

```
1 signature = sign(server_private_key, session_ID^i, user_name,
  service, auth_mode, ...)
2 client_public_key^i = verify_public_key(client_public_key, username,
  user_authorized_keys)
3 verify_signature(session_ID^i, client_public_key^i, signature)
4 verify_password(username, password)
```

Sensitive data appear in bold, and are accessible only by the corresponding monitor. Untrusted parameters provided by unprivileged compartments are not in bold. $\mathbf{x}^i$ denotes that sensitive data **x** is exported to an unprivileged compartment read-only.

Figure 4.22: Privileged Operations Implemented by the Session Monitor and Private Key Monitor in the OpenSSH Server.

key negotiation and pre-authentication stages as well as the session monitor are chrooted, setuid-jailed, and confined with a restrictive SELinux policy that only allows system calls implied by Figure 4.21.

Session Key Negotiation Stage

The session monitor implements privileged operations required to safely establish a session key as well as to prevent the signing and the signature verification oracles exposed by the baseline implementation. Figure 4.22 lists the operations implemented by the session monitor. *server_DH_private_key* is a private DH component which can be used to derive the session key. This component is random and generated for every new session; thus, it plays a role of a server random nonce. The session monitor isolates the private DH component, Principle 1. The session key is produced from the server's private DH component and a client's public DH component (nonsensitive data supplied by the session key negotiation compartment). After computing a session key, the session monitor scrubs private DH component from its memory to comply with Principle 4.

In the session key negotiation stage, a server must authenticate itself to the client by signing a session ID. We prevent a signing oracle by ensuring that the data signed by the server incorporates a server random nonce, Principle 7, which is the private DH component. Operations 1, 3, and 4 in Figure 4.22 performed by the privileged session monitor guarantee that the session ID is entangled with per-session randomness, and the session monitor supplies a trusted copy of the session ID to the private key monitor for signing, Operation 1 in Figure 4.22. Thus, an attacker cannot exploit the signing operation as whatever she signs contains a fresh private DH component and cannot be used within any other session but her own. *derive_symmetric_keys* produces two pairs of symmetric keys: one for outbound encryption and MAC, and the other for inbound decryption and MAC.

After computing a session key and server authentication, we enable the session key barrier, Principle 2, by enforcing MAC on the channel and killing the unprivileged compartment of the session key

negotiation stage, Figure 4.21.

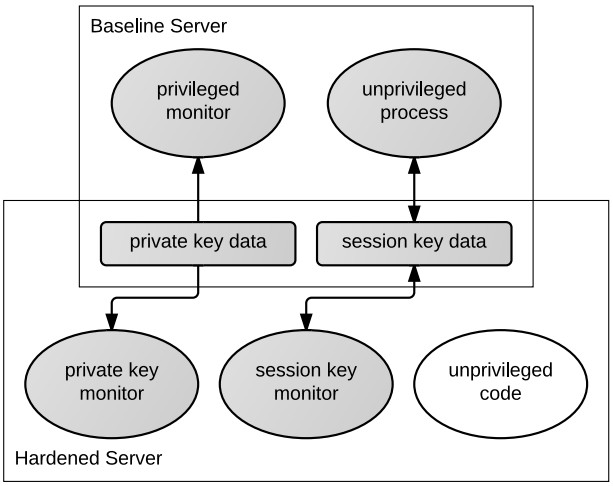
Pre-authentication and Post-authentication Stages

The pre-authentication stage performs user authentication; depending on an authentication method, it may involve privileged operations such as password checking, verification of a user's public key, and verification of a signature on a session ID, Figure 4.22. The baseline OpenSSH server isolates these operations within its monitor process running as root because the operations require access to sensitive data or root privilege. Our hardened implementation inherits this model; however, we eliminate the signature verification oracle exposed by the baseline OpenSSH server. The signature verification operation *verify_signature*, Figure 4.22, is used to verify a client's signature when the client authenticates to the server. There are three arguments to this operation where the *signature* is supplied by an untrusted source. If an attacker controls one of the other arguments, *session ID* or *client public key*, then she can bypass the signature verification procedure. Controlling the client private key allows an attacker to supply her own public key and a signature produced with her key over a benign session ID. If the session ID is under attackers control, which is the case in the baseline OpenSSH server, the attacker can replay a session ID and the corresponding signature from a previous user session. However, this pair of data is transfered within an encrypted channel, and it is unlikely that the user will wish to grant it to the attacker. Because the baseline OpenSSH server does not provide protection against the SKD attack, one can use this attack to penetrate the encrypted channel and disclose the session ID and the corresponding signature. Our implementation of the OpenSSH server thwarts SKD attacks, thus it is not possible to obtain required pair of authentication data. Moreover, it prevents an SKD attacker from disclosing user passwords if the password authentication is used where the baseline OpenSSH lacks this guarantee.

Despite the SKD defenses, we also eliminate the signature verification oracle. First, an attacker does not control the client's public key. It is initially supplied by an unprivileged compartment, but after public key validation against the user's authorized keys file, *verify_client_public_key* in Figure 4.22, the monitor keeps its own copy of the public key which is used in signature verification. The baseline OpenSSH server implements this mechanism. Unlike the baseline OpenSSH server, we apply our Principle 7 and make sure that the data used in signature verification is entangled with the server per-session randomness. The sequence of privileged operations in the session monitor ensures the presence of the server's private DH component (per-session randomness) in the session ID. The private DH component prevents both the signing oracle and the signature verification oracle. The session monitor provides trusted, entangled session ID for the signature verification operation. Thus, an attacker has no means to replay signature as the session ID is generated afresh for every session.

Discussion: Privileged Code Base

Figure 4.23 presents comparison between Privileged Code Bases (PCB) of the baseline OpenSSH server and our hardened implementation of the OpenSSH server. The latter extends the partitioning of the baseline implementation with an additional unprivileged compartment for the session key negotiation stage motivated by Principle 1 and Principle 2, and with the privileged session key monitor, Principle 10. The session monitor implements operations that provide oracle and SKD defenses, Principle 2 and Princi-



Privileged code is shaded; unprivileged code is unshaded.

Figure 4.23: Relationship Between Privileged and Unprivileged Code in the Baseline and Hardened OpenSSH Servers.

ple 7. In the baseline OpenSSH server, these operations are performed in the unprivileged process.

At first glans, one might notice that we increased the PCB with our session monitor, but this assumption is flawed. Rather the *session key data*, which integrity and confidentiality are crucial for the SKD and oracle defenses, was mislabeled as nonsensitive and allowed to be manipulated by unprivileged code in the baseline OpenSSH server. Thus, the unprivileged process of the baseline implementation becomes privileged code with respect to the session key data, and we relabel it as privileged, shaded. It is clear that our hardened implementation reduces the PCB by correctly identifying the sensitive data and factoring out the code that truly requires no access to the sensitive data.

4.4.3 Hardened OpenSSL Library

Toward demonstrating the generality of the partitioning principles presented in Section 4.3, we have also applied them to the SSL 3.0 and TLS 1.0 cryptographic protocol implementations in the OpenSSL library. As partitioning in accordance with these principles requires a fair amount of programmer effort, we found the OpenSSL library a particularly attractive target; hardening the library allows amortizing one partitioning effort over a broad range of security-conscious applications. The resulting hardened OpenSSL library is a drop-in replacement that renders any SSL/TLS application linked against it immune to SKD and oracle attacks. We note, however, that changing the library alone cannot ensure that the application *atop* the library itself handles sensitive data securely. For example, the Apache web server reuses worker processes across requests submitted by different users. If an attacker exploits a worker process, he may be able to obtain sensitive data belonging to the next user whose request is handled by that process. Moreover, the server does not isolate its private key from worker processes; thus, an attacker compromising a worker process may disclose the server's private key and impersonate the server.

We finely partition both the client and server sides of OpenSSL. Our implementation supports RSA, ephemeral RSA, Diffie-Hellman, and ephemeral Diffie-Hellman key exchanges, client and server authentication, and session caching. The OpenSSL partitioning is in fact similar in structure to that of SSH, as

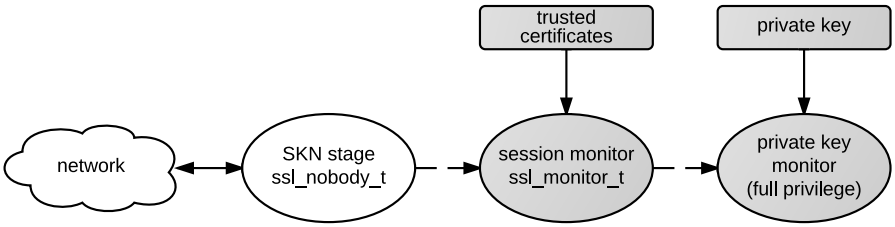


Figure 4.24: Partitioning of the Hardened OpenSSL Client and Server.

these protocols protect against similar threat models. When an application invokes SSL_accept routine (on the server) or SSL_connect routine (on the client), we instantiate private key monitor, session key monitor, and unprivileged SKN compartments, Figure 4.24. Our implementation scrubs the server's private key from the session key monitor process and the unprivileged SKN compartment before reading any input from the network. We further apply simple SELinux policies to limit the privilege of the untrusted SKN compartment and the session monitor in applications that do not run as `root` and unable to employ the chroot and setuid jails. The SELinux policies only allow system calls implied by Figure 4.24. If an application is confined with its own SELinux policy, we provide a simple macro that requires one line change in the application's SELinux policy to enforce our restrictive policies on OpenSSL compartments. The untrusted SKN compartment runs in *ssl_nobody_t* domain which allows it to read and write a TCP socket provided by the application to communicate with the peer and read and write a UNIX socket to call into the session monitor. Only the session monitor, confined with *ssl_monitor_t* can read and write a UNIX socket to call into the private key monitor, which in turn is confined with the application's default policy. Thus, we significantly limit the harm possible from compromise of the SKN compartment.

As SSL/TLS supports only public key authentication, its partitioning omits the pre-authentication stage. When the SKN stage completes, the unprivileged compartment and session monitor are terminated, and execution continues in the application's fully privileged compartment. The private key monitor preserves the privileges of the application before entering the SSL_accept and SSL_connect library calls. Therefore, this compartment continues execution of the application's code and can use the symmetric key computed during the SSL handshake to perform MAC and encryption/decryption operations on the established SSL/TLS session. By allowing the post-authentication stage to run with full privilege, we compromise security in favor of applicability. An attacker exploiting the post-authentication stage may disclose the server/client's private key and use it for impersonation. However, this design decision allows drop-in replacement of the hardened OpenSSL library in applications. We encourage developers to tighten privilege of the post-authentication stage and restrict access to private keys in applications that use the OpenSSL library.

Both SSH and SSL protocols support Diffie-Hellman key exchange, and we apply same defense mechanisms in the OpenSSL implementation as in the OpenSSH one. Unlike OpenSSH, the OpenSSL library implements RSA key exchange protocol which allows the SKD attack as well as the private key decryption oracle and deterministic session key oracle, Section 4.3.2.

Figure 4.25 shows privileged operations that the server-side session monitor of the OpenSSL library

Session monitor

```
1 server_random^i = generate_server_random()
2 session_key = compute_session_key(client_random, server_random^i,
  encrypted_pre_master_secret)
3 bool = verify_client_finished_message(handshake_messages,
  encrypted_client_finished_message, session_key)
4 encrypted_server_finished_message = compute_server_finished_message(
  handshake_messages, session_key)
  ...
```

Private key monitor

```
1 pre_master_secret = decrypt(server_private_key,
  encrypted_pre_master_secret)
  ...
```

Sensitive data appear in bold, and are accessible only by the private key monitor. Untrusted parameters provided by unprivileged compartments are not in bold. $\mathbf{x}^i$ denotes that sensitive data **x** is exported to an unprivileged compartment read-only.

Figure 4.25: Privileged Operations of the Server-Side OpenSSL Library.

implements to thwart the SKD attack and oracles when performing the RSA key exchange. The session monitor generates a server random nonce, Operation 1, which is later used to compute a session key, Operation 2. The trusted random nonce restricts the deterministic session key oracle and prevents an attacker from making the session monitor to compute a session key that matches the one in a past SSL session of a benign user. The private key monitor defends against the private key decryption oracle by decrypting pre-master secrets that are supplied by the session monitor rather than by the untrusted compartment, Operation 1 in Figure 4.25, and the session monitor never returns the result of the decryption operation to the untrusted compartment.

The OpenSSL library alternates encrypted and MACed final finished messages and clear-text change cipher messages which violates Principle 6. Therefore, the library must enforce the session key barrier after processing the last clear-text message to comply with Principle 3, and thus it must decrypt an incoming final finished message and encrypt an outgoing final finished message within the unprivileged SKN compartment. A final finished message is a hash of a session key and all previously transfered message between a client and server. To avoid the session key encryption and decryption oracles, we employ Principle 8 to the corresponding operations performed by the session monitor.

In the server-side OpenSSL library, Operation 3 and Operation 4 of the session monitor (Figure 4.25) restrict the decryption and encryption oracles by obfuscating untrusted input with hashing, Principle 8. Operation 3 decrypts the final finished message and compares it with a finished message computed locally from supplied handshake messages and the session key. The unprivileged compartment receives a boolean result of the comparison but not the decrypted clear-text message; thus, this structure cannot be used as a decryption oracle. Operation 3 further restricts a dictionary attack where an attacker supplies a user's message encrypted with a session key instead of an encrypted client finished message and attempts to guess the content of the message by supplying chosen clear-text message in a place of the handshake messages. Operation 4 in Figure 4.25 produces the encrypted server final finished message and expose no encryption oracles. Before encryption, the monitor hashes session key and handshake messages supplied by the untrusted compartment. Section 4.3.2 provides detailed description

of the session key encryption and decryption oracles and the above defenses.

Our partitioned implementation of the OpenSSL library supports session caching, crucial to high-performance applications such as SSL web servers. TLS session caching [121] requires only a client to store a session state. A TLS server generates a single secret symmetric key and uses it to encrypt the session states of the clients. Each client receives its encrypted session state from the server and stores it locally. Whenever a client wants to resume the session, it passes the encrypted state to the server, and then the server decrypts it using its secret symmetric key. In our partitioned version of OpenSSL, this secret caching key is known only to the private key compartment and scrubbed from other compartments. In SSL 3.0 protocol, a server must store session states locally. OpenSSL library does not provide a mechanism for inter-process cache and requires an application to implement it. Based on our experience with the Apache Web server, there are two possibility of organizing shared cache, shared memory and a shared file. Currently, we support only file storage and restrict access to it for the SKN compartment and session monitor using SELinux policy. We have no mechanism to deny these processes access to a shared-memory cache, as the application layer provides the cache, and it is thus not under the control of the OpenSSL library.

4.5 Evaluation

4.5.1 Security Analysis

Our partitioning principles are agnostic to privilege reduction mechanisms. We chose the SELinux policy mechanism to constrain compartments of the privilege-separated SSH and SSL/TLS protocol implementations. We made this choice over Wedge to ease adoption of the privilege-separated protocol implementations. Wedge requires kernel modifications where SELinux is pre-installed on many Linux distributions and also provided by the package management systems.

We extended standard SELinux policies of the OpenSSH client and server with the additional ssh_nobody_t, ssh_monitor_t, and ssh_pkey_t domains to confine unprivileged compartments, the session monitor, and the private key monitor, respectively. In the case of the OpenSSH server, Figure 4.21, we also use the server's default security mechanisms, the setuid and chroot jails, to constrain untrusted compartments and the session monitor (in addition to the SELinux policy). As for applications that employ the OpenSSL library, we introduced a one line change in their SELinux policies to add a macro which creates the ssl_nobody_t and ssl_monitor_t domains within an application's SELinux policy. These domains are required to confine an unprivileged compartment of the SKN stage and the session monitor, Figure 4.24.

The ssh_nobody_t and ssl_nobody_t domains constrain unprivileged compartments in the SSH and SSL/TLS protocol implementations. They enforce the most restrictive policy and only allow to read and write a network socket connected to a remote peer and communicate with the session monitor. Access to the file system and dangerous system calls, *e.g.*, ptrace, as well as opening new sockets are denied. When we fork an unprivileged compartment and the session monitor, sensitive information such as the private key is scrubbed from these compartments' memory. Thus, an attacker compromising an

Bug Location	OpenSSH	OpenSSL
Unprivileged compartment	3	4
Session key monitor	1	1
Private key monitor	1	1
Total number of bugs	5	6

Table 4.2: Number of Buffer Overflow Vulnerabilities in Compartments of the Hardened OpenSSH Server and OpenSSL Library.

unprivileged compartment gains no access to sensitive information, and there is no way she can obtain such information from the system as restricted by the SELinux policies. The SELinux policy mechanism and our partitioning principles thwart the session key disclosure and oracle attacks.

4.5.2 Robustness of Hardened Protocol Implementations against Previously-Reported Vulnerabilities

This section evaluates how effectively our partitioned cryptographic protocol implementations would have defended against exploits targeting *real* buffer overflow vulnerabilities that were reported in the past. We used applications' change logs and the CVE database [6, 33] to search for past buffer overflow bugs. For each bug, we identified its location in the application's source code using the bug's description or a proof-of-concept exploit. We mapped the bugs' locations to compartments in privilege-separated applications and assessed if an attacker could disclose sensitive information when exploiting these compartments.

OpenSSH and the OpenSSL library are open-source projects, and thus, these applications are audited by security experts and independent developers. As a result, there were a number of buffer overflow vulnerabilities found in these applications; however, the developers published descriptions of only a small fraction of them. We only consider buffer overflows identified in these published descriptions in the effectiveness evaluation.

In the OpenSSH server and client, the published vulnerability reports describe five buffer overflow bugs [17–21]. Three of them are in the unprivileged compartments of our highly-partitioned implementations of the OpenSSH server and client, one bug is in the session monitor, and another is in the private key monitor. We similarly mapped the locations of six publicly reported bugs in the OpenSSL library [15, 24–29]. Four bugs land in the unprivileged compartment, one in the session monitor, and another one in the private key monitor. Note, the latter bug is in the code that handles session caching. By further splitting the private key monitor into two compartments, one to deal with the private key and another to handle session caching, we can secure the private key from the vulnerability in the session caching code, but this does not protect session keys stored in the cache.‘

Table 4.2 summarizes our findings. Our hardened implementations of cryptographic protocols sustain compromise of unprivileged compartments with no threat to the confidentiality or integrity of user and protocol sensitive data, *e.g.*, user data, session and private keys. Compromise of the session monitor allows an attacker to launch SKD and oracle attacks, and compromise of the private key monitor may lead to disclosure of the private key. According to Table 4.2, our hardened implementations of

OpenSSH and OpenSSL protect sensitive data against 60-67% of the analyzed buffer overflow vulnerabilities (assuming they are exploitable). In contrast, the vulnerabilities in the baseline implementation of the privilege-separated OpenSSH server could allow successful SKD and oracle attacks. Moreover, the vulnerabilities in the baseline OpenSSH client and OpenSSL library, both of which are monolithic implementations, could result in disclosure of a private key.

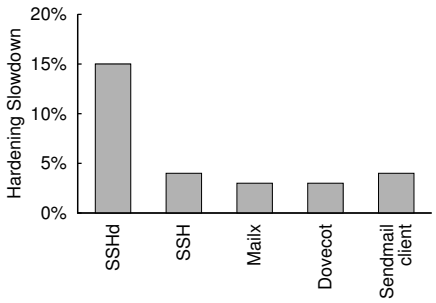
4.5.3 Performance Evaluation

In this section, we present performance evaluation of applications that we have privilege-separated according to our partitioning principles and protected from the session key disclosure and oracle attacks.

Apart from the OpenSSH client and server implementations, described above, we secured a number of client-side and server-side applications with our drop-in replacement of the OpenSSL library. We ported SSL tunneling tool stunnel (server and client parts) and converted the Mutt and Mailx mail agents to enable them to connect to IMAP and POP3 servers via our privilege-separated implementation of SSL/TLS protocol. Dovecot is a popular IMAP and POP3 server which we also converted to use our library for IMAP and POP3 connections with mail agents. Sendmail is a mail transfer agent that preinstalled in many Linux distributions, it may employ the SSL/TLS protocol to transfer mail within SMTP protocol. We linked sendmail against our library, thus provided resistance to SKD and oracle attacks in sendmail's client and server parts. Finally, we tuned the well known Apache web server to use our privilege-separated library and host web content over HTTPS. We experimented with two versions of the Apache Web server, 1.3.19 and 2.2.14.

Converting most of these applications was straightforward; it merely required replacing the OpenSSL library and making a one-line change to the application's SELinux policy, without any application code modifications. However, there are a few caveats associated with our partitioned OpenSSL library. An application using the library can substitute the I/O layer with its own. Thus, some applications (such as Apache 2) implement I/O buffers while performing SSL handshake operations. Unfortunately, we cannot handle buffering during the SSL handshake because we transfer state between the processes. Therefore, we disable any application's I/O layer for the time we perform the SSL/TLS handshake and enable it as we finish with the handshake. We argue that this modification does not effect the performance of application dramatically because there is not much to buffer during the SSL/TLS handshake, and we re-enable the buffering for the session. We also disable non-blocking I/O during the handshake phase. This is because the non-blocking I/O is handled by the application layer but not by the library. Each time the I/O fails, the SSL_accept and SSL_connect library calls return to the application which handles the I/O and then the application re-enters the library calls. Remember, we fork two additional processes each SSL_accept or SSL_connect library call; therefore, it would be quite expensive to provide non-blocking I/O in the SSL/TLS handshake. However, the application can fully utilize the non-blocking I/O as soon as the session is established. Despite these caveats, applications that substitute OpenSSL's I/O layer and make use of non-blocking I/O require no modifications in order to adopt our hardened implementation of the OpenSSL library.

Our partitioning principles require privilege separation of applications employing cryptographic



The execution slowdowns are relative to the baseline implementations.

Application	Baseline	Hardened
OpenSSH Server	0.114	0.131 (15%)
OpenSSH Client	0.114	0.118 (4%)
Mailx	0.087	0.090 (3%)
Dovecot	0.087	0.090 (3%)
Sendmail Client	0.127	0.132 (4%)

Latency of applications' operations is in seconds. The slowdowns relative to the baseline applications are in parentheses. Evaluated applications are OpenSSH 5.2p1 client and server, Mailx 12.4, Dovecot 1.2.10, and Sendmail client 8.14.4 with the baseline and hardened OpenSSL 0.8.9k libraries.

Figure 4.26: The Latency Cost of SKD and Oracle Defenses in Client and Server Applications.

protocols, and the partitioning results in additional compartments. In conventional operating systems, compartments are implemented using processes. However, process isolation incurs performance overhead associated with process creation and context switches. In order to evaluate the impact of the privilege separation, we compare the performance of the baseline client-side and server-side application with that of the implementations hardened in accordance with the principles we have propounded. We chose end-to-end metric: latency (important to users) for client-side applications and throughput (important to server operators) for server-side applications.

Figure 4.26 compares operation latencies for a range of applications. Whenever, we evaluate a client-side or a server-side application, its counterpart is a baseline application. All experiments are done on Dell desktop with 1.86 GHz Intel Core 2 6300 CPU and 1 GB RAM running Linux 2.6.30 via loopback interface to mitigate the delay introduced by the network. Our partitioning affects only protocol implementation and introduces overhead in session establishing; in order to expose this overhead, we do minimal work within the session and exit as soon as possible. In the OpenSSH evaluation, we measure the time required for a client to login to the server using public key authentication and run exit command. For the Mailx email client and Dovecot IMAP server, we measure the time required for the client to check if there is new mail available at the server using SSL/TLS. For the sendmail client, we measure the time required to send a one-line email to a sendmail server over SSL/TLS. For these applications, the OpenSSH server incurs the largest relative slowdown of 15%; however, the latency a user perceives does not increase significantly, 0.017 seconds, between the baseline and hardened cryptographic protocol implementations.

Figure 4.27 demonstrates performance evaluation of the server applications tested on Sun X4100

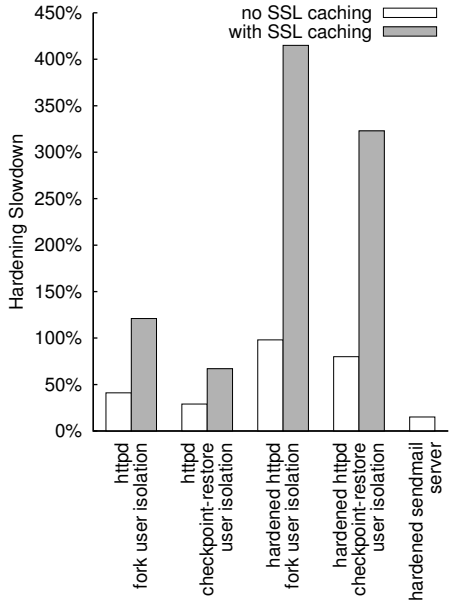
server with 2.2 GHz AMD Opteron 248 CPU and 2 GB RAM under Linux 2.6.32. We hardened implementations of the SSL-enabled sendmail server and the SSL-enabled Apache web server. For the sendmail server, we present the maximum load that the server can sustain in emails served per second over SSL/TLS. We stress the server from multiple clients sending small emails. As shown in Figure 4.27, our oracle and SKD defenses do not introduce significant performance overhead for the sendmail server, 15% loss in throughput.

To determine the maximum load the Apache web server can sustain, we stress it from multiple clients requesting a small static page over SSL/TLS. Session caching plays a significant role in the server's performance. Establishing a new SSL/TLS session is a computationally expensive operation because of heavy crypto operations. Where else resuming a cached session requires only restoring a session key. We present results for two extreme operation cases: the first case is when the session caching is disabled, each HTTP request results in a new SSL/TLS session, and the second case is when all sessions are cached. We configure HTTPS clients to use RSA key exchange when establishing an SSL/TLS session because this protocol is less computationally intensive for the server than ephemeral Diffie-Hellman key exchange and thus better exposes the overhead of hardening.

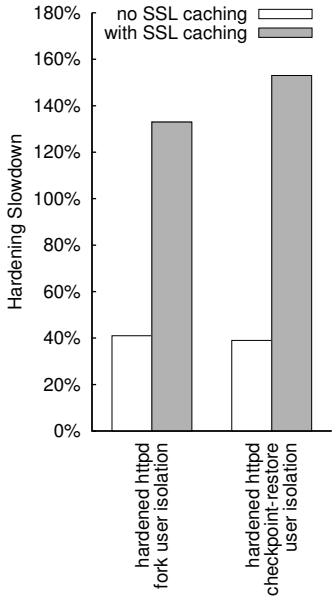
Apart from adding defenses against SKD and oracle attacks, we modified the baseline Apache web server to tighten its security to protect sensitive data *after* the SSL handshake completes. The baseline implementation uses a pool of worker processes where each process scheduled to serve a user connection. When the connection is closed, the process is rescheduled to serve a different user. Thus, worker processes are reused among different users in the baseline Apache web server. We identified a security threat: if an attacker compromises a worker process, she can leak sensitive data of the next user that the process is scheduled to serve. The oracle and SKD defenses are no use in this case as our defenses make sure that the sensitive data are handled correctly within the cryptographic protocol but not by the application atop of the protocol which is also responsible for handling sensitive data.

In order to mitigate this threat, we introduced inter-user isolation by forking a distinct worker process for each user connection, and when the connection is over, the process is destroyed rather than reused. We further scrub the server's private key from the worker's memory immediately after the trusted monitor uses it during the SSL handshake. Thus, after accepting a user's connection, an untrusted worker process no longer holds the server's private key, and no vulnerability in the Apache application code will allow the user to disclose the private key. The inter-process isolation has a significant performance cost which we show in our evaluation. We compare two implementations of this isolation. The first is a naive one in which Apache kills a worker after it serves one request and forks another to replace it. As the overhead of fork is significant, we compare against an optimized implementation based on *checkpoint-restore*, as proposed by Bittau [65]. In this approach, Apache takes a snapshot of each new worker process's pristine memory image before it serves any requests, and after each request, a trusted monitor process restores the worker's memory image to this pristine state.

Returning to Figure 4.27, let us first consider the workload in which no SSL/TLS sessions are cached, running on the hardened versions of Apache implemented using checkpoint-restore. End-to-



The cost of user isolation in the Apache web server 2.2.14 (httpd), fork and checkpoint-restore implementations, and the cost of further hardening of these servers and the Sendmail server 8.14.4 with our SKD and oracle resistant OpenSSL 0.8.9k library. The cost is expressed in execution slowdowns relative to the unmodified server implementations.



The isolated cost of the hardened OpenSSL library which is the execution slowdowns relative to the Apache server implementations that do incorporate user isolation.

Application	Baseline OpenSSL		Hardened OpenSSL	
	no caching	with caching	no caching	with caching
baseline httpd	303	1700	–	–
httpd fork user isolation	215 (41%)	770 (121%)	153 (98%) [41%]	330 (415%) [133%]
httpd checkpoint-restore user isolation	234 (29%)	1016 (67%)	168 (80%) [39%]	402 (323%) [153%]
sendmail server	53	–	46 (15%)	–

Throughput in requests per second. Evaluated applications are the baseline Sendmail server 8.14.4, baseline Apache web server 2.2.14 (httpd), Apache web server implementing user isolation with fork and checkpoint-restore. We compare throughput of these servers to the implementations secured with our hardened OpenSSL 0.8.9k library. In round parenthesis, the slowdowns of modified and hardened implementations; the slowdowns are relative to the unmodified/baseline server implementations. Percentages in squared parenthesis are the slowdowns of the Apache web servers with hardened OpenSSL library; the slowdowns are relative to Apache servers implementing user isolation but using the baseline OpenSSL library.

Figure 4.27: The Throughput Cost of SKD and Oracle Defenses in Server Applications.

end, the version of Apache providing both inter-user isolation and defenses from oracle and SKD attacks achieves more than half (55%) the throughput of baseline Apache and results in 80% slowdown; however, the baseline implementation provides none of these security benefits. The overhead of these security mechanisms is masked in part by the computational costs of the cryptographic operations required to establish a new SSL/TLS session. We note that this "fully" hardened version of Apache achieves over 70% the throughput of one that provides inter-user isolation with checkpoint-restore but omits oracle and SKD attack defenses—so for this workload using these isolation primitives, oracle and SKD attack defenses incur only moderate slowdown of 39%.

In the workload in which no SSL/TLS sessions are cached, there are no public-key cryptographic operations, so the overheads of inter-user isolation and oracle and SKD attack defenses are more exposed. Focusing on the implementations built on checkpoint-restore, Apache with inter-user isolation (but without oracle/SKD defenses) achieves 60% of the throughput of baseline Apache; this reduction

is the cost of inter-user isolation. Adding oracle and SKD defenses to the inter-user-isolated implementation further reduces throughput by 60% introducing 150% slowdown; that is the incremental cost of oracle and SKD defenses on this challenging workload. End-to-end, this last version of Apache, which incorporates all defenses and inter-user isolation, achieves only about one quarter of the throughput of baseline Apache (which lacks any of these security enhancements) with 320% slowdown. We stress that while this slowdown is significant, it represents atypically worst-case behavior: all sessions cached (never the case) and static content. On servers that distribute dynamically generated content, the overhead of protecting users' sensitive data will be amortized over far more application computation.

The original applications based on the OpenSSL library used single-process, monolithic designs. Hardening against SKD and oracle attacks requires three processes per SSL/TLS session: a private key monitor, a session monitor, and an unprivileged compartment for the SKN stage. Similarly, the hardened OpenSSH server and client use four processes per SSH session *vs.* the two employed by the baseline privilege-separated OpenSSH server. Apart from the process creation and page fault costs associated with fork and the memory copy costs associated with checkpoint-restore, anti-SKD and anti-oracle hardening incur overhead for additional context switches and the marshaling and unmarshaling of arguments and return values between compartments connected by pipes.

Again for the uncached workload, consider the throughput achieved by the full checkpoint-restore version of Apache (all defenses) *vs.* that achieved by one with the same full set of defenses, but implemented naively with fork. Checkpoint-restore offers a 20% throughput improvement over fork. While the end-to-end cost of inter-user isolation and oracle and SKD defenses is significant, the design of the underlying primitives used to implement compartments, though beyond the scope of this paper, appears to play a significant role in determining end-to-end performance.

4.6 Conclusion

We identified two class of practical exploit-based attacks on protocol implementations, the session key disclosure (SKD) attack and oracle attacks, that can disclose users' sensitive information even in the state-of-the-art, reduced-privilege applications such as the OpenSSH server and HiStar-labeled SSL web server. Thus, we demonstrate that privilege separation and DIFC alone do not secure the users' sensitive data against these attacks unless an application has been specifically structured to thwart them.

Our partitioning principles guide programmers when they privilege-separate applications employing cryptographic protocols to protect against the SKD and oracle attacks. In essence, these principles require a programmer to treat the session key and session key material as sensitive and limit privilege accordingly. The principles address oracle-prone functions with guidance of how to restrict oracles that they may expose.

To show that the principles are practical, we newly partitioned an OpenSSH client and extended the partitioning of a privilege-separated OpenSSH server. Further experience with the OpenSSL library suggests the principles generalize to other cryptographic protocols; they are target cryptographic building blocks, such as session key negotiation as well as encryptions, decryption, signing, and signature verification operations, rather than the protocols themselves.

We hope these principles will serve as a useful guide where there was none, we note that their application requires careful programmer effort. Still, our experience with OpenSSL shows that hardening a library once brings robustness against these attacks to the several applications that reuse that library. The latency cost of defending against SKD and oracle attacks is well within user tolerances for all applications we measured. Defending against SKD and oracle attacks does exact a cost in throughput on a busy SSL-enabled Apache server, however, reducing the uncached SSL/TLS session handshake rate of a server that isolates users by just under 30%, and the cached rate by 60%. While that cost is significant, as our comparison of fork and checkpoint-restore demonstrates, it depends heavily on the performance of underlying isolation primitives—a topic we believe merits further investigation.

Chapter 5

Sandboxing of Just-In-Time Compilation and Self-Modifying Code

We now explore the domain of language runtimes and discuss web applications, the threats that they impose on users' sensitive data, and our new mechanisms designed to thwart these threats.

Web applications are a form of highly untrusted software. They are designed by third-party developers untrusted by users, hosted on remote servers operated by other parties, and transfered over unprotected networks thus allowing their modification by a man in the middle. There are two parts in a web application: server-side code performs major computation on large data stored on the server, and client-side code which provides an interface to the server-side computation. The untrusted client-side code is shipped over the network and executes within a user's browser interacting directly with the user and thus omitting server and network delays that occur with server-generated, static content. The client-side code allows developers to build dynamic, highly-responsive web applications.

Despite its advantages, a web application's client-side code introduces a security threat to confidentiality and integrity of a user's sensitive data stored on the system and processed by the browser. The web application may borrow a browser's privilege and disclose/corrupt sensitive data from/in the file system and other programs executing on the same machine. Web browsers are user-privileged applications and thus provided with access to a user's files and may tamper with the execution of other software running under the same user ID, Section 2.1. Web browsers normally execute multiple web applications simultaneously each of which execute on behalf of a distinct site principal which is defined by an application's origin [61]—a combination of a protocol, domain name, and port number. For example, a web mail application handles user emails, and a web banking application manipulates a user's bank account; they execute in the same browser but act from different principals. In this multi-principal environment, there is a threat that one web application may attempt to corrupt or disclose sensitive data of another web application.

To restrict privileges of web applications and thwart the threats described above, browsers use *sandboxing*—a technique that allows safe confinement of software behavior. They sandbox untrusted code with language runtimes [1, 14, 40, 97]. As a result, web applications are mostly written in one of a handful of high level, dynamic programming languages specifically designed for untrusted content—

most commonly JavaScript [46]. The language runtimes allow browsers to restrict web applications direct access to system call API and thus prevent them from accessing files and tampering with other programs. In addition, the language runtimes provide fine-grained data access control within the browser. Thus, the browsers may enforce the Same Origin Policy (SOP) [61] which enables isolation among web applications ensuring that they access only resources from their origins. For example, an application retrieved from `https://email.com` is restricted from accessing resources, *e.g.,* pages, storage, and remote servers, of a web application labeled with the `https://bank.com` origin.

Sandboxing with a language runtime requires a browser to include the runtime in its trusted code base. However, modern language runtimes incorporate significant amount of possible flawed native code that implements extensive libraries and employ sophisticated techniques, such as Just-In-Time (JIT) compilation, inline caching, and garbage collection [60, 96, 108], mainly to close the gap between the performance of native execution and interpreted execution. These techniques and the native code complicate implementations of the language runtimes making it difficult to reason about their safety. Thus, the security of a browser that incorporates a language runtime depends on correctness of the runtime. Because of the complexity of language runtimes, there is a risk that a malicious web application may supply an exploit, compromise the language runtime, acquire the browser's privilege and disclose a user's sensitive data from the system and other web applications.

Sandboxing of *native code* may provide required safety for language runtimes. The entirety of dynamic software execution can be sandboxed, including the language platform, even if it uses just-in-time compilation, runtime code modification, or large bodies of legacy code. Thus, we can equalize the privilege of the language runtime and the privilege of a web application it executes. Most of the sandboxing mechanisms for native code rely on hardware protection domains, such as the common process abstraction [63, 79, 94, 98, 118, 120, 136]. They offer no universal applicability and particularly in the case of language runtimes when a fine granularity of protection domains is required. Web applications frequently interact with pages' content to render elements of user interface. Placing a web application and a page's content in different processes in order to enforce data access policies such as the SOP will have severe performance impact caused by hardware context switching [78].

Our work is based on Software-based Fault Isolation (SFI) [131]—a sandboxing technique that relies on machine code verification through static analysis and runtime software guards where the static verification is not possible. It is independent of the operating system and does not require the process abstraction and thus may place trusted and untrusted code in the same address space. However, SFI has not been applicable to sandboxing software that modify machine code at runtime, *e.g.*, JIT-enabled language runtimes. We remove this obstacle with an extension to SFI techniques and thus allow efficient sandboxing of language platforms that rely on just-in-time code generation and runtime code modification. The extension presents new mechanisms and safety constraints on code modification.

We implement our SFI extension for self-modifying code in Native Client (NaCl) [125, 132]—a production-quality SFI system. Our extension allows safely and efficiently add, delete, and modify code at runtime without race conditions and suspending threats even on hardware-threaded, concurrent

systems. The extension's overhead is modest manly because the new safety constraints are light and affect code modification rather than code execution.

We sandbox the V8 JavaScript engine [40] with extended NaCl. V8 is a part of the Google Chrome web browser [5], and web browsers of Android OS [3] and WebOS [44]. V8 implements JIT-compiler and makes heavy use of runtime code modification required for inline caching, a technique first introduced in implementation of Smalltack 80 [72]. Inline cache stores object properties such as member offsets directly in the machine code of a function. Each access to the inline cache checks first if the objects properties have not changed and then access the properties directly. Inline cache allows significantly improve the performance of a language runtime. The V8 Benchmark Suite [39] runs 12x slower when inline cache is disabled in unmodified V8. The positive impact of inline cache is in order of magnitude higher than any performance overhead introduced by the NaCl sandbox.

We ported the V8 JavaScript engine to two popular hardware platforms: the x86-32 and x86-64 Instruction Set Architectures (ISA) [55, 90]. NaCl employs different techniques for sandboxing on these platforms, and therefore its performance overhead differs across the platforms. NaCl uses hardware segments on the x86-32 architecture which allow to introduce low performance overhead for sandboxed code. The hardware segments are not available on the x86-64 ISA, and the sandboxing comes at a higher cost. As a result, our NaCl ports of V8 have different performance overheads but modest in both cases; the slowdown on the x86-32 architecture is 30% on average, and 60% on the x86-64 architecture.

The contribution of our work is an extension to SFI that enables support for self-modifying code, and sandboxing the V8 language runtime with extended NaCl. In particular, we implement:

- A set of locally-verifiable machine code constraints and mechanisms that allow safe, SFI-based sandboxing of machine code that requires runtime code modification on commodity hardware, despite the simultaneous execution of that code on other processors by untrusted user threads.
- Implementation of our SFI extension in Native Client for the x86-32 and x86-64 architectures, as well as ports of the V8 JavaScript runtime allowing software to embed languages like JavaScript as untrusted components with modest overhead.

5.1 Background: Software-based Fault Isolation

It is necessary to limit privilege of untrusted software to protect systems they run on from possible malicious operations that they may attempt to perform. Software-based Fault Isolation (SFI) [79, 107, 131] is a sandboxing technique that allows such protection by executing an untrusted code in confined and controlled environment. Commonly, this technique is employed for safe execution of third-party drivers, plugins, and code extensions. One particular example of SFI is Native Client (NaCl) [125, 132]—an open-source project that allows in-browser execution of untrusted native modules supplied by third-party server operators over the untrusted network.

NaCl enables safe execution of untrusted, multi-threaded, user-level machine code by imposing constraints on what instructions can be executed, in what sequence, and what memory addresses these instructions may use. NaCl achieves such tight control with static analysis and *software guards*—inline

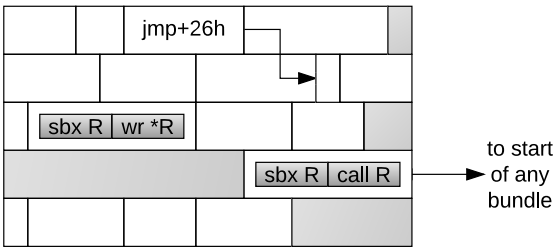
instruction sequences that perform runtime safety checks before executing potentially harmful operations. Each NaCl software guard is combined into a *pseudo instruction* with an adjacent, potentially-unsafe instruction, making each guard to constrain the behavior of only a single instruction.

NaCl controls system call interface for untrusted code by disallowing control-flow transfers to the operating system. In order to perform a system call, an untrusted module must call into NaCl's trusted service runtime which invokes a system call on behalf of the untrusted module. The service runtime provides a fine-grained control over system call interface and may implement arbitrary security policies. Although, it resides in the same address space as the untrusted code, the module cannot change service runtime code or data as ensured by the NaCl sandbox.

The NaCl platform provides a programming model for high-level languages using ILP32 (32-bit Integers, Longs and Pointers) primitive types and a portable subset of POSIX-like system calls. A developer is required to produce code that can be reliably disassembled. *Reliable disassembly* is achieved through the use of alignment and other machine-code constraints to guarantee runtime-reachable instructions can be statically identified.

NaCl performs static verification of the code to ensure that the code implements NaCl's safety constraints before executing it. Thus, NaCl must trust only its simple static verifier to ensure that the untrusted code can be safely executed in the sandboxed environment, but not a bulky compiler producing untrusted code. The verifier checks simple, local properties of the code, where those properties, in aggregate, imply global execution invariants. In order to guarantee that the required code properties hold for all possible execution paths, SFI restricts direct and indirect control flow transfers to target legitimate instructions, thus implementing a form of control-flow integrity [54]. Illegal control-flow transfers are those that may target a middle of an instruction or circumvent an inline software guard allowing unsafe execution of the guarded instruction. In case of a direct control-flow transfer, the verifier checks them statically, but such verification is not possible for indirect control-flow transfers as the destinations are known only at runtime. As in PittSFIeld [107], NaCl requires masking of all indirect control-flow transfers, so that they target fixed 32-byte (16-byte on ARM) bundles. The verifier ensures that none of instructions and pseudo instructions cross bundle boundary which guarantees that all indirect control-flow transfers target valid instructions. The alignment is provided via NOP padding which is also required to place CALL instructions in the end of bundles, so that a return address is always aligned. Native Client additionally prohibits use of the x86 RET opcode. Returns should be implemented with a POP/JMP sequence. Checking the return address in a register avoids a potential time-of-check-time-of-use race if it were checked on the stack. In addition, NaCl execution is constrained to a single, contiguous code region, with access to a restricted set of NaCl platform support routines, and memory access is limited to a single, contiguous region of untrusted data memory, which also contains the stacks for threads.

Native Client supports three architectures, x86-32, x86-64, and ARM. Like other SFI systems, Native Client also makes use of operating system and hardware support where appropriate, *e.g.*, to ensure that data is not executable, and that code is not writable and is loaded at the correct address. In particular,



Each fixed-width (32 byte) row denoting a NaCl instruction bundle starting at an aligned address. Vertical lines show instruction boundaries, while shaded regions show NOP padding; arrows indicate permitted direct and indirect control flow. Two NaCl pseudo instructions are shown as gradient-shaded boxes.

Figure 5.1: Overview of Abstract NaCl Machine Code.

on the 32-bit x86 platform, Native Client relies on hardware segments, not software guards, to constrain reads and writes to data memory, as well as code execution. However, on ARM and 64-bit x86, bounded segments are not available, and software guards must be used for each computed access to memory; therefore, the baseline overhead of Native Client varies on each platform. NaCl is designed to minimize the overhead of safety: even untrusted, hand-optimized media codecs can be executed at near full speed, since SFI allows safe use of hand-written assembly and model-specific instructions, such as the MMX and SSE extensions on x86.

Figure 5.1 gives an overview of the structure of NaCl machine code for an abstract ISA, showing the details of one NaCl *code region*, comprising five NaCl instruction bundles. Native Client independently verifies the safety of each such code region, one at a time. For the first instruction, a direct jump forward by 38 bytes (or 0x26), NaCl verification establishes that the target is a valid instruction boundary within the code region. The two remaining instructions are NaCl pseudo instructions that use the register `R` to write to memory at address `R` and call the function at address `R`. For these pseudo instructions, NaCl verification establishes the correctness of the software guards (`sbx`) used to sandbox the register `R`. For CALL instructions, both their target address and the pushed return address must be the start of an aligned instruction bundle. NaCl verification also imposes many structural constraints on machine code; for instance, direct jumps may not target the inner boundaries of pseudo instructions. NOP padding may need to be used to ensure proper alignment and, as clearly shown in Figure 5.1, large amounts of such padding may be required, especially around CALL instructions.

5.2 Threat Model

Our goal is to safeguard systems that execute untrusted code by means of language runtimes. We intend to enable applications to embed language runtimes as untrusted components and restrict their privilege according to the privilege of the code they execute. Thus, if an attacker compromises a language runtime with maliciously crafted untrusted code, she gains no additional privilege than the supplied code already has. An application developer should be able to restrict privilege of a language runtime to deny access to the system's sensitive components, *e.g.*, files and system call interface, as well as sensitive data within the application.

```
int nacl_dyncode_create(void* target, void* src, size_t size);
int nacl_dyncode_modify(void* target, void* src, size_t size);
int nacl_dyncode_delete(void* target, size_t size);
```

Figure 5.2: Extended NaCl's Interface for Runtime Code Modification.

Consider an unprotected web browser handling web applications that include JavaScript code. Assume that its language runtime contains remotely exploitable vulnerabilities which an attacker may compromise by suppling a specifically-constructed JavaScript code that when executes, injects native code of the attacker's choice. The compromised language runtime grants the injected code the browser's privilege and allows to disclose/corrupt sensitive data on the system and in other web applications.

Our threat model targets the specific scenario described above. Therefore, we assume that a remote attacker may compromise *only* the language runtime, and other software components, such as the operating system, software embedding the language runtime, Native Client sandbox, and other programs running on the system, are secure and resistant to attacks. Note that any code dynamically introduced within the NaCl sandbox, either JIT-compiled JavaScript or an attacker's injected native code, must satisfy NaCl's constraints at risk of not being executed. The application developer must correctly restrict privilege of the sandboxed language runtime; otherwise, the attacker may be able to access sensitive data allowed by the lax privilege.

Our threat model does not address denial-of-service attacks where an attacker attempts to disrupt a system-provided service by exhausting the system's resources such as CPU, storage, network, *&c*.

5.3 Design and Implementation

We extended Native Client with a number of trusted runtime calls that allow dynamic code insertion, modification, and deletion. The added calls are required by language runtimes and enable their sandboxing with NaCl, as shown in Figure 5.2.

The typical use of our extension is when a JIT-enabled language runtime compiles a function that is about to be executed into native code. The native code is saved in a temporary buffer (in a data segment). Next, the language runtime invokes the *nacl_dyncode_create* service runtime call, whose arguments are *target*, the location in an executable segment in which to place the newly generated native code, and *src*, the buffer where the code is currently held. The trusted service runtime will install this code in the requested place after validating it.

After the code's creation, if the language runtime needs to modify the existing code, it uses the second service runtime call, *nacl_dyncode_modify*. The caller must specify a pointer to the code to be replaced in an executable segment and a pointer to the new code provided by the untrusted language runtime.

Finally, if the language runtime decides that some piece of code is not needed anymore, it may request that the service runtime delete it with by calling *nacl_dyncode_delete*.

5.3.1 Dynamic Code Creation

The unit of dynamic code insertion and deletion is a *region*. A region may be of a variable length, but its start and end addresses must be bundle-aligned. The NaCl service runtime deletes only whole regions; it is not possible to free a part of a region. When the service runtime processes the *nacl_dyncode_create* call, it performs the following operations:

- Verifying that the destination address of the inserted code is in a NaCl module's executable memory and bundle-aligned.
- To avoid the time-of-test-time-of-use race condition, the code is copied to the private memory of the trusted NaCl runtime.
- NaCl validates the code using its standard validator.
- The target address range in the NaCl module's executable memory is checked to be unused and reserved in one atomic operation.
- The code is safely copied to the target address.

JIT-enabled language runtimes introduce code in small portions on a per-function basis, *e.g.*, when executing JavaScript code, a function call may result in compilation and insertion of the function's native representation if it is not there. We significantly reduce overhead related to NaCl validation as we require validating only newly introduced code, but not the code already residing in the untrusted module. Because of this optimization, the NaCl validator must be able to validate independent code regions. Therefore, the *direct* and indirect control-flow transfers between code regions must target the beginning of a NaCl bundle so that they can be validated locally without looking at the destination instructions in other regions. To ensure this requirement, NOP alignment may be required.

An untrusted NaCl module may attempt to exploit the copying of validated code to the executable memory of an untrusted module. While a copy operation is performed, a different untrusted thread may concurrently execute copied instructions in the insertion region in an attempt to execute a disallowed instruction that may appear as a result of partial copying of a full instruction and thus break out of the NaCl sandbox. In order to prevent this scenario and guarantee safe copying of a new code region, NaCl fills the first byte of each NaCl bundle with the HLT instruction before copying new instructions to this bundle. The HLT instructions prevent execution of bundles and a region in whole by another untrusted thread while copying is carried out. When the bundle is filled in with the new instructions the HLT byte is substituted with the intended value.

5.3.2 Dynamic Code Modification

When enabling dynamic code modification, we must ensure that the constraints imposed by the NaCl sandbox are not violated. In order to preserve the NaCl sandbox's guarantees, we introduce additional constraints on dynamic code modification that are checked by the verifier. Figure 5.3 lists all these constraints. Essentially, the constraints in Figure 5.3 imply immutability of instruction boundaries and the NaCl guard instructions in the modified code. These immutability requirements significantly restrict

1. NEW must satisfy all NaCl safety verification constraints, as outlined in Section 5.1.
2. Both NEW and OLD must start at the same address, be of equal size, and lie within a single code region.
3. Any direct control-transfer instructions in NEW must target valid instruction boundaries in the *same* code region.
4. NEW and OLD must start and end at instruction boundaries, and all instruction boundaries between must be identical.
5. No pseudo instructions are added or removed. NEW may not introduce new pseudo instructions. All pseudo-instructions in OLD must occur in NEW and have identical guard instructions.

Machine code OLD is replaced with machine code NEW.

Figure 5.3: Extended NaCl's Constraints on Runtime Code Modification.

```
// for an instruction pair OLDI and NEWI
if (diff of (OLDI, NEWI) is aligned qword) {
  //fast path
  atomic aligned qword write to update OLDI;
} else {
  // slow path
  OLDI[0] = 0xf4; // HLT instruction
  serialize(); // barrier
  OLDI[1:n] = NEWI[1:n];
  serialize(); // barrier
  OLDI[0] = NEWI[0];
}
```

Figure 5.4: Pseudo Code for Safe Code Modification.

possible code modifications; however, in practice, we found that they do not limit code modifications required by inline caching, the primary goal of self-modification in the ported language runtimes.

Code validation of dynamically modified code is performed at bundle granularity and applied only to bundles that are being modified. The NaCl validator may need to validate an additional bundle if it is targeted by a direct control-flow instruction from the modified code to ensure that the destination of the control-flow instruction is a valid instruction.

When *nacl_dyncode_modify* is invoked, some untrusted threads may concurrently be executing the modified region. Therefore, the service runtime must ensure that while it is modifying code, an untrusted thread executes either the old or a new instruction, but no other instruction which may be composed from bytes of the old and new instructions. Such corruption is proven to be possible by Sundaresan et al. [128], and we also verified it with our own experiments.

AMD and Intel processors support atomic code modification. An 8-byte aligned modification is viewed atomically by the processor according to the relevant documentation [55, 90][1], and our own experiments confirm this behavior. Thus, we can safely modify one 8-byte-aligned instruction at a time.

[1]Page 8-8 Vol. 3A in Intel 64 and IA-32 Architectures Software Developer's Manual [90], page 48 in AMD64 Architecture Programmers Manual Volume 1: Application Programming [55].

If a modified instruction is not 8-byte-aligned or longer than 8 bytes, it is still possible to safely modify it, but ensuring safety in such cases requires further mechanisms. The pseudocode in Figure 5.4 shows how we perform safe code modification. When an instruction cannot be modified via our fast path, we write a HLT byte to the start of the modified instruction, thus preventing its execution.

We then issue a serialization barrier to synchronize the instruction stream and code memory view for all hardware threads, including other cores or processors, as required by the Intel 64 and IA-32 Architectures Manual (page 8-4 Vol. 3A) [90]. The one-byte write is atomic, therefore an untrusted thread may execute either the old instruction or the HLT instruction before the first serialization barrier. In between the barriers, the thread may only execute the HLT instruction. Finally, after the second serialization barrier, the thread may execute either the HLT instruction or the new instruction. The second serialization barrier guarantees that a concurrently executing thread will not observe an illegal instruction composed from the first byte of `NEWI` and the remaining bytes of `OLDI`. One might think that because x86 is a total store ordered (TSO) architecture–*i.e.,* each core views other cores' writes in order–that the second serialization barrier may not be necessary. However, the TSO guarantee applies only to data memory, and does not take account of instruction prefetch. The x86 architecture reference manual explicitly requires that when one core modifies code, a serializing instruction must be executed before another core attempts to execute the modified code (page 8-4, Vol. 3A) [90]. The second serialization barrier meets this requirement.

We omit the last serialization barrier after the last modifying instruction in Figure 5.4. We rely on higher-level software to implement synchronization that would allow an executing thread to observe the last modification in its execution stream before executing the modified instruction, if such synchronization is required. Omitting the third serialization barrier does not allow the execution of an illegal instruction as the concurrent thread may execute only either the HLT instruction or the new instruction.

Our technique relies on a serialization barrier primitive. The common approach on x86 processors is to use a serializing instruction (such as cpuid) on all hardware threads that must observe one another's prior writes. As NaCl sandboxing is a user mode mechanism, we require a serialization barrier that can be triggered from user mode. Conveniently, certain system calls serialize all processors as a side-effect. We used the mprotect system call, which triggers inter-processor interrupts of remote hardware threads for a "TLB shoot-down", serializing all processors. NaCl invokes mprotect to modify the execute permission on a dummy memory page allocated specifically for this purpose. The kernel implementation of mprotect writes into the CR3 register and thus serializes all processors in the course of performing the TLB shoot-down.

5.3.3 Dynamic Code Deletion

Language runtimes can delete redundant code regions with the nacl_dyncode_delete service runtime call. The service runtime must ensure that there are no untrusted threads executing this region before deleting it and possibly reusing the region for new code. Otherwise, there is a risk that a suspended thread may wake up and start execution in the middle of a substituted instruction which violates NaCl sandbox constraints. To prevent this case, the service runtime performs a number of operations. First, it writes

HLT instructions in the beginning of each bundle of the deleted region to prevent any thread entering this region. Note, that direct and indirect control-flow transfers between regions always target beginning of a bundle. Then the service runtime waits while each thread makes a service runtime call. Thus, it registers that the thread has left the deleted region. If the thread makes a call from the deleted region, its return address will be bundle-aligned and target a HLT instruction. The alignment of return addresses is required by the NaCl sandbox, Section 5.1. After ensuring that all the threads left the deleted region, the service runtime safely deletes the region.

If only one thread executing, the service runtime immediately performs the deletion of the code region. With multiple executing threads, the call to nacl_dyncode_delete returns EAGAIN and performs the operations stated above. When a subsequent invocation of this call with the same arguments returns successfully,, this means that all threads have left the region and the region has been deleted. This nonblocking property of the nacl_dyncode_delete service runtime call allows a thread to perform useful work while waiting for deletion of a region.

5.4 Evaluation

5.4.1 Security Analysis

Native Client employs the standard hardware page protection mechanism to restrict execution of data segments and modification of code segments within an untrusted module. This prohibits conventional code injection attacks where a vulnerability allows to redirect execution to unvalidated code injected into a data segment. An attacker may attempt to reuse existing code in an untrusted module's address space and launch an attack similar to the return-to-libc attack [126] which does not require code injection. However, the NaCl sandbox constrains the untrusted module's memory accesses to target the untrusted address space and proxies all system calls through the trusted service runtime, Section 5.1, where NaCl safely examines system call arguments and enforces a programmer's security policies. Thus, such reusing of untrusted code does not allow to escape the NaCl sandbox and escalate privilege. The only way an attacker may introduce new code is by using extended NaCl's interface for code modification, Figure 5.2. However, NaCl adds the new code only if it implements the sandboxing constraints and reject it otherwise. Thus, the added code is sandboxed in the same way as the code that already resides in the untrusted module.

If an attacker compromises a NaCl-sandboxed language runtime and injects malicious code, she gains no extra privilege as NaCl sandboxes the injected code and reduces its privilege accordingly. Thus, the injected code is restricted from accessing sensitive data outside its untrusted address space and cannot employ the system call interface to disclose sensitive data stored on the system.

5.4.2 Porting Effort

To evaluate our extension to the NaCl sandbox, we produced ports of V8 version 2.2.19 for the x86-32 and x86-64 architectures and a port of V8 Crankshaft version 3.1.4 for the x86-32 architecture. We ported two versions of V8-32 because at the time of publishing of our paper [57], the V8 team released the new Crankshaft optimizing compiler [109] that enables function inlining, loop-invariant code mo-

tion and common subexpression elimination optimizations [70, 111], and we were keen to see how the new compiler performs with the extended NaCl sandbox. The port of V8-32 version 2.2.19 as well as implementation of the NaCl extension on the x86-32 architecture is a work of our collaborator, Jason Ansel.

NaCl on both platforms prohibits data in executable memory of an untrusted module, and the NaCl validator requires only valid instructions in the executable memory. However, V8 places functions' code with appended metadata, *e.g.*, relocation records, in the executable memory which conflicts with the above requirement. We modified V8 to split code and metadata and place them within different memory locations so to satisfy the NaCl validator. In the V8-32 Crankshaft version, we took a different approach and masked some metadata with `PUSH` instructions. The masking `PUSH` instructions are unreachable by functions' code, but each instruction contains 4 bytes of metadata within its argument. The instructions encoding metadata appear as valid ones for the NaCl validator.

NaCl's sandboxing mechanisms differ on the x86-32 and x86-64 architectures. On the x86-32 architecture, hardware memory segments and software guard instructions ensure the safety of indirect memory access and control-flow instructions. The software guards mask the target addresses of control-flow instructions and force them to target beginning of bundles. In order to satisfy the NaCl validator, we modified the V8-32 compiler to emit the required guard instructions, align `CALL` instructions to bundles' ends, and replace `RET` instructions with `POP` and `JMP` sequences, as discussed in Section 5.1. As for the x86-64 architecture, the hardware segments are not available there. Therefore, in addition to the changes to the V8-32 compiler, our modified V8-64 compiler emits software guards to ensure that memory access and indirect control-flow instructions target an untrusted module's address space. On the x86-64 architecture, NaCl imposes constraints on modification of the `RSP` and `RBP` registers. These registers must contain legitimate address (targeting untrusted address space) at all times [125]. Thus, our port of V8-64 emits software guards to sandbox instructions that modify the `RSP` and `RBP` registers. We altered 171 methods of a class responsible for code generation in order to implement required software guards in V8-64.

NaCl enforces the ILP32 data model on both architectures, x86-32 and x86-64, to facilitate source code portability between the architectures. However, the x86-64 architecture does not support the 4-byte `PUSH` and `POP` instructions; thus, the stack is 8-byte aligned despite NaCl's ILP32 data model. This stack alignment mismatch prevents V8-64 from correctly calculating pointer offsets to locate data on the stack. V8-64 assumes that the stack alignment is the same as the pointer size defined by NaCl's ILP32 model and calculates 4-byte-multiple displacements in instructions referencing the stack. As a result, all these instructions miss their targets because the stack is 8-byte aligned by the `PUSH` and `POP` instructions. Thus, in oder to make V8-64 work within the NaCl sandbox, we had to manually inspect computed data references and classify whether a reference targets the stack or heap. Then, for the stack references, we introduced 8-byte-multiple offsets, and for the heap references 4-byte-multiple offsets. 393 operations required our attention; 306 of them referenced the stack and had to be converted to use 8-byte-multiple offsets. Often, it was not possible to identify if an operation emitted by V8's JIT compiler

is a stack-referencing or heap-referencing one by just looking at the compiler's source code. Therefore, we evaluated such instructions at runtime by setting debugging break points at these instructions and analyzing their target addresses. As you may imagine, this process was tedious.

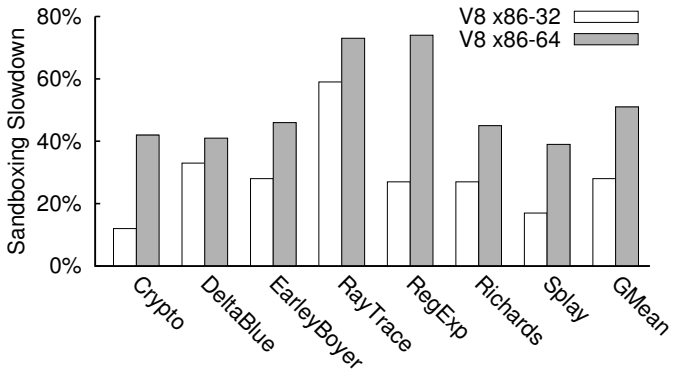
The ILP32 data model affects the optimized representation of integers in V8-64. V8-32 stores integers as direct values where the list significant bit is used as an identification marker reducing the effective range of integers to 31 bit. Thus, the bottom half of the 32-bit integer range is stored in the optimized representation, direct values shifted to the left by one bit, and the top half of 32-bit integer range is stored as JavaScript objects on the heap. The access to such boxed integers is significantly more expensive than the access to the integers stored as direct values. Baseline V8-64 stores the full range of 32-bit integers as direct values. It leverages 8-byte registers and 8-byte integer size and shifts a 32-bit integer value to the left by 4 bytes thus setting the one-bit identification marker to zero. In our port of V8-64 with the ILP32 model, we had to adopt V8-32's integer representation model and re-implement arithmetic operations on integers according to the new model. Apart from the substantial porting effort, the integer representation change impacts performance of some tests in the SunSpider100 Benchmark Suite, as discussed below.

5.4.3 Performance Evaluation

In this section, we evaluate the cost of privilege reduction with NaCl sandbox when applied to language runtimes, in particular the V8 JavaScript engine. Running a language runtime in a reduced-privilege compartment allows to protect sensitive data from unauthorized disclosure and corruption. Results in this section are from a quad-core Intel Xeon X5550 Nehalem 2.67 GHz processor, except Table 5.2 as noted. Our test system run Ubuntu 10.04, kernel version 2.6.32. Performance results are an average of 10 runs. The observed standard deviation ranged from 0 to 2.32%. We summarize performance measurements using the geometric mean [81]. This section presents the performance results of V8-32 version 2.2.19 and V8-64 version 2.2.19, and the Section 5.4.3 discusses our latest evaluation of the Crankshaft compiler in V8-32 version 3.1.4.

Figure 5.5 and Figure 5.6 show the overheads of our sandboxed V8-32 and V8-64 running the V8 Benchmark Suite [39] and the SunSpider100 Benchmark Suite [35]. We have made a change in the SunSpider Benchmark Suite to report time of 100 runs for each benchmark instead of a single run. This change enables more accurate measurements. All reported overheads are relative to unmodified V8.

As shown in Figure 5.5, the geometric mean of all benchmark overheads is 21% for V8-32 and 51% for V8-64. One of the highest performance overheads shows the RayTrace benchmark in both cases. To investigate the cause of such performance overhead, we run this benchmark under PIN [101], a binary instrumentation tool, and pfmon [13], a performance monitoring tool based on hardware performance counters. With PIN, we counted the number of executed instructions for baseline V8-64 and V8-64 running within the NaCl sandbox. The latter one executes 1.8x more instructions than the unmodified version. 39% of the additional instructions are NOP padding required for bundle-alignment (preventing instructions from crossing bundle boundaries) and aligning `CALL` instructions. The second largest contributor of the additional instructions is software guards of memory access instructions, they



NaCl sandboxing slowdown (lower is better) for the V8 JavaScript Benchmark Suite, relative to unmodified V8. GMean is the geometric mean of the overheads of all benchmarks in the suite.

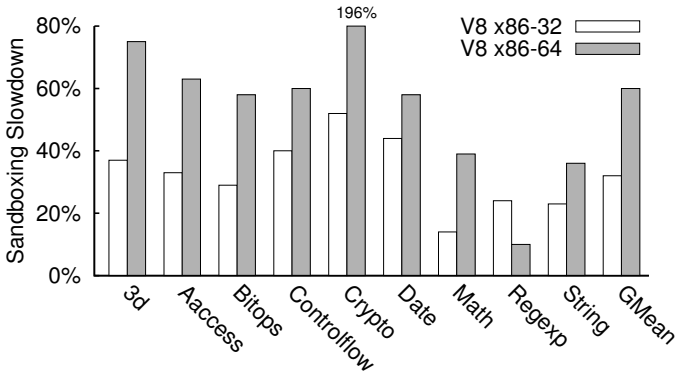
Benchmark	Unsafe		Sandboxed	
	x86-32	x86-64	x86-32	x86-64
Crypto	4380	4511	3910 (12%)	3176 (42%)
DeltaBlue	6555	5380	4921 (33%)	3825 (41%)
EarleyBoyer	20332	19370	15827 (28%)	13247 (46%)
RayTrace	9277	7979	5849 (59%)	4615 (73%)
RegExp	3367	3534	2660 (27%)	2035 (74%)
Richards	4911	4659	3864 (27%)	3217 (45%)
Splay	15305	14160	13098 (17%)	10188 (39%)
GMean	**7528**	**7059**	**5868 (28%)**	**4683 (51%)**

Raw benchmark scores (higher is better) for unmodified V8 (unsafe) and sandboxed V8, with sandboxing relative slowdown in parentheses.

Figure 5.5: NaCl Sandbox Overhead for the V8 Benchmark Suite.

make 38% of additional instructions. The rest of the instructions comes from other software guards and the NaCl service runtime. pfmon using hardware performance counters shows that V8-64 running in the NaCl sandbox experience 4x increase in branch misprediction rate. NaCl requires replacing branch-predictable `RET` instructions with indirect `JUMP` instructions that always result in branch mispredictions. Sandboxing also increases instruction cache pressure which signaled by 2x increase in L1 instruction cache misses and 2x change in instruction TLB misses.

The SunSpider100 Benchmark Suite shows similar average performance overhead as the V8 Benchmark Suite with 32% for sandboxed V8-32 versus unmodified V8-32 and 60% performance overhead for sandboxed V8-64 as compared with baseline V8-64. The latter case has an outlier which has twice overhead as any other benchmark. The crypto benchmark group is composed of three benchmarks: crypto-aes with 1.8x slowdown, crypto-md5 with 5.0x slowdown, and crypto-sha1 with 3.9x slowdown. The latter two composed from frequently-invoked, small functions and rely on heavy use of 32-bit integer arithmetic. The small functions result in sandboxed and NOP-aligned `CALL` instructions that increase the code base, and replaced `RET` instructions increase the branch misprediction rate. Because of the difference in the integer representation, discussed in the previous section, our port of V8-64 stores integers greater than 2^{31} as boxed heap integers instead as direct register values. In baseline V8-64, all 32-bit integers are stored as direct values. Access to a heap number is significantly more expensive than the access to a direct value; in addition, the former access type requires execution of NaCl's memory access guards. All these factors contributes to the significant performance overheads of the crypto-md5 and



NaCl sandboxing slowdown (lower is better) for the SunSpider100 JavaScript Benchmark Suite, relative to unmodified V8. Each bar represents a group of up to four benchmarks. GMean is the geometric mean of all benchmark overheads.

Benchmark	Unsafe		Sandboxed	
	x86-32	x86-64	x86-32	x86-64
3D	3097	3434	4230 (37%)	5996 (75%)
Access	2867	3248	3815 (33%)	5296 (63%)
BitOps	2240	2047	2892 (29%)	3235 (58%)
ControlFlow	179	199	250 (40%)	318 (60%)
Crypto	1193	857	1812 (52%)	2538 (196%)
Date	2060	2236	2970 (44%)	3541 (58%)
Math	2310	2374	2639 (14%)	3295 (39%)
RegExp	1097	957	1359 (24%)	1057 (10%)
String	5147	5269	6325 (23%)	7186 (36%)
GMean	**1693**	**1676**	**2241 (32%)**	**2689 (60%)**

Raw benchmark scores (lower is better) for unmodified V8 (unsafe) and sandboxed V8, with sandboxing relative slowdown in parentheses.

Figure 5.6: NaCl Sandbox Overhead for the SunSpider100 Benchmark Suite.

crypto-sha1.

We identified different sources of performance overhead in sandboxed V8-32 and V8-64 and estimated the contribution of each of them. Table 5.1 shows the estimated breakdown of sandboxing overhead for the V8 Benchmark Suite. The numbers for the table were produced by disabling one source of overhead at a time and measuring the performance improvement this gives. Because the performance overhead of different features are not independent, the numbers in Table 5.1 are only estimates. NOP padding required for bundle-alignment and aligning `CALL` instructions introduces significant execution slowdown. In the case of sandboxed V8-64, NOP padding results in 37% of performance degradation. Software guards add 24% of performance overhead, and substitution of `RET` instructions with the `POP/JUMP` sequence contributes further 22% of performance overhead. As for V8-32, the respective number are 23%, 25%, and 17%. Other sources of performance overhead, not shown in Table 5.1, are invocation of service runtime calls, slower dynamic code generation, copies required for modified code, and changes required to separate code and meta-data. As mentioned above, boxing top half of 32-bit integers that is required in our V8-64 port significantly slows down some benchmarks.

Table 5.2 compares the overhead of our extended NaCl sandboxing when running the V8 Benchmark Suite on several modern processors, implementing different microarchitectures. For V8 on x86-32, overheads are consistent, ranging from 28% to 34%. For x86-64, overheads range from 41% to 56%,

Source of Overhead	V8-32	V8-64
NOP padding to align bundles	4%	16%
NOP padding to align calls	19%	21%
Software guards for function returns	25%	22%
Software guards for indirect jumps and indirect memory accesses	17%	24%
Runtime validation of code modifications	2%	5%

Indirect memory access guards are required only for V8-64. These measurements are from the V8 Benchmark Suite, and they were generated by disabling NaCl sandboxing features one at a time. Since the performance impact of these features is not independent, these overheads are not additive.

Table 5.1: Analysis of Sources of NaCl Sandboxing Slowdown.

Microarchitecture	Unsafe		Sandboxed	
	x86-32	x86-64	x86-32	x86-64
Intel Xeon X5550 Nehalem 2.7GHz	7528	7059	5869 (28%)	4683 (51%)
Intel Core2 Quad Q6600 2.4GHz	5612	5128	4535 (24%)	3296 (56%)
AMD Phenom II X4 905E 2.5GHz	5030	4793	4026 (25%)	3390 (41%)
AMD Athlon 4450E 2.3GHz	3853	3447	2856 (35%)	2385 (45%)
AMD Opteron 8214 HE 2.2GHz	3633	3224	2701 (35%)	2226 (45%)
Intel Atom N450 1.7GHz	1395	1176	1041 (34%)	589 (100%)

The numbers (higher is better) are from the V8 Benchmark Suite running natively (Unsafe) and in our extended NaCl sandbox, the relative slowdown is in parentheses.

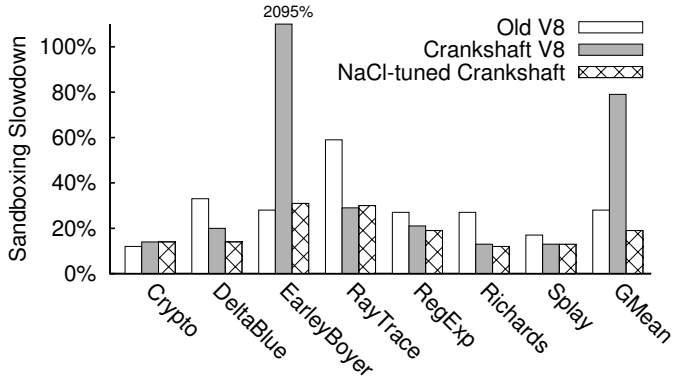
Table 5.2: Performance Scores on Different Microarchitectures for the V8 Benchmark Suite.

and the numbers seem to suggest consistently less relative overhead on AMD processors than on the fast Intel processors. The Intel Atom is a notable outlier, with 100% sandboxing slowdown measured for V8-64. While we haven't yet fully explored the Atom's outlier performance, we note that these results match those previously reported for the x86-64 NaCl sandbox [125]. Interestingly, raw performance is better on x86-32 for both the unsafe and sandboxed versions, most likely because V8 has been more heavily optimized for x86-32.

Performance Evaluation of Crankshaft V8

While we were working on extending NaCl to support sandboxing of self-modifying code, the V8 team released a new implementation of the JavaScript engine with the Crankshaft compiler [109]. The new compiler significantly improves JavaScript execution performance by enabling dynamic, profile-driven optimizations, in particular, loop-invariant code motion [111], common subexpression elimination [70], better register allocation, and function inlining. In addition, Crankshaft V8 allows use of unboxed 32-bit integers.

We produced two ports of Crankshaft V8-32 version 3.1.4, the initial port and then the one with NaCl-specific tweaks. In the latter implementation, we increased the number of a function's invocations required to trigger generating an optimized (specialized) version of the function. Thus, we account for the increased cost of function generation due to the requirement of code validation. In a specialized version of a function, V8 places a lot of bailout jumps implemented with indirect jump instructions that redirect execution to the function's general version if the execution deviates from the specialized case. The indirect jumps require significant amount of NOP padding and negatively affect branch misprediction rate. To avoid this overhead, we implement bailout jumps with a small table of direct-jump trampolines.



NaCl sandboxing slowdown (lower is better) relative to unmodified V8-32 implementations (Old V8-32 version 2.2.19, Crankshaft V8-32 version 3.1.4, NaCl-tuned Crankshaft V8-32 version 3.1.4) for the V8 Benchmark Suite. GMean is the geometric mean of all benchmark overheads.

Benchmark	Old V8-32	Crankshaft V8-32	Crankshaft⋆ V8-32
Crypto	3910 (12%)	12521 (14%)	12583 (14%)
DeltaBlue	4921 (33%)	13718 (20%)	14413 (14%)
EarleyBoyer	15827 (28%)	1030 (2095%)	17294 (31%)
RayTrace	5849 (59%)	8923 (29%)	8908 (30%)
RegExp	2660 (27%)	2274 (21%)	2298 (19%)
Richards	3864 (27%)	10862 (13%)	10955 (12%)
Splay	13098 (17%)	3704 (13%)	3699 (13%)
GMean	**5868 (28%)**	**5459 (79%)**	**8250 (19%)**

Raw benchmark scores (higher is better) for sandboxed V8-32 implementations (Old V8-32 version 2.2.19, Crankshaft V8-32 version 3.1.4, NaCl-tuned Crankshaft V8-32 version 3.1.4). In parentheses, there is the sandboxing slowdown relative to corresponding unmodified V8-32 implementations. Crankshaft⋆ V8-32 denotes NaCl-tuned Crankshaft V8-32 version 3.1.4.

Figure 5.7: NaCl Sandbox Overhead for the V8 Benchmark Suite.

Figure 5.7 shows the performance of our V8-32 ports running the V8 Benchmark Suite within the NaCl sandbox. We compare the performance of the following NaCl ports: V8-32 version 2.2.19, Crankshaft V8-32 version 3.1.4, and NaCl-tuned Crankshaft V8-32 version 3.1.4, as well as their slowdowns relative to the corresponding baseline implementations. Our tuned implementation of Crankshaft V8-32 beats the performance of the unmodified V8-32 version 2.2.19 used previously, Figure 5.5. It is particularly gratifying that in a span of a few months, the original overhead of our extended NaCl sandboxing has been more than offset by the independent optimization of the language runtime. In the tuned Crankshaft V8-32, the relative overhead of the NaCl sandbox is reduced by almost a third, going from 28% to 19%. We believe this decrease is because of Crankshaft's function inlining which reduces both NaCl NOP padding as well as the function-return branch misprediction rate. The SunSpider100 Benchmark Sute shows a similar absolute performance improvement, and the sandboxing slowdown falls from 32% to 24%.

5.5 Conclusion

Software systems embed complex and thus error-prone language runtimes to execute and sandbox untrusted code. The complexity of language runtimes comes from implementing advanced techniques such as Just-In-Time (JIT) compilation, large libraries of native-code support routines, and intricate mechanisms for multi-threading and garbage collection. Thus, a language runtime may provide a way for a remote attacker to exploit its vulnerabilities and compromise a system that embeds the language run-

time. For example, web browsers include JavaScript engines to execute client-side code supplied within untrusted pages. A malicious page developer may construct JavaScript code that when executed by the browser, exploits the browser's language runtime, obtains the browser's full privilege, and discloses sensitive data allowed by the privilege.

Privilege reduction is a powerful technique which can mitigate the harm that an attacker exploiting error-prone software components may cause. Majority of privilege reduction primitives are based on a process abstraction required to protect sensitive data. They make a poor fit for sandboxing software components that require frequent access to sensitive data placed in a separate process for protection. Each access to sensitive data causes a context switch between a process isolating an untrusted software component and a process isolating sensitive data. This context switch overhead significantly harms the system's performance. Language runtimes may fall into this category, *e.g.*, JavaScript code executed in the browser frequently manipulates pages' content to render user interface.

Software-based Fault Isolation (SFI) is a sandboxing technique that requires no process abstraction and may isolate untrusted code and sensitive data within a single process. It can mediate access to sensitive data as well as access to the OS's system call interface with machine code verification through static analysis and software guard instructions. However, it has not been applicable to sandboxing software that modify machine code at runtime which is the case of language runtimes. Language runtimes heavily rely on JIT compilation and runtime optimizations requiring code modification, *e.g.* inline caching, to achieve performance comparable to the performance of native execution.

Our work demonstrates that it is possible to sandbox programs that make use of self-modifying machine code with traditional software-based fault isolation techniques. We extended the SFI techniques with new safety constraints that allow untrusted, multi-threaded machine code to safely add, delete, and modify itself at runtime. We implemented our extension in Native Client, a production-quality SFI sandbox, and sandboxed the V8 JavaScript engine with extended NaCl. We produced ports of V8 2.2.19 for the x86-32 and x86-64 architectures as well as a port of V8 3.1.4 for the x86-32 architecture that implements the Crankshaft compiler and a set of new optimizations. The performance cost of NaCl sandboxing is modest: 32% for V8-32 and 60% for V8-64 as demonstrated by the V8 Benchamark suite. The difference in execution slowdowns across architectures is due to architecture-specific sandboxing mechanisms that NaCl employs on each platform. The Crankshaft compiler significantly improves the performance of unmodified V8 with a range of optimizations. It also reduces the performance overhead introduced by the extended NaCl sandbox; the NaCl slowdown drops from 32% to 19%. Thus, our performance evaluation shows that it is possible to sandbox untrusted self-modifying code at acceptable performance cost.

Chapter 6

Keeping Sensitive Data in Browsers Safe with ScriptPolice

This chapter explores the domain of web applications and presents our work on improving secrecy guarantees for users' sensitive data in web applications by enforcing simple privilege reduction polices in web browsers.

Today's ubiquitous web applications run as ensembles of computation on network-accessible server(s) and computation in a web browser executing locally on the user's hardware. As web applications have grown in sophistication to encompass venerable native-code "desktop" applications, including document editors, spreadsheets, mapping tools, and highly interactive games, the browser has grown in functionality to accommodate them, most notably by executing JavaScript code retrieved over the network. Such JavaScript code is routinely written by untrusted parties—by the operators of web sites, or in some cases, by untrusted third parties whose JavaScript libraries a site's developer uses for convenience. In addition, users enhance their browser's functionality by downloading and installing *extensions,* also typically written in JavaScript, which read and write the content displayed in browser windows by web applications, communicate with the network, or both. Extensions see wide use. For example, AdBlock Plus, which blocks the display of ads in pages, has been downloaded more than ten million times from the Google Chrome web store.

As users routinely manipulate sensitive information with web browsers, by using, *e.g.,* web-based email, online banking, and medical record-browsing applications, the execution of untrusted JavaScript in the browser presents urgent confidentiality concerns. The primary means by which the browser protects sensitive information from unauthorized disclosure to a remote party is the well-known *same-origin policy,* or SOP, which roughly dictates that a script retrieved from one domain name can only read and write content retrieved from the same domain name [61]. Thus, a script in a weather site open in one tab cannot reach across origin boundaries to read the user's bank balance displayed in another tab. Unfortunately, the SOP *fails* to protect all sensitive data in the browser. Extensions *cannot* be bound by the SOP, as they often must read and write content from all origins in order to do their job. Thus, the SOP does not protect sensitive information in web applications from extensions, many of which have full visibility into all tabs' content. Extensions can communicate with the network, as well. The consequent high

privilege of extensions—the ability to read all tabs and send to the network—creates an attractive target for miscreants to abuse to exfiltrate sensitive data from the browser without authorization.

If one examines the relationship between extensions and pages further, two distinct threats become apparent. First, an extension may have originally been written deliberately to read sensitive information from one or more tabs and directly disclose that information to an unauthorized remote party. Second, an extension may not have been designed deliberately to disclose sensitive information, but may be vulnerable to a *privilege escalation* attack, in which a maliciously constructed page manages to inject JavaScript code into an extension, and thus misappropriate the elevated privileges of the extension to exfiltrate sensitive information from another tab. Here, a maliciously crafted page exploits an honest but vulnerable extension. In both cases, the end result is the same: unauthorized exfiltration of sensitive information from the browser.

The threat of malicious extensions is not merely hypothetical. We identified an extension for Google Chrome that silently sends potentially sensitive content from a user's open tabs to a remote server. This communication is unexpected by the user as it is totally extraneous to the extension's advertised functionality (namely, displaying vCards embedded in pages in pop-up windows). We have further identified and exploited vulnerabilities in widely used extensions for Google Chrome that allow a carefully constructed web page to send any cookie or page content from any of a browser's tabs to a remote attacker. For example, we have written a page containing JavaScript that, when viewed in a browser with a popular RSS feed subscription extension installed, sends the browser user's Gmail authentication credentials (stored in a cookie) to the remote attacker (in flagrant violation of the SOP, and of the confidentiality of the user's sensitive data). In this *browser privilege escalation* attack, our page injects JavaScript into the RSS extension. Many other such vulnerabilities have been reported [69].

In this chapter, we describe how to defend a browser user's sensitive data robustly against both these categories of attack by confining the execution of JavaScript extensions to comply with simple but effective policies. We describe two classes of policy: a family of three *containment* policies, which block the exfiltration of sensitive information to the network by an extension—whether the extension is maliciously written or honest but vulnerable to injection; and a *prevention* policy, which blocks pages from injecting code into extensions. A practical system to defend against these attacks must have several crucial attributes. It must not be evadable by maliciously written extensions, which may be crafted with knowledge of the defense's design. It must work with *today's* "legacy" extensions, without requiring changes to their code, and without breaking their correct functioning. And finally, it must not reduce the performance of the web browser noticeably, as users are generally willing to "pay" very little for enhanced privacy.

Table 6.1 summarizes threats and threat-coverage of our policies. As shown in the table, the prevention policy protects vulnerable extensions not only from unauthorized data exfiltration but also from an attack we term *mis-execution* where a maliciously crafted page injects JavaScript code into an extension so as to interfere with the extension's correct execution. Thus, we envision the browser enforcing containment and prevention policies simultaneously as they complement each other.

Threat	Containment	Prevention
malicious extension, disclosure	✓	×
privilege escalation, disclosure	✓	✓
privilege escalation, mis-execution	×	✓

Table 6.1: Threats and Policy Coverage.

While we are not the first to consider the threats of script injection into extensions and maliciously written extensions, prior defenses do not meet the set of practicality criteria set out above. Some have applied information flow control (IFC) to prevent the execution of scripts injected by pages into extensions, but have done so in a JavaScript interpreter, where baseline execution is already sufficiently slow that the overhead of IFC is not very significant [75]. Today's JavaScript engines, such as Google Chrome's V8, are just-in-time- (JIT-) compiled, and so fast that the overhead of interpreted IFC becomes unacceptably great (as we demonstrate experimentally in Section 6.5.3). Others have deployed IFC in an attempt to thwart information disclosure by malicious extensions, but have not tracked some implicit flows [73]. When such flows are tracked, the wide propagation of sensitivity labels may cause false-alarm detections of disclosures. When they are not tracked, however, a malicious extension author can straightforwardly "launder" sensitivity labels, and thus evade the defense utterly. Still others have investigated statically verifying that extensions do not disclose sensitive data to the network without authorization, but have required that extensions be rewritten in a new language amenable to such an analysis, since general JavaScript is not [85].

We propose ScriptPolice, a policy system for the Google Chrome browser's V8 JavaScript environment that meets all the practicality criteria we have described. To avoid the implicit flow quagmire given malicious extensions, ScriptPolice enforces a family of three containment policies using *discretionary access control (DAC)*. DAC as typically envisioned limits read access to sources of information. But such a naïve policy breaks many innocuous extensions that need to read sensitive data, but don't ever try to send to the network (see "Local model" in Section 6.3.1). It also creates thorny conflicts of interest between parties with incompatible goals, such as a page developer who wants users to see ads because they generate revenue, *vs.* a user who wants an extension to block the display of ads (see Section 6.6.2). We instead contribute a different form of DAC that is especially well suited to confining malicious extensions by controlling *write* access to *sinks* of information (in this case, the network). Finally, ScriptPolice enforces a prevention policy using a fast, JIT-compiled IFC implementation. These four policies in total are each simple, general, and ship "baked into" the browser—they cover a wide range of extensions, while allowing them to retain their full functionality.

Our contributions in this work include:

- Novel vulnerabilities in popular extensions that allow disclosure of sensitive information from web pages of any origin by maliciously constructed pages.

- Flexible source- and sink-based DAC policies tailored to the canonical behaviors of legacy browser extensions. We characterize three simple canonical behaviors of legacy browser extensions, and a

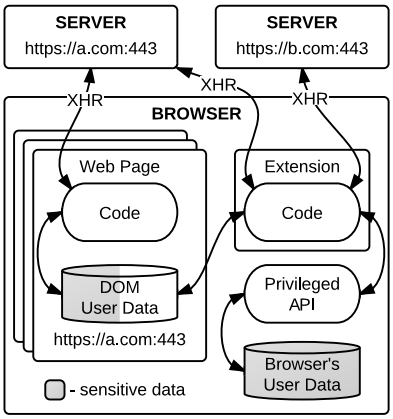


Figure 6.1: Extension Architecture.

containment policy tailored for each that applies DAC either upon reads of page data or writes to the network, as appropriate to an extension's behavior. These policies prevent leakage of sensitive data while preserving extension functionality.

- Design and implementation of a fast JIT-compiled IFC system for JavaScript and IFC-based scheme for prevention code-injection attacks. While others have applied IFC to the JavaScript interpreter, the interpreted IFC offers an average 37x per-operation execution time overhead *vs.* without IFC, JIT-compiled IFC offers an average 1.25x overhead.

- Dynamic taint specialization, an optimization that leverages JIT compilation to omit label propagation instructions (and their overhead) from code that does not process tainted data, dynamically recompiling to include such instructions only when code encounters tainted data at run time. This IFC design is the first to do full taint tracking with execution overhead *only in proportion to the volume of tainted data processed.*

- ScriptPolice, an efficient and effective browser policy system. Implemented for the Google Chrome browser's V8 JavaScript engine, ScriptPolice incorporates all the above techniques. It allows a wide range of legacy extensions to function correctly, and prevents sensitive data exfiltration and script injection, all while typically increasing page load times by 5% or less.

6.1 Background: Browser Architecture

Figure 6.1 shows the main components of a modern browser. The protocol, domain name, and port number of a URL form a displayed page's *origin.* A web page includes JavaScript code[1] and content. The code in the page accesses the page's content in object form through the *Document Object Model (DOM)* [86]. The browser also stores *cookies* set by web pages; these cookies are often used to authenticate users to web sites, and so are sensitive information [110].

The SOP isolates the scripts within a web page so that they can only read and write the DOM and cookies within the same origin. In addition, scripts in a web page may send requests and retrieve

[1]Throughout this chapter, we use "JavaScript," "script," and "code" interchangeably when doing so is not ambiguous.

responses over the network with the JavaScript *XMLHttpRequest* (XHR) object [59], but only to the same-origin server. The SOP thus isolates the code in one page from the server-side and client-side data associated with another page of different origin. In Figure 6.1, for example, scripts in the page with origin `https://a.com:443` cannot access the client-side or server-side data under the different origin `https://b.com:443`. Note that the SOP does not prevent a page's script from including foreign-origin content and disclosing data to foreign-origin servers by means other than XHR. For example, a script can set the `src` property of an image HTML element to a URL including arbitrary data, after which the browser will send that data to the destination server as it retrieves the requested image.

As also shown in Figure 6.1, a browser may further incorporate one or more extensions, installed by the user to enhance its functionality. Extensions run with higher privilege than pages. They are not bound by the SOP, and may read and write the server-side and client-side data of pages of all origins. Moreover, extensions may use a privileged API to read sensitive data managed by the browser, such as user browsing histories and bookmarks. Thus, in Figure 6.1, an extension can read page `https://a.com:443`'s DOM, cookies, and server-side data (via XHR). The page, however, may not use the privileged API to read browser-managed data.

In accordance with the principle of least privilege [122], the Google Chrome browser includes mechanisms that allow limiting the set of page origins whose data an extension may read and write, and the set of operations in the privileged API an extension may invoke [62]. At extension installation time, the browser asks the user to approve the list of permissions requested by the extension, which the extension's author specifies in a *manifest.* Extension authors are urged by Google to request only the minimal privileges an extension requires to operate. Limiting the privilege of an extension limits the potential harm should an attacker compromise an extension. However, many extensions require a broad set of privileges [62, 80], such as read access to data in pages of all origins. In such cases, Chrome's controls over privilege are typically too coarse-grained to prevent harm (*e.g.*, the disclosure of sensitive data from a page).

The Google Chrome browser additionally requires developers to *privilege-separate* [118] extensions. Privilege separation entails splitting extensions into unprivileged and privileged parts, so that as much of the extension's code as possible falls in the unprivileged part, and only this unprivileged code interacts directly with pages [62]. The intent is to minimize the potential damage if an attacker manages to construct a page that compromises an extension—if the part of the extension that interacts with the page directly is unprivileged, the attacker will hopefully not be able to escalate her level of privilege significantly. An extension's unprivileged *content scripts* execute within the pages that the extension may access, at the same privilege level as the page's own scripts. A content script may thus read the page's content and cookies via the DOM and issue XHRs to the page's origin. Unlike all other scripts within a page, the content script may communicate with the extension's privileged part via a `postMessage`-like API. The privileged part of an extension is the HTML *background page*, which may embed *core scripts* that run within the extension's privileged domain. These scripts may invoke the browser's privileged API.

Since only content scripts process page data (which may be supplied by an attacker), it is more likely that the attacker compromises a content script than a core script. The content script provides no additional privilege apart from the ability to send arbitrary messages to the core script. Privilege separation notwithstanding, compromise of an extension's core part is still possible, as we demonstrate with exploits of popular extensions in Section 6.5.1.

Throughout this work, we focus on the extension architecture of the Chrome web browser. Firefox also supports extensions, but its architecture offers less control of extensions' privileges than Chrome's.

6.2 Threat Model

The two classes of browser security policy we pursue in this work, containment and prevention, target different attacks, each with a different threat model.

There are other means of disclosing sensitive data from web pages, namely, cross-site scripting (XSS) [129], which we do not address in this work. However, in Section 6.6.3, we discuss the possibility of enforcing containment and prevention policies on web applications with ScriptPolice to thwart sensitive data disclosures caused by XSS attacks. We also do not address denial-of-service attacks by extensions and pages where an extension or a page is specifically designed to waste the system's resources such as memory, storage, network, *&c.* in order to prevent the browser from correct functioning.

6.2.1 Containment Policy Threat

Extensions may be maliciously written to read and disclose sensitive data from pages. Maliciously constructed pages may also inject code into vulnerable extensions at runtime that accomplishes such disclosures. In either case, when at installation time an extension requests access to web pages with arbitrary origins, should the user grant this request, she creates the risk of disclosure of her sensitive content, as with the extension in Figure 6.1.

We focus on the confidentiality of sensitive content within pages because the per-origin, all-or-nothing, deny-allow access control that Chrome enforces for extensions today is too coarse-grained. Pages typically contain a mix of sensitive and non-sensitive data. A user may be willing to disclose non-sensitive content to a remote party (*e.g.,* as when using an extension to look up a non-sensitive word in a page in an online dictionary). She may further be willing for an extension to process both sensitive and non-sensitive content, so long as the browser guarantees it cannot disclose sensitive content to unauthorized origins via the network.

We exclude from our containment threat model multiple colluding extensions and scripts within pages that themselves leak sensitive data from within the same page. In Section 6.6.4, we describe simple extensions to ScriptPolice that defend against these further threats.

6.2.2 Prevention Policy Threat

Despite the diverse security mechanisms in Chrome, script injection attacks by pages on extensions have succeeded in the past [99, 100, 124]. Carlini *et al.* studied one hundred Chrome extensions and found seventy vulnerabilities (many of which were script injection vulnerabilities) across forty of them [69]. We have identified novel *zero-day* vulnerabilities in several widely used extensions, and implemented

working exploits that run arbitrary script code with these extensions' elevated privileges. The vulnerable extensions in question include those for subscribing to RSS feeds, parsing open data formats, reformatting web content, and checking for email [102–105]. We have reported all vulnerabilities and exploits to the extensions' authors.

We focus on prevention of code injection attacks into vulnerable extensions via the following two attack vectors. The first one concerns a vulnerable extension processing a malicious page's content with its unprivileged content script. The content script passes data from the page to the extension's core. While the core script processes the attacker-supplied data, it triggers a vulnerability in the extension's background page. The attacker injects arbitrary JavaScript into the background page and runs it with the extension's privileges. The second attack vector involves an extension retrieving HTML and script content within its background page via HTTP. The HTTP protocol provides no integrity guarantees for transferred data. Thus, the extension may receive modified content that includes a malicious script injected by a man in the middle, which then executes with the extension's privileges. A variation of this attack is when an extension retrieves content over HTTPS but from a third party controlled by an attacker; thus, the attacker may supply malicious scripts that again execute with the extension's privilege.

There may exist other attack vectors for injecting scripts into an extension. We only address those explicitly enumerated above.

6.3 Design

We now describe the design of the policies that ScriptPolice enforces so as to defend against the threats articulated in the previous section. Before delving into those details, however, we first briefly give an overview of the setting in which ScriptPolice will be used.

We envision that the four policies we describe in this chapter will be "baked into" the user's browser. They are not tailored to individual extensions, and neither web site authors nor extension authors need concern themselves with defining policies. The three containment policies we describe are each tailored to a canonical extension behavior; we describe these canonical extension behaviors in Section 6.3.1. And the single prevention policy fits all extensions.

We further envision that extension authors will express in a manifest which privileges their extensions need, just as they must do today for Google Chrome extensions. We require one small enhancement to today's Google Chrome manifest: that extension authors state which origins an extension must communicate with. We note for now that there is no gain to be had from exaggerating the requested privileges, as will become apparent when we discuss the details of the containment policies.

Finally, we envision that web content will include markup to indicate which elements in a page are sensitive. We believe this assumption is eminently reasonable, in part because many sites already have such information at their disposal: for example, social networking and other sites today explicitly store which elements of a user's profile the user wishes to be public and which should be kept private. Overall, the developer of a site should have a good understanding of which information displayed on the site's pages is sensitive (*e.g.,* bank account balances, credit card numbers, grades on a transcript, *&c.*). In cases where the developer is unsure of the user's preferences, the user may express them in a profile.

ScriptPolice is agnostic about the form of this markup indicating which DOM elements are sensitive, so long as it is at a DOM-element granularity. (In our prototype, we define a `priv` attribute for DOM elements, where sensitive DOM elements have `priv="true"`.)

We now turn to the three canonical extension behaviors that motivate our three containment policy designs.

6.3.1 Canonical Extension Behaviors

Understanding how extensions use page data is helpful in defining containment policies intended not to interfere with an extension's functionality (provided that this functionality does not exfiltrate sensitive data, of course). Different extension behaviors suggest different policies; we formalize this mapping in Section 6.3.2.

Local model. An extension reads data from a page, processes it locally, and sends no data to the network. The FlashBlock extension [9] fits this model: it removes Flash from a page the user visits, but requires no network access.

Remote model. An extension reads content from a page and sends it to a remote server (or remote servers) for processing. The remote server must be known at the extension installation time. The Google Dictionary extension [11] fits this model: it retrieves a user-chosen word from a page, sends it to Google's server, and displays the word's definition retrieved from the server.

Promiscuous model. An extension reads content from a page, sends it to a remote server (or remote servers) unknown at extension installation time, and displays the result of the remote processing. Download Master [7] is an example of a promiscuous extension: it renders links retrieved from a page in a popup window and provides a convenient way to download content referenced by those links. When a user clicks on a link, the browser sends a GET request to a remote server and retrieves the resource referred to by the link's URL. As this URL may contain a page's data, Download Master may disclose data from that page to arbitrary remote servers.

Hybrid model. This model combines features of the Local model and either the Remote or Promiscuous model. Within the Hybrid model, an extension reads content from a page, processes it locally, and displays a result which we term the *local result*. In addition, it sends (possibly distinct) content from the page to a remote server for further processing, and then retrieves and displays a *remote result*. If the destination server is known at extension installation time, the extension's remote functionality fits the Remote model and otherwise the Promiscuous model. We have not (yet) found an extension that matches the Hybrid model. This model is of note because it admits the possibility of an extension producing both local and remote results on data read from a page. We refer to hybrid extensions when discussing limitations of our containment policies in Section 6.6.3.

6.3.2 Containment Policies

An unsuspecting user may install an extension that is either vulnerable to script injection or includes malicious code, and grant that extension permission to access one (or all) origins at installation time. In both cases, the consequence can be that an extension may read sensitive information from a page and

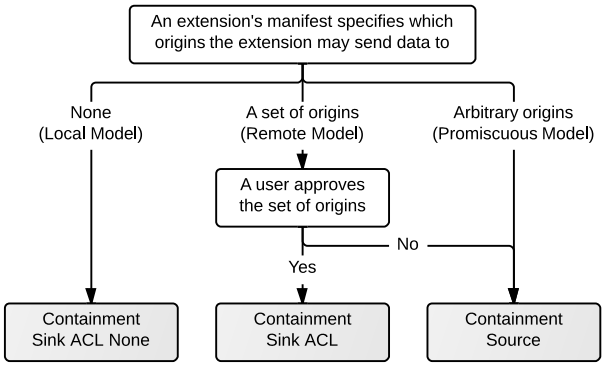


Figure 6.2: The Containment Policy Assignment Scheme.

disclose it to an unauthorized server. ScriptPolice allows the browser to enforce a containment policy that restricts such disclosure of pages' sensitive data by extensions.

We implement ScriptPolice's containment policies with Discretionary Access Control (DAC). The DAC mechanism entails making access-control decisions at sensitive data sources (pages) and network data sinks. Thus, a containment policy may prevent a script from obtaining sensitive data at the sources or restrict it from disclosing data to unauthorized origins via network sinks.

Our choice of DAC over IFC for implementation of the containment policy stems from our desire to offer a robust defense against maliciously written extensions. ScriptPolice's IFC implementation does not propagate taint across implicit data flows [71], nor do prior IFC systems, for many implicit flows [73]. Despite an IFC-based containment policy, an extension deliberately written to disclose pages' sensitive data could launder taint via implicit data flows and still disclose sensitive information to unauthorized origins. However, propagating taint along implicit flows causes all data written within a conditional statement's body to become tainted, and runs the risk of tainting data that are not in fact sensitive, and thus throwing spurious exceptions when non-sensitive data leave the browser. Note that the IFC-based Prevention policy is immune to this problem by virtue of the threat model it addresses: it targets extensions that are "honest but vulnerable," *i.e.,* are not written to launder taint.

We define two design requirements that our containment policy must fulfill. First, the policy must achieve its goal, *i.e.,* restrict disclosure of pages' sensitive data by malicious and vulnerable extensions. Second, the policy must allow extensions to do their work on sensitive and non-sensitive data unless working with sensitive data contravenes the policy's goal, *e.g.,* if an extension requires sending sensitive data to an unauthorized server. To achieve the latter requirement, we designed three containment policies, one for each of the canonical extension behaviors in Section 6.3.1: Containment Sink ACL None, Containment Sink ACL, and Containment Source.

Figure 6.2 shows the policy assignment scheme that the browser follows to choose the containment policy with which to confine an extension. An extension's manifest file enumerates to which remote origins that extension sends pages' data. If the manifest file contains no remote origins, the browser classifies the extension as a local one and applies the Containment Sink ACL None policy. If the manifest file specifies specific origins, the browser identifies a remote extension, and consults the user to learn

whether she approves sending sensitive information from pages to these origins. With the user's consent, the browser enforces the Containment Sink ACL policy, automatically customized to allow releasing data only to the approved origins. Otherwise, the browser enforces the Containment Source policy. Finally, an extension may require sharing page data with origins only determined at runtime, *i.e.,* as declared at extension installation time, with arbitrary origins. In this case, the browser classifies the extension as promiscuous and confines it with the Containment Source policy. Note that policy selection and the described automated customization take place once at an extension's installation, and the chosen policy is enforced when the extension executes.

Containment Sink ACL None. The Containment Sink ACL None policy restricts extensions from sending *any* information to remote servers. It thus denies disclosure of sensitive information that extensions may read from pages. The restriction on network communication prevents extensions from performing remote functionality; therefore, this policy is enforced *only* on local extensions which do not need the network.

Containment Sink ACL. Containment Sink ACL is a generalization of the Containment Sink ACL None policy. It targets remote extensions. As shown in the policy assignment scheme in Figure 6.2, the browser enforces Containment Sink ACL if a user authorizes sharing sensitive information with the origins specified in an extension's manifest file. The browser automatically customizes the default Containment Sink ACL policy to allow information to flow to the authorized origins. Thus, the customized policy allows an extension to read sensitive and non-sensitive information from pages and send it to user-authorized remote servers. However, the policy restricts the extensions from disclosing any information to unauthorized origins. The extension may perform local and remote functionality on sensitive and non-sensitive data.

An extension may wish to include third-party content via *static* links, *e.g.,* AdBlock Plus retrieves a list of ad filters from a remote server using a static URL. In this case, we require the extension to declare all static URLs for remote resources in its manifest file. When a user installs such an extension, the browser tailors the Containment Sink ACL or Containment Sink ACL None policy to allow these URLs. With the customized policy, the extension may use the declared URLs to retrieve remote resources, but it cannot disclose sensitive data by embedding such data into the URLs. We describe below how we prevent leaks by "modulating" information over requests for static URLs. Note that users need not authorize static links.

Containment Sink ACL and Containment Sink ACL None impose no restrictions on access to pages' data, and an extension confined with one of these policies may perform its function on sensitive and non-sensitive data alike. Thus, these policies do not interfere with the functionality of local and remote extensions. The policies require no modification to pages, and only Containment Sink ACL requires user authorization of remote origins.

Both Containment Sink ACL policies allow potentially malicious code to handle sensitive data, and so risk disclosure of sensitive information via covert channels [92]. We block one important such channel: we prevent an extension from encoding page data covertly by requesting static URLs in a

"modulated" pattern. To do so, we prefetch *all* of an extension's manifest-declared static URLs before the first execution of the extension's content script for each page, and force all subsequent requests for static URLs during the extension's execution to be served out of cache. We leave consideration of more esoteric covert channels for future work.

Containment Source. Our last policy is Containment Source. It denies an extension read access to sensitive data in a page. With the ScriptPolice interposition mechanism described in Section 6.4.1, the policy disables all operations on DOM elements containing sensitive data. Thus, a malicious extension cannot read sensitive data and disclose it to the network. The Containment Source policy is enforced on all promiscuous extensions and on remote extensions whose remote origins the user does not authorize to obtain sensitive information from pages. Because the policy denies access to sensitive data, it limits extension functionality on this class of data. For example, the Google Dictionary remote extension confined with the Containment Source policy displays definitions for words marked as non-sensitive and shows nothing for sensitive content.

The Containment Source policy must distinguish between sensitive and non-sensitive content in a page. To do so, it leverages the annotations described at the start of Section 6.3.

The Containment Source policy imposes no restrictions on an extension's network communication; it simply denies read access to sensitive data. Unlike the Containment Sink ACL policies, Containment Source is immune to sensitive information disclosure via covert channels.

6.3.3 Prevention Policy

Recall that our aim in the prevention policy is to protect extensions from pages' malicious injection of scripts that cause disclosure of sensitive data.

We implement the Prevention policy atop a fine-grained taint tracking [113] information flow control (IFC) mechanism that we have designed and built for JavaScript. Our IFC implementation entails tagging run-time data to indicate when the data are from a suspect source from an extension's point of view. Such data are *tainted.* At every operation (*e.g.,* arithmetic, copy, string concatenation, *&c.*), taint propagates in the following way: if any operand is tainted, the result becomes tainted. The policy may decide to restrict certain operations if they involve tainted data, *e.g.,* as with injection of code derived from data from pages.

The Prevention policy restricts *unauthorized* script injections into an extension's privileged execution environment. It taints data from pages and the network at the appropriate sources, as they may provide malicious input. It then blocks the injection of code derived from tainted data into the extension's background page. To do so, the Prevention policy monitors operations that allow dynamically adding new code in the background page. If the extension attempts to introduce new code via one of these operations, and the string in which the code is embedded is tainted, the policy throws an exception.

6.4 Implementation

We now describe the implementation of ScriptPolice, a system for the Google Chrome browser that interposes on interactions of JavaScript code with the rest of the browser to enforce security policies on

Data Sources

```
data = html_element.innerText; // text from DOM
data = document.cookie; // cookie
data = window.getSelection(); // cursor selection
data = window.location.href; // document URL
data = xhr_object.responseText; // XHR data
```

Data Sinks

```
html_element.href = "http://server.com/data";
html_element.src = "http://server.com/data";
xhr_object.open("GET", "data", false);
```

Code Sources

```
html_element.appendChild(script_element);
html_element.onclick = function(){ code };
html_element.href = "javascript:code";
html_element.src = "javascript:code";
```

HTML Sources

```
html_element.innerHTML = "new HTML";
html_element.appendChild(new_element);
html_element.replaceChild(element, new_element);
document.write("new HTML");
```

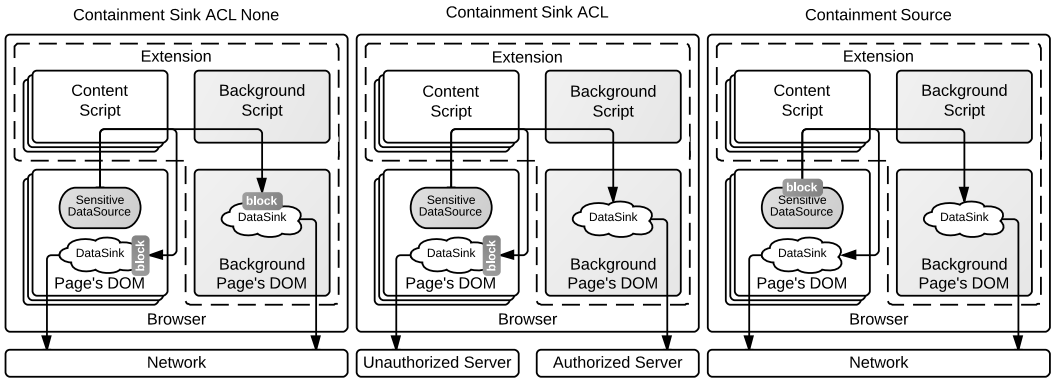
Figure 6.3: Examples of Sources and Sinks.

extensions. We begin by reviewing the interface between the JavaScript execution environment and the rest of the browser. We then describe the implementation of our containment and prevention policies. We also demonstrate how policies are specified in ScriptPolice, and we conclude this section with two optimizations that improve performance of ScriptPolice's IFC mechanism.

6.4.1 Interposition Interface and Policy Implementation

When an extension invokes operations implemented in the browser, *e.g.,* to read or write the DOM or send information to the network, ScriptPolice intercepts the call and takes action according to any policies in effect. Just as an operating system's system call interface is a natural "choke point" at which to enforce policies, as done in systrace [117] and SELinux [98], so is the well-defined interface between JavaScript and the rest of the browser. On invocation of a DOM API call, ScriptPolice invokes any JS function(s) bound to that DOM API call by a policy. Such policy functions may examine an API operation and its arguments and then make a decision on whether to allow or deny the operation.

The interaction between an extension's JavaScript code and the rest of the browser occurs through JavaScript objects that comprise a well defined JavaScript-browser API. This API includes JavaScript objects that allow access to the DOM (*i.e.,* content) of pages [86]; the XHR JavaScript object, which allows access to the network [59]; JavaScript objects that allow access to locally stored cookies [110]; and in the case only of JavaScript code in extensions' background pages, the JavaScript objects that comprise the privileged extension API [84]. Recall from Section 6.1 that all these objects are bound by the SOP, *apart* from those in the privileged extension API.



Shaded elements denote the extension's privileged components.

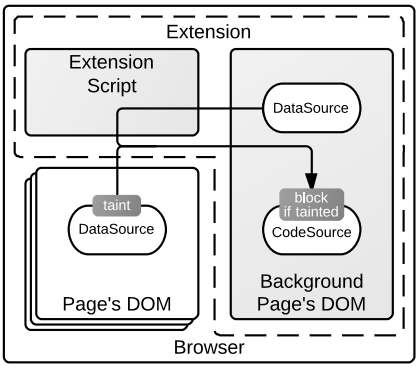
Figure 6.4: The Containment Policies.

We term the objects through which data enters the JavaScript execution environment *data sources*, and the objects through which data exits it *data sinks*. Sensitive data may arrive through data sources and be disclosed through data sinks. Data sinks that cause information to flow to the network are of particular note. For example, a script may convey information to the network by issuing requests for resources, *e.g.,* an image, frame, or script. The browser sends the data embedded in these requests' URLs to the specified destination servers. Alternatively, the code may send data with XHR objects. Figure 6.3 shows examples of data sources and sinks.

A hallmark of the JavaScript execution environment is that scripts generate further script code at runtime. Such dynamically generated JavaScript code enters the execution environment via a well-defined set of JavaScript objects. We term objects through which code enters the environment *code sources*, exemplified in Figure 6.3. For example, appending a Script HTML element (`script_element` in Figure 6.3) to the DOM with `appendChild()` causes execution of that script. And attaching an event handler to a DOM element (*e.g.,* an `onclick` handler) causes the specified code to execute when the specified event is triggered. Finally, one may embed scripts in URLs by using the `javascript:` protocol. When the browser retrieves a resource whose URL takes this form, it executes the script embedded in the URL, rather than issuing a GET request.

Finally, JavaScript code may also introduce new HTML elements at runtime, which may in turn contain code sources or data sinks. We term the JavaScript objects that allow adding HTML to a page's DOM *HTML sources*. Note that to enforce policies correctly, ScriptPolice must interpose on all code sources, data sources, and data sinks—including those introduced dynamically by HTML sources.

Figure 6.4 shows the implementation of the Containment Sink ACL None, Containment ACL, and Containment Source policies confining an extension. In the case of Containment Sink ACL None, the policy restricts the extension from releasing any information to network data sinks from the page's DOM or the DOM of the extension's background page. Similarly, the Containment Sink ACL policy blocks sending any information through network data sinks if the destination origin is not one of the user-approved origins. The Containment Source policy monitors data sources corresponding to DOM elements marked as sensitive (as described below) and denies any operations on them for the extension's



Shaded elements denote the extension's privileged components.

Figure 6.5: The Prevention Policy.

content script. Thus, the extension cannot read sensitive data. Finally, Figure 6.5 shows the Prevention policy which taints any data read from the page, and blocks all attempts to supply the tainted data to a code source of the privileged background page at the risk of executing page-supplied scripts. We describe how policies are specified in ScriptPolice in Section 6.4.2.

The Containment Source policy must be able to identify DOM elements that contain sensitive data. We rely on the web developer to specify which HTML elements in pages contain sensitive data. To do so, the developer simply wraps sensitive content in an HTML container element with the attribute `priv="true"`. Our policies identify sensitive elements by checking if an element's or its parent's `priv` attribute is set to `true`. If an adversary attempts in an injected script to change the `priv` attribute so as to mislabel sensitive content, ScriptPolice will always examine the *unmodified* object first (with the original value of the `priv` attribute), and note that it contains sensitive data before making any changes to the attribute.

6.4.2 Expressing Policies

We now briefly illustrate how ScriptPolice policies are specified. A policy is a simple enumeration of the *policy atoms* that must be applied to an extension's scripts. A policy atom pairs a *monitoring function* with corresponding *binding code*. Binding code inspects JavaScript objects and matches those to which the associated monitoring function should be bound. At every operation on a source or sink object, ScriptPolice invokes the monitoring functions attached to that object (if any). A monitoring function may examine the source or sink, the requested operation on it, and the operation's arguments. Based on this information, the monitoring function decides on one of several courses of action. It may throw an exception to signal that an unauthorized operation has been attempted. Alternatively, it may silently skip execution of the operation and allow the extension to continue execution. Finally, it may ask ScriptPolice to taint the operation's result. The former two decisions allow policies to deny operations on sources and sinks. For example, the Prevention policy denies code injection from code sources, and the Containment Source policy denies read access to data from sensitive data sources—those annotated as such in a page, and marked as such in the page's DOM.

We have built a library of simple policy atoms, each with a mnemonic name. We have further implemented a simple policy construction tool. This tool allows the developer simply to enumerate policy atoms of her choice by name in a *policy file.* The policy specified by such a file is the union of all its policy atoms. The tool reads a policy file and expands the mnemonic names for policy atoms into their corresponding monitoring functions and binding code, yielding a single JavaScript file containing the full policy.

Let us consider as an example a fragment of the policy file that describes our Prevention policy:

```
targets = [ ...
  "js_taintblock_src",
  "js_taintblock_href",
  "js_taintblock_append_replace_child",
  "js_taintblock_inner_html",
  "js_taintblock_write",
  ...  ]
```

We proceed top-down, first describing its policy atoms, then exploring the policy atoms' monitoring functions and binding code. Each line is a policy atom. `src`[2] and `href` atoms prevent unauthorized code injection by blocking the assignment of tainted URLs that use the `javascript:` protocol to `src` and `href` properties of DOM elements, respectively. The `append_replace_child`, `inner_html`, and `write` policy atoms monitor HTML sources to ensure that any HTML elements added to the DOM do not introduce inlined code via event handlers or `src` or `href` properties.

Consider the binding code for the `src` policy atom. This code binds the policy atom's monitoring function to all DOM objects with `src` properties:

```
function HandleAll(obj) {
  if ("src" in obj)
    AddFunction(obj, js_taintblock_src);
}
```

Finally, consider monitoring function for the `js_taintblock_src` policy atom:

```
function js_taintblock_src(op, obj, name, val) {
  if (op == op_set && name == "src")
    if (IsTainted(val))
      if (val.indexOf("javascript:") == 0)
        return throw_action;
}
```

This short fragment of JavaScript code checks whether the value assigned to the `src` property of a DOM element is tainted; if so, it has been derived from a page source (in accordance with the rest of the Prevention policy). If the value is tainted *and* is a URL that begins with `javascript:`, the monitoring

[2]To enhance readability, we omit the `js_taintblock_` prefix at the start of policy atom names.

function blocks the assignment, as it may inject malicious code into an extension's privileged execution environment.

6.4.3 IFC Performance Optimizations

Two principal optimizations improve the performance of our IFC implementation: generating native (non-interpreted) code for taint tracking, and specialized taint tracking.

Native Taint Operations. In our initial implementation of IFC for V8, we only implemented taint tracking in V8's "slow path" interpreter. This approach only required simple modifications to the V8 runtime. However, V8's speed at executing JavaScript comes mostly from its just-in-time (JIT) compilation of JavaScript into native code. Forcing execution of operations on tainted data onto the interpreted slow path hampers performance severely—by a factor of 20x–78x as compared with native, JIT-compiled code, as we evaluate in Section 6.5.3.

We therefore enhanced our IFC implementation for V8 to emit just-in-time compiled native code stubs for operations on tainted data types. These native code stubs check if an operation's arguments are tainted, and if so, they taint the operation's result. Moving to native JIT-compiled code improved the performance of tainted operations significantly, as we evaluate in Section 6.5.3.

Specialized Taint Tracking. The V8 JIT compiler enables *type specialization,* an optimization that has been shown to offer greater than 10x speedup [83]. We implement the specialization of JIT-compiled code to *non-tainted* data types, so as to reduce the execution overhead associated with taint propagation when processing non-tainted data. When the V8 runtime emits a native stub to perform an operation, *e.g.,* an arithmetic operation, it analyzes the supplied arguments. If the arguments are not tainted, the runtime generates code specialized for a non-tainted data type which lacks the taint checks on arguments and taint propagation code. This stub's execution incurs no taint propagation overhead when processing non-tainted data. Later, if a tainted argument is supplied to this stub, V8 recompiles the stub and adds the taint check and taint propagation code. Thereafter, the stub may process the tainted data type as well as the non-tainted one; however, in the latter case, the stub incurs the unnecessary overhead of the taint checks on arguments.

Our optimization attempts to specialize for a non-tainted data type first when emitting a native code stub for an operation, and then falls back to the general (tainted and non-tainted) stub if processing of a tainted data type is required. Note that type specialization is enabled and performed independently from specialized taint tracking. We demonstrate the benefit of this optimization for non-tainted data types in Section 6.5.3.

For comparison, TaintCheck [113], a prior taint tracking system that enables dynamic taint analysis in unmodified binaries, introduces taint propagation code at every memory access, regardless of whether the instruction operates on tainted data. As a result, TaintCheck incurs execution time of as much as 24x that of native execution, even when no tainted data are processed by an application.

Extension	Model	Containment Policy
Adblock Plus (8057K)	Local	Sink ACL None
Answers (9K)	Remote	Sink ACL or Source
Checker Plus (356K)	Remote	N/A*
Download Master (271K)	Promiscuous	Source
FlashBlock (532K)	Local	Sink ACL None
Google Dictionary (1500K)	Remote	Sink ACL or Source
Microformats (10K)	Promiscuous	Source
RSS Feed Reader (631K)	Promiscuous	Source
RSS Live Links (32K)	Promiscuous	Source
RSS Subscription (849K)	Remote	Sink ACL or Source
Skype Links (43K)	Local	Sink ACL None
ezLinkPreview (13K)	Remote	Sink ACL or Source

The number of downloads for each extension is shown in parentheses. *Checker Plus for GMail requires communicating with `google.com` to retrieve the user's email and needs no access to pages. Page access is thus denied by Google Chrome's manifest mechanism.

Table 6.2: The Tested Extensions, Their Models, and Matching Containment Policies.

6.5 Evaluation

We now evaluate the efficacy of the containment and prevention policies as implemented in the ScriptPolice system, and how enforcement of the policies impacts extension functionality and performance.

Constraining an extension's execution within a policy introduces execution overhead. As users are generally unwilling to sacrifice much in the way of performance to enjoy enhanced privacy, it's important to assess to what extent such overheads increase the latency experienced by browser users. To answer this question, we compare end-to-end page load times with extensions running without ScriptPolice *vs.* those for identical scenarios in which extensions are constrained by the various ScriptPolice policies. We further examine the slowdown in extension execution time alone incurred by ScriptPolice policies.

Finally, we use microbenchmarks to ascribe execution time costs to the various components of ScriptPolice, and to assess the performance benefits of the native-code IFC and specialized taint tracking optimizations.

Table 6.2 lists the twelve extensions from the Google Chrome store that we used to evaluate ScriptPolice, including the canonical behavior models they fit and the containment policies accordingly applied to them based on the assignment scheme in Figure 6.2. Recall that at install time for Remote extensions, the browser consults the user about the origins the extension communicates with. We chose this set of extensions based on extension popularity and so as to span the full range of canonical behavior models; we also include somewhat less popular extensions that we found to be vulnerable to code injection attacks.

All extensions in Table 6.2 apart from Checker Plus for GMail include in their manifests a request for permission to access content in pages from all origins. Thus, throughout this evaluation, we presume they may obtain sensitive data from all pages. In fact, while testing the Microformats extension with ScriptPolice and the Containment Sink ACL None policy, we were surprised to observe that as the extension collects VCards from visited pages, it secretly sends them to a remote server; it was the exceptions thrown by ScriptPolice that alerted us to this behavior. We note that other extensions with

such broad page access privileges may exfiltrate sensitive information, such as users' login credentials and passwords.

A brief summary of our salient findings:

- The containment policies (which are DAC-based, and thus relatively simple to reason about), are correct by construction. They preserve extension functionality for the twelve extensions we have evaluated, except in cases where there is a risk of disclosure of sensitive data, where they deny extension functionality to protect the sensitive data.
- The prevention policy correctly thwarts script injection attacks on all four of the zero-day extension injection vulnerabilities we identified, incurs no false positives on any of the other eight extensions we tested it with, and preserves extension functionality for the twelve extensions we have tested.
- Policy enforcement with ScriptPolice typically incurs less than a 5% increase in page load time, virtually imperceptible to users.
- While interpreted IFC incurs an average 37x per-operation execution time overhead, native-code, JIT'ed IFC incurs an average 1.25x per-operation execution time overhead.
- Specialized taint tracking reduces the execution time of operations on non-tainted data by 7% on average.
- The execution overhead of interposing on sinks and sources (interpreted operations in our current implementation) dwarfs that of label propagation for IFC (native-code operations).

6.5.1 Policy Correctness and Extension Functionality

We now consider qualitative measures of ScriptPolice's efficacy. Do the containment and prevention policies achieve their respective goals of preventing exfiltration of sensitive information and blocking injection? And do extensions continue to function correctly under ScriptPolice's policies?

Containment Policies. By construction, it's clear the Sink ACL None and Sink ACL policies prevent unauthorized disclosure of sensitive information. Sink ACL None allows no network writes by the confined extension, and Sink ACL only allows network writes to origins explicitly approved by the user. Promiscuous extensions are confined by the Source policy, and if an extension can't read sensitive data, it can't leak it.

Extensions that follow Local behavior are confined with Sink ACL None, can read all sensitive and non-sensitive data, by definition do not engage in sending to the network, and are confined to disallow them from doing so. Thus, by construction, they function correctly. Similarly, extensions that follow Remote behavior whose destination origins are approved by the user also can read all sensitive and non-sensitive data, and are confined by the policy only to send data to user-approved origins. Thus, they too by construction function correctly. Extensions that follow promiscuous behavior, or Remote behavior, but whose specified destination origins are not approved by the user, are confined with the Source policy. While the Source policy blocks extension functionality on sensitive data, this result seems fundamentally unavoidable if the policy is to prevent disclosure of sensitive data to unauthorized parties.

Extension Model	Policy	Functionality	
		Nonsensitive	Sensitive
Local	Sink ACL None	✓	✓
Remote	Sink ACL	✓	✓
Remote	Source	✓	×
Promiscuous	Source	✓	×

Table 6.3: Extension Functionality under Containment Policies as Determined by Extension Models.

Throughout our evaluation of ScriptPolice, all of the extensions in Table 6.2 behaved consistently with the above for their respective containment policies.

Table 6.3 summarizes the functionality retained by extensions for varied containment policies and extension models. Note that an extension that is denied read access to sensitive data does not stop executing; it produces no result for those data and continues processing available non-sensitive data.

Prevention Policy. We studied ten extensions that process data from pages within their privileged execution environments, and identified a number of zero-day vulnerabilities in four of them: RSS Subscription, Microformats, Easy Reader, and Checker Plus for GMail [102–105].

The first two extensions allow script injections into the extensions' background pages via `href` properties of HTML anchor elements. Furthermore, Microformats allows injection of arbitrary `<script>` elements and elements with event handlers, because it inadequately screens page-supplied HTML data.

Easy Reader embeds data from RSS feeds supplied by the Google Reader web service into the extension's background page without content filtering. This behavior makes it possible to inject inline scripts into the `href` properties of anchor elements. Google Reader could use this vulnerability to escape the browser's SOP and access browser-managed user data (users' bookmarks and history) and data in other tabs. What is even more dangerous is that the extension retrieves feeds via HTTP, which renders it vulnerable to a network attacker modifying content retrieved by the extension.

Checker Plus for GMail retrieves emails over HTTPS and displays them in an extension-privileged pop-up window. The extension does not attempt to filter out any scripts that may be embedded into the emails; thus, GMail can escape the SOP.

Again, all extensions in Table 6.2 with the exception of Checker Plus for GMail may access data from all origins. In addition, all these extensions are granted access to the *tabs* API [84] which by itself is sufficient to disclose sensitive data from any page. The tabs API allows an extension to inject and execute arbitrary scripts within the browser's tabs. For each vulnerable extension, we constructed an exploit that discloses an authentication cookie from the GMail application concurrently running in a separate tab.

Our Prevention policy blocks all attacks identified in these vulnerable extensions. Thus, for this initial test set of extensions, the policy correctly achieves its goal. We applied the Prevention policy to all extensions in Table 6.2 and confirmed that they function correctly. None of the evaluated extensions under the Prevention policy produced a false positive result, (*i.e.,* detecting an attack when there is none).

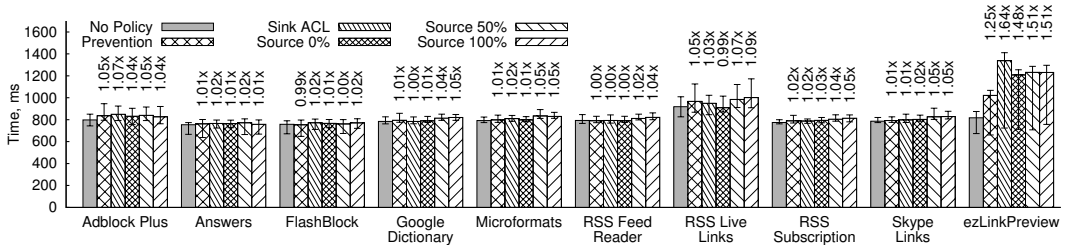


Figure 6.6: Page Load Time of youtube.com and ScriptPolice Overhead.

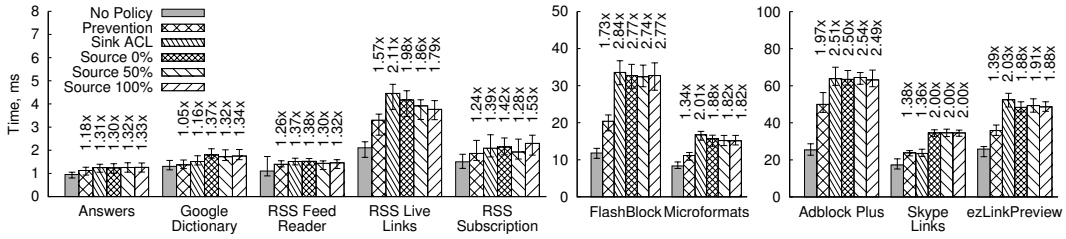


Figure 6.7: Extension Execution Time when Loading youtube.com and ScriptPolice Overhead.

6.5.2 Execution Overhead

We first evaluate the effect of ScriptPolice's policies on page load time and extension execution time. We use a range of extensions, shown in Table 6.2, and the `youtube.com` page. We consider results for `youtube.com` here as `youtube.com` is a frequently vistied site with rich content, including ads and Flash, which trigger filtering work by AdBlock Plus and FlashBlock. Further, because `youtube.com` is well provisioned, the time required to transfer content over the network from it is relatively low-variance across runs. We further present concise results of page load performance for 9 Alexa top 100 sites in the end of this section.

As page load time is an important response-time metric for users, we examine how much delay ScriptPolice policies add to page load time. To capture the page load delay that our system may impose, we measure the time required for the browser to load a page's static content and execute an extension's content scripts. We start the timer when the browser issues an initial request for a page and stop it on the latter of the following two events: completion of loading of the page's static content (the `window.load` event) or the completion of the execution of the last extension content script.

In the following graphs, each bar is the median of one hundred runs, and error bars show the 25th and 75th percentiles. We indicate slowdown relative to the baseline test case (labeled with "No Policy") just above each bars. "Source X" denotes Containment Source applied to a page, X% of whose DOM text elements are sensitive. The policy marks random text DOM elements as sensitive before scripts execute. When Containment Sink ACL is applied to a promiscuous extension, the policy allows all network communication so the extension may perform its task. We exclude Checker Plus for GMail and Download Master from some graphs because they do not execute any scripts at page load time.

Figure 6.6 shows the time required to load the `youtube.com` page and execute an evaluated extension under the range of ScriptPolice policies. The general trend in the results is that the increase

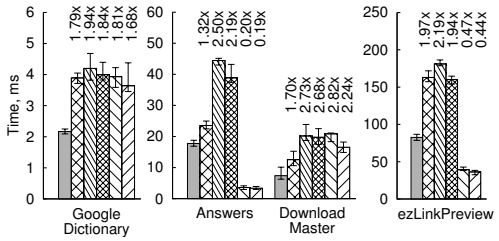


Figure 6.8: Extension Execution Time when Handling User Actions (median of ten runs) and ScriptPolice Overhead.

in page load times due to ScriptPolice tends to be short enough as to be imperceptible to users. The extension execution overhead added by ScriptPolice is insignificant against the time required to retrieve all objects in a page and render them. Indeed, the 25th–75th percentile regions overlap for the baseline and ScriptPolice results across all extensions. In some instances, median page load times appear shorter when ScriptPolice is used than when it is not. These results reflect the variability in network transfer time and server response time when retrieving pages. Our examination of extension execution time alone below reveals that the magnitude of the variability indicated by the 25th–75th percentile regions in these end-to-end page load times is greater than the extension execution overhead added by ScriptPolice.

Now let us consider ScriptPolice's impact on extension execution time alone. Figure 6.7 shows the time required for an extension to execute its content scripts when loading `youtube.com`. For most extensions, the slowdowns are acceptable and below 2.0x; however, FlashBlock demonstrates the greatest slowdown of 2.8x when the extension executes under the Containment Sink ACL policy. Note that `youtube.com` includes Flash and thus is a typical workload for the FlashBlock extension. Although the relative overheads seem to be high for AdBlock Plus, FlashBlock, RSS Live Links, Skype Links, and ezLinkPreview, the absolute overheads suggest that our system adds no more than 40 ms of additional extension execution time—again, insignificant for typical users.

To evaluate extensions' responsiveness, we further measure the extension execution time required to handle a user action. Thus, for Answers and Google Dictionary, we obtain and display a definition of a user-selected word. Download Master collects HTTP links from a page and displays them in a popup window. ezLinkPreview renders a preview window for a selected link. Figure 6.8 shows the execution overhead imposed by ScriptPolice's policies when extensions handle user actions. One may notice that the Answers and ezLinkPreview extensions confined with Containment Source 50% and Containment Source 100% perform significantly better than when these extensions are not confined with any policy. These policies deny access to sensitive data, and therefore extensions do almost no work when such access is denied. The greatest relative overhead of 2.8x occurs for Download Master confined with Containment Source 50%, and the greatest absolute overhead of 100 ms occurs for ezLinkPriveiew confined with Containment Sink ACL. The rest of the extensions incur less than 30 ms of additional execution time. A slowdown of 100 ms would be perceptible by users. However, it is not noticeable in this case because ezLinkPreview's execution overlaps with other delays, such as for content retrieval over the network. In particular, it retrieves a resource referenced by the link to display it in a popup window,

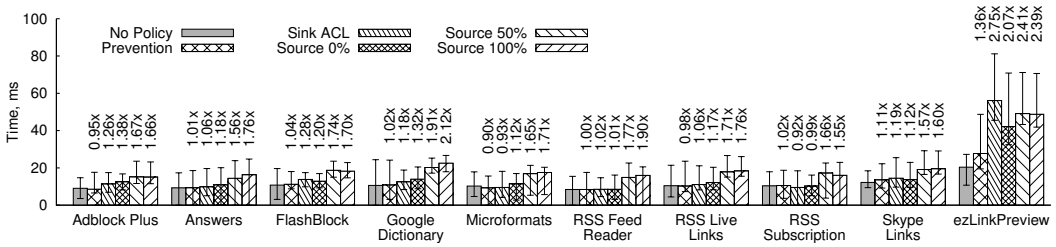


Figure 6.9: Garbage Collection Time when Loading youtube.com and ScriptPolice Overhead.

Extension	Page Load Time					Extension Execution					Garbage Collection				
	Prev	**Sink**	**Src0**	**Src50**	**Src100**	**Prev**	**Sink**	**Src0**	**Src50**	**Src100**	**Prev**	**Sink**	**Src0**	**Src50**	**Src100**
Adblock Plus	1.03x	1.06x	1.02x	1.01x	1.03x	1.66x	1.89x	1.80x	1.80x	1.80x	1.14x	1.19x	1.21x	1.38x	1.42x
Answers	0.96x	0.96x	0.95x	0.95x	1.01x	1.26x	1.33x	1.38x	1.39x	1.40x	1.04x	1.05x	1.06x	1.46x	1.45x
FlashBlock	1.01x	1.01x	1.02x	1.02x	1.03x	1.84x	2.30x	2.27x	2.29x	2.29x	1.29x	1.29x	1.32x	1.71x	1.67x
Google Dictionary	0.99x	0.98x	0.98x	0.98x	1.04x	0.99x	1.04x	1.21x	1.18x	1.17x	1.21x	1.24x	1.25x	1.58x	1.53x
Microformats	1.05x	1.00x	1.06x	1.06x	1.01x	1.30x	1.84x	1.67x	1.66x	1.66x	1.05x	1.09x	1.15x	1.32x	1.32x
RSS Feed Reader	1.02x	1.02x	1.00x	1.00x	1.01x	1.06x	1.11x	1.06x	1.09x	1.09x	1.01x	1.01x	1.00x	1.33x	1.32x
RSS Live Links	1.02x	1.00x	0.97x	1.02x	1.00x	1.31x	1.53x	1.40x	1.43x	1.44x	1.16x	1.14x	1.12x	1.47x	1.49x
RSS Subscription	1.02x	1.01x	1.02x	1.09x	1.03x	1.28x	1.41x	1.37x	1.40x	1.38x	1.03x	1.04x	1.01x	1.43x	1.41x
Skype Links	1.02x	1.00x	1.03x	1.03x	1.03x	1.31x	1.23x	1.73x	1.72x	1.74x	1.17x	1.16x	1.15x	1.45x	1.50x
ezLinkPreview	1.07x	1.12x	1.05x	1.14x	1.11x	1.61x	1.86x	1.72x	1.70x	1.69x	1.65x	2.39x	1.99x	2.32x	2.27x

Numbers are geometric means of overheads on pages from our evaluation set. **Prev** – the Prevention policy; **Sink** – Containment ACL Sink; **SrcX** – the Containment Source policy and a test page with X% DOM elements labeled as sensitive.

Table 6.4: ScriptPolice Overheads on Page Load Time, Extension Execution Time, and Garbage Collection Time for Selected Pages.

and the network and server delays associated with retrieving the remote resource mask the execution overhead imposed by ScriptPolice's policies.

Our system increases extension execution time which in tern increases the time spent in garbage collection. Figure 6.9 shows the garbage collection time for all scripts,[3] page scripts and extension scripts, when loading the youtube.com page. Non-deterministic nature of garbage collection makes the numbers highly variable as demonstrated by the 25th and 75th percentiles. For almost all cases, the ScriptPolice garbage collection overhead is between 1.0x and 1.5x which is no more than 10 ms of additional garbage collection time in the worst case. The garbage collection numbers for Containment Source 50% and Containment Source 100% are exaggerated, as they include artificial garbage collections caused by the policy's marking sensitive DOM elements, as described in the beginning of this section.

To eliminate any phenomena idiosyncratic to `youtube.com`, we present the performance overhead evaluation on a set of web pages randomly selected from the top twenty sites listed by Alexa [2]: `amazon.com, baidu.com, ebay.com, msn.com, twitter.com, wikipedia.com, yahoo.com, yandex.ru, and youtube.com.`

Table 6.4 shows the geometric mean of the duration by which policies lengthen page load time, extension execution time, and garbage collection time across the selected pages listed above, where each overhead is a mean of a hundred runs. Our policies typically make only an indistinguishable increase in pages' load times. These results are broadly consistent with those for `youtube.com`. Finally, our system lengthens garbage collection time by 1.3x in the worst case. The garbage collection

[3]The tool used for measuring garbage collection does not distinguish garbage collection caused by the extension's scripts and page's scripts.

Operation	Vanilla No Taint	No JIT Taint	JIT Taint
Add-Int	102.99	4.56 (23x)	78.41 (1.31x)
Sub-Int	103.85	4.74 (22x)	78.66 (1.32x)
Mul-Int	101.35	4.80 (21x)	76.70 (1.32x)
Div-Int	78.86	4.68 (17x)	64.47 (1.22x)
Add-Float	89.04	4.52 (20x)	65.30 (1.36x)
Sub-Float	88.87	4.34 (21x)	65.49 (1.36x)
Mul-Float	89.02	4.36 (20x)	65.62 (1.36x)
Div-Float	86.37	4.25 (20x)	64.44 (1.34x)
Add-Str	59.06	4.70 (13x)	51.30 (1.15x)
Load-Int	111.80	1.49 (75x)	86.95 (1.29x)
Load-Obj	112.41	1.64 (69x)	95.08 (1.18x)
Store-Int	94.20	1.20 (78x)	94.71 (0.99x)
Store-Obj	97.41	1.18 (83x)	92.00 (1.06x)

Numbers are operations/μs with relative slowdowns in parentheses. **Taint/No Taint** – operations on tainted/non-tainted data, **JIT/No JIT** – JITing/interpreted operations (native optimization is enabled/disabled). **Load/Store** loads/stores a type specified in the operation's name from/to a JSObject.

Table 6.5: Execution Overhead of Operations on Tainted Data and Evaluation of the Native Taint Optimization.

numbers for Containment Source 50% and Containment Source 100% are exaggerated, as they include artificial garbage collections caused by the policy's marking sensitive DOM elements, as described in Section 6.5.2.

6.5.3 Microbenchmarks and Optimizations

ScriptPolice incorporates two mechanisms for policy enforcement: the interposition interface and IFC. Below, we evaluate the distinct costs of these mechanisms. We first consider IFC, as used by the Prevention policy to track the propagation of data derived from pages within extensions.

We benchmark the performance of various arithmetic, string, and JSObject operations on tainted data *vs.* the performance of these operations on non-tainted data. Table 6.5 shows these measurements in operations per microsecond. The benchmark results presented in Table 6.5 are geometric means of ten runs, and the standard deviations among the runs are below 5%.

First, consider the performance of our initial IFC-enabled V8 implementation that did not JIT operations on tainted data (Section 6.4.3) but interpreted them in the V8 runtime, as shown in the *No JIT* column in Table 6.5. Interpreting operations is extremely slow, incurring 37x overhead on average, as for every operation execution must enter the V8 runtime. Crossing the boundary between native, JITed code and the V8 runtime for each operation results in a drastic degradation of performance. Emitting JIT'ed native code for operations on tainted data significantly improves performance; the average slowdown drops from 37x to 1.25x, as shown in the *No JIT vs. JIT* columns.

The *JIT* column in Table 6.5 shows the costs of operations on tainted data relative to those of same operations on non-tainted data, shown in the *Vanilla* column. To propagate taint, ScriptPolice must emit additional label propagation code before and after each operation that processes tainted data. The additional code untaints arguments, thus allowing an operation to proceed with non-tainted data, and

Operation	Spec No Taint	No Spec No Taint
Add-Int	102.99	99.84 (1.03x)
Sub-Int	103.85	99.92 (1.04x)
Mul-Int	101.35	95.00 (1.07x)
Div-Int	78.86	77.63 (1.02x)
Add-Float	89.04	76.29 (1.17x)
Sub-Float	88.87	76.18 (1.17x)
Mul-Float	89.02	76.25 (1.17x)
Div-Float	86.37	75.57 (1.14x)
Add-Str	59.06	56.57 (1.04x)
Load-Int	111.80	106.77 (1.05x)
Load-Obj	112.41	107.14 (1.05x)
Store-Int	94.20	99.26 (0.95x)
Store-Obj	97.41	96.83 (1.01x)

Numbers are operations/μs with relative slowdowns in parentheses. **No Taint** – operations on non-tainted data, **Spec/No Spec** – the specialized taint tracking optimization is enabled/disabled. **Load/Store** loads/stores a type specified in the operation's name from/to a JSObject.

Table 6.6: Evaluation of the Specialized Taint Tracking Optimization.

Operation	Vanilla	Interpret	Policy Check	Policy Invoke
Call	46.57	19.87 (2.34x)	2.93 (15.87x)	2.12 (21.93x)
Load	47.69	9.74 (4.89x)	2.80 (17.03x)	1.94 (24.59x)
Store	26.40	8.43 (3.13x)	2.37 (11.14x)	1.84 (14.37x)

Numbers are operations/μs with relative slowdowns in parentheses. **Interpret** shows the cost of interpreting operations without policy checks or monitoring function invocations; **Policy Check** adds the cost of performing a policy check, but no monitoring function invocations; **Policy Invoke** is the additional cost of invoking a monitoring function that does no work.

Table 6.7: Execution Overhead of Operations on Sources and Sinks.

then taints the operation's result if the arguments were tainted. The taint propagation code incurs 1.25x average slowdown.

We now consider the performance benefit afforded by specialization for non-tainted data types, as described in Section 6.4.3, which allows the omission of label propagation instructions for operations that compute on non-tainted data. Table 6.6 shows the performance improvement achieved by this specialized taint tracking optimization. We present the performance of operations on *non-tainted* data when the optimization is enabled in the *Spec* column, and when it is not in the *No Spec* column. In the former case, V8 emits no label propagation code, and in the latter case, it emits such code even though the operations do not process tainted data. The specialized taint tracking optimization reduces execution time by 7% on average.

Our optimizations reduce the overhead of label propagation to tractable levels. However, ScriptPolice policies also invoke an interposition interface on data sources, code sources, and data sinks. The current implementation of interposition in ScriptPolice is in fact the major source of execution overhead in policy enforcement, as shown below.

In Table 6.7, we present measurements of the latency of operations on a data source in V8 under

ScriptPolice *vs.* in "vanilla" V8. (Note that V8 implements sources and sinks in the same way.) We further categorize the total cost of operations on sources and sinks into three components. First, we show the overhead caused by interpreting operations on sources and sinks in the *Interpret* column. Our current implementation does not support JITing for such operations. Then, we measure the cost of a policy check, which examines if a policy function is attached to a source or sink, as shown in the *Policy Check* column. Note that the interpretation cost is included in the policy check cost. Finally, we look at the cost of a policy function call, shown in the *Policy Invoke* column, where each operation on the source results in an invocation of a policy function that does no work but returns immediately. The *Policy Invoke* overhead is the total overhead of operations on sources and sinks, and it includes all three of the prior components.

Interpreting operations on sources and sinks results in an average 3.5x slowdown. A policy check increases the average slowdown up to 15x, and a monitoring function invocation further raises the overhead to 20x. In a real-world scenario, we expect the slowdowns introduced by monitoring functions to be greater than the one presented in our benchmark evaluation because a single source or sink may have multiple monitoring functions performing actual work.

6.6 Discussion

In this section, we discuss possible limitations of ScriptPolice policies, in particular, whether we identified all sources and sinks. Then, we present an interesting case of conflict of interest between a user and page developer which arises if we would apply the straight-forward Containment Source policy to all extensions. We further discuss trade-offs between the implementations of containment and prevention policies based on the DAC mechanism *vs.* the implementations built upon the IFC mechanism. Finally, the section concludes with a discussion on possible solutions for a problem of shared data stores.

6.6.1 Completeness of Interposition

Does ScriptPolice interpose on all sources and sinks? And what are the consequences if it does not? Clearly, if a source or sink is not monitored by a policy, an extension or page author may be able to evade the enforced policy by routing sensitive data (for a containment policy) or code (for a prevention policy) through such a source or sink.

Any argument we make about identifying all sources and sinks is based on manual scrutiny of the DOM API, the XMLHTTPRequest API, and the extension API. Our careful study of the DOM API reveals that there are relatively few code sources from which page data may be imported into the V8 execution environment as code. As such, we are fairly confident that our prevention policy monitors all code sources in the DOM API. There are also few *network* data sinks in the DOM API, so we are similarly confident that a policy may monitor all of them if required. There are, however, far more data sources in the DOM API. While we have exerted great effort to identify and monitor all of them in our containment and prevention policies, we cannot rule out that we may have missed a few. Adding additional data sources and sinks to a policy, however, is straightforward—that is, we believe the *methodology* of monitoring data sources and sinks should, with appropriate community scrutiny and feedback, be able

to cover the DOM in all its breadth.

6.6.2 Conflicts of Interest: User vs. Page Developer

Enforcing containment of sensitive data with naïve DAC policies, *e.g.,* the Containment Source policy, can create a subtle problem of a conflict of interests between a user and page developer. This conflict is best explained by example. Consider applying Containment Source to the AdBlock Plus extension, which hides ads embedded in pages. The user expects the web developer to correctly identify elements containing sensitive data in her pages so that the browser's containment policy can safeguard this data. However, the developer may be paid for displaying ads, and he may not want to lose revenue as he would if AdBlock Plus filtered his site's ads. In order to prevent the extension from stripping off the ads, the developer may mislead the policy by marking the ads as sensitive data, thus denying access to them by AdBlock Plus. But AdBlock Plus must read content in order to identify embedded ads!

To resolve such conflicts of interest, a user should wield absolute authority. The user should overrule the enforced containment policy if she is not satisfied with the restrictions imposed by the policy and wants to enable an extension's functionality. Note, however, doing so forces the user to give up the benefits of the containment policy, and effectively trust that an extension neither includes malicious data disclosures nor is vulnerable to privilege escalation by a malicious page.

In contrast, our Containment Sink ACL and ACL None policies, which confine local and remote extensions such as AdBlock Plus, do not encounter this conflict, as they do not rely on the marking of sensitive DOM elements, and thus do not require page author cooperation.

6.6.3 DAC policies vs. IFC policies

We presented implementations of the containment policy based on DAC. However, we also built an IFC-based implementation of the containment policy, which we did not describe because of constraints on taint propagation via implicit data flows in our IFC mechanism, Section 6.3.3. The IFC-based containment policy taints data originating from DOM elements tagged as sensitive within a page and restricts extensions from releasing the tainted data out of the browser via network data sinks. Note that it enforces no such restriction on the use of *nonsensitive* data.

The IFC policy offers broader applicability than any of our DAC policies *alone*, as the IFC mechanism allows an extension to access and process sensitive data, provided it does not release data derived from sensitive data to unauthorized origins. Thus, the IFC-based policy fits *all* extension models presented in Section 6.3.1 and achieves combined applicability of Containment Sink ACL and Containment Source. Moreover, the IFC-based containment policy provides better applicability for hybrid extensions, Section 6.3.1, where it may allow an extension to produce local and remote results for nonsensitive data and a local result for sensitive data (assuming a user denies sending sensitive data to a remote server). With our Containment Source policy, which we would have enforced in the scenario above, the extension may produce both results on nonsensitive data but none on sensitive data.

The IFC-based containment policy could be useful in preventing unauthorized disclosure of browser-managed sensitive data (*e.g.,* browsing history and bookmarks). The IFC-based policy could allow extensions to use these data without disclosing them to the network. This policy safely supports

richer functionality than the simple allow-deny policy currently enforced by the Chrome browser's extension permission mechanism.

Similarly, beyond the IFC-based prevention policy we have described, we also built an implementation of this policy based on DAC. When built atop the DAC mechanism, the prevention policy restricts *all* code injections into extensions' privileged background pages. We found the applicability of the DAC-based prevention policy narrower than that of the IFC-based prevention policy. In particular, the former policy breaks the Easy Reader extension [8], as the DAC policy cannot distinguish if an injected script is provided by a page or supplied by the extension. It thus prevents *all* script injections, even benign ones, such as when an extension dynamically introduces its own scripts. Easy Reader dynamically adds an HTML skeleton for each entry in an RSS feed to display it in a pop-up window. The skeleton contains event handlers that cause the DAC-based prevention policy to throw an exception. The IFC-based prevention policy instead allows such event handlers, as the HTML skeleton is derived from the extension's source, and thus does not become tainted.

Our containment policies focus on enforcing confidentiality guarantees for sensitive data in web pages. However, ScriptPolice is a general policy engine, and one may design policies that target integrity of sensitive data. We leave this for future work.

In this work, we have focused on extensions, as they execute with high privilege and present a significant threat to browser security. However, ScriptPolice and its policies should also be useful for preventing code injection attacks (*e.g.,* cross-site scripting attacks) in web applications, as well as for containment of possibly malicious code in third-party libraries used by web applications. ScriptPolice should be of further use to protect sensitive information in server-side JavaScript, as used in server applications implemented in the Node.js framework [114].

6.6.4 Challenges of Shared Data Stores

ScriptPolice enforces its containment policies on extension scripts with respect to the network: it prevents extensions that read sensitive data from pages from writing to network destinations other than those explicitly enumerated for the user and approved by her at extension installation time (if any). But extension scripts may also write to data stores within the browser: a page's DOM, an origin's site-local storage or cookie storage, *&c.* These data stores are *shared:* not only can extensions read and write them, but scripts within pages may also read and write them (within the containing page's origin). Such shared stores present a well known, fundamental challenge to information control systems, in general, and to privacy on the Web in particular [91]. In some cases, a user may desire an interaction between an extension script and page script that is mediated by communication via the DOM. For example, a spell-checker extension could correct the spelling of a word within a page by writing to that page's DOM, and a script within that page might send the corrected word to its own origin, or possibly even to some other origin (*e.g.,* a script that posts comments to a distinct comment-hosting site).

But shared data stores such as the DOM also present a privacy risk to a per-script policy system. ScriptPolice's containment policies allow extensions to write to web pages—such a write in and of itself, after all, does not risk a leak of sensitive information, as the browser user is entitled to see her

own sensitive data. However, because the DOM is shared between the extension and scripts within the page, and the policy enforced on the extension does not confine subsequent operations by scripts on the page's DOM, a malicious extension could essentially trick a page's script into leaking sensitive information without authorization! Consider, for example, that an extension denied write access to the network by the Containment Sink ACL None policy might read sensitive information from a page then write that sensitive information back to the same page, but without any DOM annotation that indicates the data are sensitive. Scripts within the page might then unwittingly send this "laundered" sensitive data to untrusted remote origins. In a variant of this attack, two colluding malicious extensions could leak sensitive data. Suppose one extension is confined by the Containment Sink ACL None policy, as before, and a second one is confined by the restrictive Containment Source policy, which prohibits the reading of any sensitive data from any page, but allows network writes to any remote origin. The first extension "launders" sensitive data back to the page from which it came, as before, then the second freely reads the "laundered" sensitive data from the page, and freely leaks it to a remote origin. Clearly, the sharing of the DOM is a threat to privacy.

For extensions permitted to read sensitive data from pages, either of two entirely straightforward enhancements to ScriptPolice ACL policies would robustly prevent leaks through shared stores like the DOM. First, ScriptPolice could tag *all* DOM elements written by such extensions with the `priv="true"` attribute, thus effectively labeling all DOM writes by extensions as sensitive. Page authors concerned about exfiltration of sensitive data would already need to include checks in their in-page scripts that disallow sending sensitive DOM elements so marked to third-party origins. Alternatively, ScriptPolice could force such extensions' changes to pages to be written to a *shadow DOM* [76, 88], which is not visible to scripts in pages or other extensions, but is visible to the renderer. In this way, the user would see page modifications made by an extension, but page scripts and other extensions would not. We have not yet implemented these enhancements, but doing so presents no significant design or implementation challenge. Unfortunately, while these enhancements would robustly protect users' privacy, they might in some cases interfere with the functionality expected by the user. For example, a comment-posting script within a page might not be allowed to read a *non-sensitive* spell-corrected word written to a page by an extension (or, in the case of the shadow DOM, might read the *uncorrected* word, despite the user's seeing the corrected one in the rendered window!). This conflict between robust privacy and functionality requiring (non-malicious) multi-script interactions is fundamental: it arises because scripts share data stores.

6.7 Conclusion

Modern web browsers rely on the same-origin policy to maintain the confidentiality and integrity of users' sensitive data in web applications. Because the same-origin policy does not limit the behavior of extensions, and extensions execute with elevated privilege, browsers today do not in fact robustly protect sensitive data. A developer may maliciously craft an extension that uses its elevated privileges to disclose pages' sensitive data. And a vulnerable extension may allow a page designer or man in the middle to inject malicious code into an extension, and thus achieve privilege escalation and disclose

sensitive information.

In this chapter, we have described ScriptPolice, a practical and general policy system for the Google Chrome web browser that robustly protects sensitive data—both from malicious extension authors, and from malicious page authors who exploit vulnerable extensions. ScriptPolice supports simple and general containment and prevention policies that defend against these threats. Our early experience with ScriptPolice suggests that these simple policies effectively safeguard sensitive data for a wide variety of web applications and browser extensions at acceptable performance cost.

Chapter 7

Discussion

We have worked on strengthening confidentiality and integrity guarantees for sensitive data in three domains of network-facing applications: the native domain, language runtime domain, and web application domain. In each domain, we applied a different system to reduce the privilege of software components: Wedge[1], the Native Client SFI sandbox, and ScriptPolice, respectively. In this chapter, we discuss the differences among these systems and attempt to identify if a single privilege reduction system may suite all three domains. The chapter concludes with directions for future research.

Wedge is based on the process abstraction, the hardware page protection mechanism, and SELinux policies. It is designed to privilege-separate legacy applications and enforce the principle of least privilege. Wedge may sandbox unmodified native code within privileged sthreads (processes). SELinux policies constrain the system call interface and prevent unprivileged sthreads from tampering with other processes and accessing the file system. Wedge enforces *static* policies on sthreads, *i.e.*, a fixed policy is specified at an sthread's creation time. However, flexible runtime policies may be implemented with the trusted broker abstraction—a trusted and privileged sthread mediates access to sensitive data for an unprivileged sthread via an IPC channel—which Wedge implements with callgates. To perform operations on sensitive data, an unprivileged sthread must call into a privileged compartment which results in a context switch between the compartments. This context switch overhead may have a significant performance impact on applications that make heavy use of sensitive data, *e.g.*, the Apache web server, as discussed in Section 3.5.4.

Software-based Fault Isolation (SFI) as implemented in the Native Client (NaCl) sandbox allows isolating untrusted and trusted native software components within the address space of a single process. Thus, one might potentially avoid the context-switch overhead incurred by Wedge by placing a trusted broker in the same process as a sandboxed software component. In fact, NaCl's trusted service runtime is a form of a trusted broker; it constrains the system call interface and may isolate sensitive data in its private address space. NaCl incurs load-time overhead because it verifies machine code to ensure that the code implements NaCl's safety constraints. This static verification takes place only once at the code's load time. NaCl also incurs runtime overhead introduced by software guard instructions that enforce NaCl's safety constraints and by NOP padding required for instruction alignment, as described

[1] Isolation with processes and the SELinux policies used for partitioning protocol implementations in Chapter 4 are similar to sthreads and the SELinux policies as used in Wedge, and therefore we omit discussing process isolation separately.

Comparison criteria	Wedge	NaCl	ScriptPolice
Sandboxing of native code	yes	yes	no
Sandboxing of system calls	SELinux	trusted service runtime	trusted JavaScript runtime
Sandboxing of in-memory data	sthread (process abstraction)	NaCl module (multiple modules in a single process)	instance of JavaScript runtime
Flexible runtime policies	callgate (trusted broker abstraction)	trusted service runtime	ScriptPolice policies
OS portability	Linux	Windows, MacOS, Linux	Windows, MacOS, Linux
Trusted code base	OS	OS, NaCl verifier, NaCl service runtime	OS and JavaScript runtime
Performance impact	context switch overhead	NaCl static verifier and software guard instructions	taint propagation and policy checks
Application porting effort	application partitioning	application partitioning and porting to NaCl sandbox	none

Table 7.1: Comparison of Privilege Reduction Systems.

in Section 5.1. Some porting effort may further be required to adapt legacy code to run within the NaCl sandbox.

ScriptPolice is a policy system built on top of the V8 JavaScript engine, and thus it is applicable only to software components written in JavaScript. It mediates JavaScript's interactions with the host application, *i.e.*, the browser, and constrains the system call API as well as access to sensitive data residing outside of the JavaScript execution environment.

Table 7.1 summarizes the comparison of the privilege reduction systems used in our work. Most likely, Wedge's context-switch overhead would make Wedge (and process isolation in general) perform unacceptably poorly for sandboxing language runtimes, and particularly the V8 JavaScript engine. JavaScript normally operates on pages' content via the Document Object Model (DOM), such that splitting the DOM and JavaScript engine will result in many expensive context switches.

Conversely, ScriptPolice cannot fit the native and language runtime domains as it does not support sandboxing of native code. However, the NaCl sandbox might be applicable and perform acceptably in all three domains of network-facing applications. Similarly to Wedge's sthreads, NaCl modules can be adapted to isolate untrusted software components. NaCl is today supported on more OSes than Wedge, but relies on a larger trusted code base, as summarized in Table 7.1. NaCl can isolate language runtimes and potentially enforce ScriptPolice's containment and prevention policies on extensions, with the further advantage of running the V8 JavaScript engine as an untrusted component.

One promising avenue for future research is to explore if NaCl may be of help to enforce confidentiality and integrity guarantees for web applications in the Google Chrome browser. Figure 7.1 shows the current architecture of the Chrome browser. The browser applies privilege separation [118] and implements a modular architecture [63]. The privileged browser kernel acts on behalf of a user and performs a limited number of simple operations, whereas per-browser-tab unprivileged rendering processes (renderers) handle untrusted content from the network and aggregate the most error-prone operations. A renderer process computes on content of a web site loaded in a browser tab and isolates

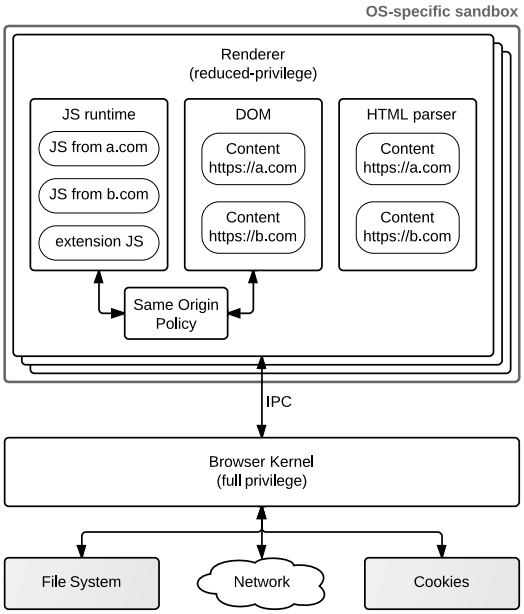


Figure 7.1: Architecture of the Google Chrome Browser.

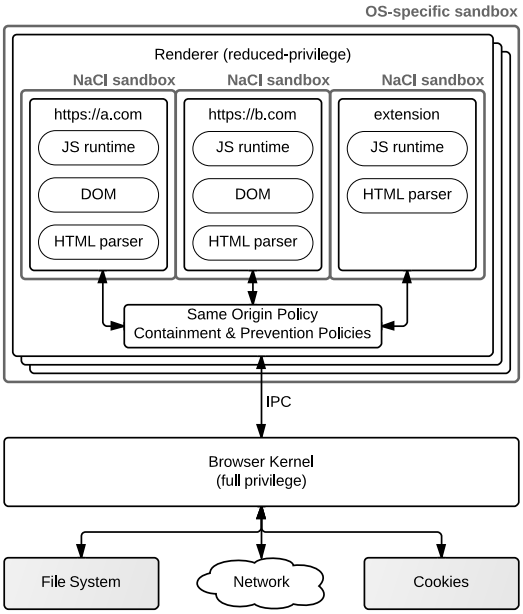


Figure 7.2: Restructured Architecture of the Google Chrome Browser.

high-risk components such as the HTML parser, the JavaScript engine, and the DOM. These components are complex and have been the source of vulnerabilities in the past. Google Chrome sandboxes renderer processes with OS-specific mechanisms [36–38] and restricts them from accessing the file system and the network; they also cannot tamper with other processes via system calls. Access to the network and persistent resources such as cookies and files is mediated by the user-privileged browser kernel.

The Chrome architecture protects users' sensitive information stored and processed by the system running the browser from a remote attacker who successfully compromises an unprivileged renderer. It is more likely that the attacker exploits a vulnerability in a renderer process rather than in the browser kernel as the latter handles no network input. The compromised renderer process is sandboxed, and it lacks privilege to access sensitive data on the system. However, despite the modular architecture, the attacker may still disclose a user's sensitive data processed by a web application within the compromised renderer. Consider a renderer process in Figure 7.1 that handles a site that aggregates content from origins `https://a.com` and `https://b.com` by using an iframe. In addition, there is an extension executing its scripts over content of both origins. If a page author from one of the origins or an extension author exploits the renderer process by supplying malicious content (HTML, CSS, JavaScript), she may disclose sensitive information from the other origin. Such a disclosure of sensitive information is possible because Same Origin Policy (SOP) decisions take place within renderers, and thus a compromised renderer may deviate from the SOP. Moreover, a compromised renderer process may read cookies from all web applications executed by the browser [63], and cookies are frequently used for authentication.

Future work might fruitfully investigate strengthening the security architecture of the Google Chrome browser by enabling per-origin isolation in renderer processes with the NaCl sandbox. Figure 7.2 shows a restructured browser architecture where each origin whose content appears in a renderer

process is isolated within an unprivileged NaCl module, and all computations such as HTML rendering and JavaScript execution are confined with the NaCl sandbox. The trusted service runtime arbitrates cross-origin SOP decisions. Thus, if an attacker supplies malicious content from one origin, she may compromise the NaCl module dedicated to her origin, but the NaCl module will not be able to access the other origins' content isolated within different modules. The NaCl service runtime may also enforce ScriptPolice's containment and prevention policies and restrict malicious and vulnerable extensions from disclosing sensitive data of web applications to unauthorized origins. Note that this proposal merely extends Chrome's architecture: the existing renderer sandbox is still in place.

The NaCl sandbox does not rely on the process abstraction for data isolation, which is why we believe it is a good fit for sandboxing content that requires frequent cross-origin communication, *e.g.*, an extension computes on pages of arbitrary origins. However, NaCl in its current state does not support multiple untrusted modules within a single address space. Our experience with Native Client suggests that this problem is solvable, and therefore, we believe further work in this direction is warranted.

Chapter 8

Conclusion

In this dissertation, we present our work on providing confidentiality and integrity guarantees for sensitive data through privilege reduction in network-facing applications. We explored three domains of network-facing applications.

In the domain of native applications, a remote attacker threatens confidentiality and integrity of sensitive information in monolithic, network-based applications as any exploitable vulnerability may allow the attacker to disclose and corrupt sensitive data. We looked at applying Wedge for splitting legacy, monolithic applications into reduced-privilege compartments to secure sensitive data. Our Wedge implementation of the finely partitioned SSL-enabled Apache web server has proven flexible and powerful enough to protect Apache against not only simple key disclosure attacks, but also against complex man-in-the-middle attacks at acceptable performance cost.

Our experience in strengthening the SSL-enabled Apache web server showed that privilege-separating cryptographic protocol implementations to protect sensitive data can be challenging and error-prone. We confirm this with two classes of practical exploit-based attacks on cryptographic protocol implementations, the session key disclosure attack and the oracle attacks, that can disclose users' sensitive data even in the state-of-the-art, reduced-privilege applications such as the OpenSSH server and the HiStar-labeled SSL web server. Privilege separation and DIFC alone do not secure the user sensitive data against these attacks unless an application has been specifically structured to thwart them.

We offer principles to guide programmers in partitioning cryptographic protocol implementations to defend against session key disclosure and oracle attacks. In essence, following these principles yields in correctly treating session key material and oracle-prone functions as sensitive, and limiting privilege accordingly. To demonstrate the practicality and wide applicability of the principles, we applied them to partition the OpenSSH client and to extend partitioning of the OpenSSH server. Further, we privilege-separated the OpenSSL library (client and server sides) and produced a drop-in replacement for the stock OpenSSL library. While structuring cryptographic protocol implementations requires careful programmer effort, our experience with hardening the OpenSSL library shows that one partitioning effort can be amortized over a broad range of security-conscious applications.

In the language runtime domain, modern browsers grant full privilege to embedded dynamic language runtimes which, for efficiency, rely on advanced techniques such as just-in-time compilation, large

libraries of native-code support routines, and garbage collection. It is difficult to ensure safety of language runtimes because of their complexity even though they process untrusted content and thus at risk of remote exploits. In case of a compromise of a language runtime, an adversary gains a browser's full privilege and may disclose and corrupt a user's sensitive data within web pages of different origins. To thwart this threat, we propose sandboxing language runtimes with our extended implementation of Native Client SFI to reduce the language runtimes' privileges towards data of foreign-origin pages. Our extension to NaCl is a set of constraints and mechanisms to enable NaCl's support for self-modifying code which is necessary for sandboxing language runtimes implementing JIT compilers and performance optimizations requiring runtime machine code modification. We applied extended NaCl to sandbox the JIT-enabled Google V8 JavaScript engine and produced implementations for the x86-32 and x86-64 architectures which incur moderate performance overhead,

As for the web application domain, we looked at how browsers ensure confidentiality and integrity of sensitive data in web applications and pages. They enforce the same origin policy which allows a web application to access data only within its own origin, mainly defined by the application's domain name. Because the same origin policy does not contain extensions, and extensions execute with elevated privilege and may manipulate data of every web application, browsers today do not robustly protect sensitive data. A developer may craft a malicious extension that uses its elevated privileges to disclose pages' sensitive data. A different scenario where browsers' defense fails is when a vulnerability in a privileged extension allows an adversary to design a malicious page injecting code into the vulnerable extension. The injected code escalates its privilege and may disclose sensitive data. We found zero-day code-injection vulnerabilities in four extensions from the Google Chrome store, and demonstrated how to exploit them with maliciously designed pages that disclose authentication cookies from pages opened in other tabs.

We built ScriptPolice—a practical and general policy system for the Google Chrome browser—and proposed enforcing simple and general security policies, prevention and containment policies, on execution of extensions. The containment policies restrict malicious extensions from disclosing sensitive data, and the prevention policy robustly protects sensitive data from malicious page authors who exploit vulnerable extensions. We applied our policies to a range of extensions and confirmed that the extensions function correctly (where an extension's function does not contradict the enforced policy) and that the polices achieve the goal of preventing sensitive information disclosure at acceptable performance cost.

Bibliography

[1] Adobe flash player. http://www.adobe.com/software/flash/about.

[2] Alexa – The Web Information Company. http://www.alexa.com.

[3] Android operating system. http://www.android.com.

[4] Apache HTTP server. The Apache Software Foundation. http://httpd.apache.org.

[5] Chrome browser. http://www.google.com/intl/en/chrome/browser.

[6] Common Vulnerabilities and Exposures. http://cve.mitre.org/index.html.

[7] Download Master (Google Chrome Extension). https://chrome.google.com/webstore/detail/download-master/mcceagdollnkjlogmdckgjakjapmkdjf.

[8] Easy Reader (Google Chrome Extension). https://chrome.google.com/webstore/detail/easy-reader/jgcemdmmcnofcafeaebdhakefingiolb.

[9] FlashBlock (Google Chrome Extension). https://chrome.google.com/webstore/detail/flashblock/gofhjkjmkpinhpoiabjplobcaignabnl.

[10] Freshbooks—an online invoicing software. http://www.freshbooks.com.

[11] Google Dictionary (Google Chrome Extension). https://chrome.google.com/webstore/detail/mgijmajocgfcbeboacabfgobmjgjcoja.

[12] Google's web-based word processor, spreadsheet, presentation, form, and data storage service. http://docs.google.com.

[13] The hardware-based performance monitoring interface for linux. http://perfmon2.sourceforge.net.

[14] Mono project. http://www.mono-project.com.

[15] OCSP stapling vulnerability in OpenSSL. http://www.openssl.org/news/secadv_20110208.txt.

[16] Office Web Apps—web-based version of Microsoft's office productivity suite.

[17] OpenSSH: A Bug in the sshd Privilege Separation Monitor. http://www.openssh.com/txt/release-4.5.

[18] OpenSSH Buffer Mismanagement Vulnerabilities. http://www.securityfocus.com/bid/8628/info.

[19] OpenSSH Challenge-Response Buffer Overflow Vulnerabilities. http://www.securityfocus.com/bid/5093/info.

[20] OpenSSH Security Advisory: Buffer Management Errors. http://www.openssh.com/txt/buffer.adv.

[21] OpenSSH Security Advisory: Input Validation Error that Can Result in an Integer Overflow and Privilege Escalation. http://www.openssh.com/txt/preauth.adv.

[22] OpenSSH security vulnerabilities. OpenSSH. http://www.openssh.org/security.html.

[23] OpenSSH security vulnerabilities. CVE Details. http://www.cvedetails.com/vulnerability-list/vendor_id-7161/Openssh.html.

[24] OpenSSL: CVE-2006-3738. http://www.openssl.org/news/secadv_20060928.txt.

[25] OpenSSL: CVE-2007-5135. http://www.openssl.org/news/secadv_20071012.txt.

[26] OpenSSL: CVE-2010-0740. http://www.openssl.org/news/secadv_20100324.txt.

[27] OpenSSL: CVE-2010-3864. http://www.openssl.org/news/secadv_20101116.txt.

[28] OpenSSL: CVE-2010-4180. http://www.openssl.org/news/secadv_20101202.txt.

[29] OpenSSL: CVE-2012-2110. http://www.openssl.org/news/secadv_20120419.txt.

[30] OpenSSL project. http://www.openssl.org/.

[31] OpenVPN security vulnerabilities. CVE Details. http://www.cvedetails.com/vulnerability-list/vendor_id-3278/Openvpn.html.

[32] Pandora—an automated music recommendation service. http://pandora.com.

[33] SecurityFocus. http://www.securityfocus.com/.

[34] Slacker—an interactive internet radio service. http://slacker.com.

[35] Sunspider benchmark suite. http://www2.webkit.org/perf/sunspider/sunspider.html.

[36] The Chromium Projects: Linux Sandbox. https://code.google.com/p/chromium/wiki/LinuxSandboxing.

[37] The Chromium Projects: OSX Sandbox. http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design.

[38] The Chromium Projects: Windows Sandbox. http://www.chromium.org/developers/design-documents/sandbox.

[39] V8 benchmark suite. http://v8.googlecode.com/svn/data/benchmarks/v7/run.html.

[40] V8 javascript engine. http://code.google.com/p/v8.

[41] V8 security vulnerabilities. CVE Details. http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-17734/Google-V8.html.

[42] vsftpd: probably the most secure and fastest FTP server for UNIX-like systems. https://security.appspot.com/vsftpd.html.

[43] Wave—a web-based accounting application. http://waveaccounting.com.

[44] WebOS mobile operating system. http://developer.palm.com.

[45] *Secure hash standard*. National Institute of Standards and Technology, 1995. Federal Information Processing Standard 180-1.

[46] Ecmascript language specification, 2001. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-327.pdf.

[47] Vulnerabilities in Microsoft XML core services could allow remote code execution. Microsoft Security Bulletin MS06-061, November 2006. http://technet.microsoft.com/en-us/security/bulletin/ms06-061.

[48] Expat UTF-8 character XML parsing remote denial of service vulnerability. SecurityFocus, January 2009. http://www.securityfocus.com/bid/36097.

[49] Novell File Reporter Agent: XML tag remote code execution vulnerability. SecurityFocus, April 2011. http://www.securityfocus.com/bid/47144/.

[50] Sony Online Entertainment announces theft of data from its systems, May 2011. http://www.soe.com/securityupdate/pressrelease.vm.

[51] Vulnerability in Windows kernel-mode drivers could allow remote code execution. Microsoft Security Bulletin MS11-087, December 2011. http://technet.microsoft.com/en-us/security/bulletin/ms11-087.

[52] Anonymous hacks oil giants, leaks employees passwords. CY.TALK, July 2012. http://privacy.cytalk.com/2012/07/anonymous-hacks-oil-giants-leaks-employees-passwords/.

[53] OpenSSL vulnerabilities. OpenSSL, 2012. http://www.openssl.org/news/vulnerabilities.html.

[54] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security*, 2009.

[55] Advance Micro Devices Inc. *AMD64 Architecture Programmers Manual Volume 1: Application Programming*, 2009.

[56] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. http://www.phrack.org/phrack/49/P49-14.

[57] J. Ansel, P. Marchenko, D. Schuff, B. Chen, and U. Erlingsson. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[58] C. Arthur. Yahoo Voice hack leaks 450,000 passwords. Guardian, July 2012. http://www.guardian.co.uk/technology/2012/jul/12/yahoo-voice-hack-attack-passwords-stolen.

[59] J. Aubourg, J. Song, and H. R. M. Steen. XMLHttpRequest, December 2012. http://www.w3.org/TR/XMLHttpRequest.

[60] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 2003.

[61] A. Barth. The Web Origin Concept. IETF RFC 6454, December 2011. http://tools.ietf.org/html/rfc6454.

[62] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Annual Network & Distributed System Security Symposium*, 2010.

[63] A. Barth, C. Jackson, C. Reis, and G. C. Team. The security architecture of the chromium browser. Technical report, Stanford University, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[64] D. J. Bernstein. qmail. http://cr.yp.to/qmail.html.

[65] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, University College London, UK, 2009. http://discovery.ucl.ac.uk/18902/1/18902.pdf.

[66] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[67] D. Blazakis. Interpreter exploitation: pointer inference and JIT spraying. In *Proceedings of Black Hat DC*, 2010.

[68] I. Bukanov. Javascript garbage-collection hazards. Mozilla, 2006. http://www.mozilla.org/security/announce/2006/mfsa2006-01.html.

[69] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the Google Chrome extension security architecture. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[70] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, 1970.

[71] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 1977.

[72] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1984.

[73] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009.

[74] T. Dierks and C. Allen. The TLS protocol version 1.0. IETF RFC 2246, January 1999. http://www.ietf.org/rfc/rfc2246.txt.

[75] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[76] X. Dong, M. Tran, Z. Liang, and X. Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[77] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[78] Ú. Erlingsson. High-performance binary applets, 1997. http://www.cs.cornell.edu/home/ulfar/cuba/paper.

[79] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[80] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[81] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 1986.

[82] A. O. Freier, P. Karlton, and P. C. Kocher. The Secure Sockets Layer (SSL) protocol version 3.0. IETF RFC 6101, August 2011. http://tools.ietf.org/html/rfc6101.

[83] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[84] Google Inc. Chrome.* APIs. http://developer.chrome.com/extensions/api_index.html.

[85] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[86] A. L. Hors, P. L. Hgaret, G. Nicol, L. Wood, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification, April 2004. http://www.w3.org/TR/DOM-Level-3-Core.

[87] M. Humphries. PSN was running on unpatched Apache server with no firewall. Geek.com, May 2011. http://www.geek.com/articles/games/psn-was-running-on-unpatched-apache-server-with-no-firewall-2011055/.

[88] L. Ingram and M. Walfish. Treehouse: Javascript sandboxes to helpweb developers help themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[89] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2012. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[90] Intel Inc. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1*, 2010.

[91] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th conference on World Wide Web*, 2006.

[92] R. A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the 18th Annual Computer Security Applications Conference*, 2002.

[93] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[94] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[95] M. Lee. Over 21,000 plain text passwords stolen from Billabong. ZDNet, July 2012. http://www.zdnet.com/over-21000-plain-text-passwords-stolen-from-billabong-7000000842/.

[96] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co, 1st edition, 1999.

[97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[98] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[99] Lostmon. Gmail Checker Plus Chrome extension XSS, 2010. http://lostmon.blogspot.co.uk/2010/06/gmail-checker-plus-chrome-extension-xss.html.

[100] Lostmon. Notifier for Google Wave Chrome extension XSS/CSRF, 2010. http://lostmon.blogspot.co.uk/2010/06/notifier-for-google-wave-chrome.html.

[101] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[102] P. Marchenko. Checker Plus for Gmail (Google Chrome Extension) Code Injection Vulnerability, 2012. http://nrg.cs.ucl.ac.uk/jstaint/exploits/mail_checker.

[103] P. Marchenko. Easy Reader (Google Chrome Extension) Code Injection Vulnerability, 2012. http://nrg.cs.ucl.ac.uk/jstaint/exploits/easy_reader.

[104] P. Marchenko. Microformats (Google Chrome Extension) Code Injection Vulnerability, 2012. http://nrg.cs.ucl.ac.uk/jstaint/exploits/microformats.

[105] P. Marchenko. RSS Subscription (Google Chrome Extension) Code Injection Vulnerability, 2012. http://nrg.cs.ucl.ac.uk/jstaint/exploits/rss_subscription.

[106] P. Marchenko and B. Karp. Structuring protocol implementations to protect sensitive data. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[107] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[108] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 1960.

[109] K. Millikin and F. Schneider. A new crankshaft for v8, 2010. http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html.

[110] Mozilla.org. HTTP cookies. http://developer.mozilla.org/en/Web_Development/HTTP_cookies.

[111] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[112] S. Musil. Formspring disables user passwords in security breach. CNet, July 2012. http://news.cnet.com/8301-1009_3-57469944-83/formspring-disables-user-passwords-in-security-breach/.

[113] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[114] Node.js. http://nodejs.org.

[115] S. Pichai. Introducing the Google Chrome OS. Google Official Blog, Jul 2009. http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html.

[116] E. Protalinski. 8.24 million Gamigo passwords leaked after hack. ZDNet, July 2012. http://www.zdnet.com/8-24-million-gamigo-passwords-leaked-after-hack-7000001403/.

[117] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[118] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[119] R. Rivest. The MD5 Message-Digest Algorithm. IETF RFC 1321, April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[120] W. Robert, A. Jonathan, L. Ben, and K. Kris. Capsicum: practical capabilities for unix. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[121] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) session resumption without server-side state. IETF RFC 5077, January 2008. http://tools.ietf.org/html/rfc5077.

[122] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[123] Scut. Exploiting format string vulnerabilities. TESO Security Group, September 2001. http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf.

[124] Secunia. Dokodemo Rikunabi 2013 Unspecified Cross-Site Scripting Vulnerability, 2012. http://secunia.com/advisories/48813.

[125] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[126] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[127] Simes. How to break out of a chroot() jail. BPFH.net, May 2002. http://www.bpfh.net/simes/computing/chroot-break.html.

[128] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[129] The Web Application Security Consortium. Cross site scripting. http://projects.webappsec.org/w/page/13246920/CrossSiteScripting.

[130] C. Vallabhan. Nvidia hacked; user records compromised. ITP.net, July 2012. http://www.itp.net/589777-nvidia-hacked-user-records-compromised.

[131] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.

[132] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[133] T. Ylonen and E. C. Lonvick. The Secure Shell (SSH) authentication protocol. IETF RFC 4252, January 2006. http://www.ietf.org/rfc/rfc4252.txt.

[134] T. Ylonen and E. C. Lonvick. The Secure Shell (SSH) protocol architecture. IETF RFC 4251, January 2006. http://www.ietf.org/rfc/rfc4251.txt.

[135] T. Ylonen and E. C. Lonvick. The Secure Shell (SSH) transport layer protocol. IETF RFC 4253, January 2006. http://www.ietf.org/rfc/rfc4253.txt.

[136] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[137] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.