Structuring Protocol Implementations to Protect Sensitive Data

Petr Marchenko and Brad Karp
University College London, Gower Street, London WC1E 6BT, UK
{p.marchenko,bkarp}@cs.ucl.ac.uk

Abstract

In a bid to limit the harm caused by ubiquitous remotely exploitable software vulnerabilities, the computer systems security community has proposed primitives to allow execution of application code with reduced privilege. In this paper, we identify and attack the vital and largely unexamined problem of *how to structure* implementations of cryptographic protocols to protect sensitive data despite exploits. As evidence that this problem is poorly understood, we first identify two attacks that lead to disclosure of sensitive data in two published state-of-the-art designs for exploit-resistant cryptographic protocol implementations: privilege-separated OpenSSH, and the HiStar/DStar DIFC-based SSL web server. We then describe how to structure protocol implementations on UNIX- and DIFC-based systems to defend against these two attacks and protect sensitive information from disclosure. We demonstrate the practicality and generality of this approach by applying it to protect sensitive data in the implementations of both the server and client sides of OpenSSH and of the OpenSSL library.

1 Introduction

Cryptographic protocols are entrusted to preserve the integrity and secrecy of 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 software. Clearly, the desired outcome for secure system designers implementing a secure data transfer protocol like SSH [14] or SSL/TLS [4] is *end-to-end* integrity and secrecy for sensitive data, despite not only in-network threats, but also threats that may arise from the behavior of the protocol implementation(s) at the ends of the wire. The dismal past two decades of remotely exploitable vulnerabilities in software deployed widely on network-attached hosts are thus real cause for alarm—even if the abstract design of a cryptographic protocol is correct, the protocol's very implementation is a worryingly weak link in achieving end-to-end security goals.

In the quest for a lasting end-to-end defense for sensitive data against disclosure or corruption by a remote attacker, whatever vulnerabilities and exploits come to light in the future, the systems research community has in recent years sought to put the venerable *principle of least privilege* [11] into better practice in the software running on network-connected servers. This design tenet dictates that the 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. Here, privilege corresponds to access rights for system resources: to read or write the filesystem, memory, or network, to invoke a system call, *&c.* In the context of exploitable vulnerabilities and sensitive information, least privilege amounts to designing an application with the expectation that exploits will occur, but limiting the harm that they may cause by restricting the actions that an attacker may take post-exploit.

Early work [6, 10] explored how to minimize privilege on compartments instantiated as standard UNIX processes. More recently, the community has devoted considerable effort to providing various operating system primitives intended to make it easier for programmers to adhere to the principle of least privilege. These primitives range from operating system support for decentralized information flow control (DIFC), which tracks the flow of sensitive information in an application at run-time and limits the privileges of any compartment exposed to sensitive information [7, 13, 15, 16], to process-like primitives that deny privileges by default to newly created compartments, and thus lessen the likelihood of accidental propagation of privileges between compartments against the programmer's intent [2].

While these results all represent important advances over the prior state of the art, we believe that proposals to date for new primitives to encourage progammers' adherence to least privilege largely ignore a central, vital question: *how should a programmer structure code and limit privilege to prevent disclosure or corruption of sensitive data by an attacker who can exploit a vulnerability?* Regardless of the primitives used, this daunting question looms. To their credit, the proposers of these primitives present examples of how to structure application code to use them. But these examples are typically offered as existential evidence that the primitives themselves are useful; no guidance or principles are offered for how one may restructure a legacy application's code to use the primitives and robustly provide the desired end-to-end secrecy and/or integrity guarantees.

Moreover, the structures of these example applications are complex, as they are typically split into many compartments. To wit, the OKWS web server spreads its code among at least 5 compartments (processes) [6], the sthread-partitioned Apache/SSL web server consists of 9 compartments (sthreads and callgates) [2], and the HiStar/DStar-labeled Apache/SSL web server consists of 7 compartments (processes) [16]. Each application's many compartments are configured with different privileges and labels, respectively, and they are interconnected in complex patterns. Structuring code to use these primitives appears difficult. Indeed, as we show in Section 3, even highly security-conscious programmers using state of the art techniques have not adequately considered how to defend cryptographic protocol implementations from exploit-based attacks. In particular, we describe two general classes of attack on inadequately structured code for cryptographic protocols, and demonstrate that two state-of-the-art cryptographic protocol implementations, one in privilege-separated OpenSSH [10] and the other in a DIFC-labeled Apache/SSL web server [16], are vulnerable to these attacks.

In this paper, we offer a practical improvement over the status quo: principles to guide programmers in structuring cryptographic protocol implementations so as to enforce secrecy and integrity guarantees for sensitive user data *end-to-end,* including in cases where a remote attacker exploits untrusted application code.

Our contributions include:

- We define two major classes of attack on cryptographic protocol implementations: *session key disclosure attacks* and *oracle attacks*.
- We provide protocol-agnostic principles for structuring cryptographic protocol implementations to protect sensitive data against disclosure and corruption when an exploitable vulnerability is present in code that processes network input.
- As evidence of the practicality of these principles, we present restructured implementations of the OpenSSH server and client to limit privilege so as to protect users' sensitive data from an adversary who can remotely exploit the implementation. We further present a restructured implementation of the OpenSSL library 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.

2 Background

We now summarize the state of the art in protecting sensitive data in network server software. The two main approaches in use are privilege separation and decentralized information flow control (DIFC).

2.1 Privilege Separation with Processes

In a monolithic application, in which all code executes in a single compartment (under UNIX or Linux, a process), all instructions execute with full privilege. 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. Privilege separation [10] has proven effective in mitigating these threats. This technique follows from the observation that an application need not execute individual operations with the union of all privileges needed by all operations during the application's entire lifetime. Many vulnerability-prone operations, such as parsing, do not require access to sensitive information or the filesystem. If we partition a monolithic application into compartments and restrict some compartments' privileges, an exploit in an unprivileged compartment will not be able to disclose or corrupt sensitive information to which it does not have access. Code that runs in privileged compartments, however, must be carefully audited to protect the sensitive data it can access.

The privilege-separated OpenSSH server [10] divides the server's code into separate standard UNIX/Linux processes. This partitioning includes a network-facing unprivileged process that performs key exchange and authentication protocols, and a privileged monitor process running as `root` that exports an interface to the unprivileged process to allow invocation of privileged operations, such as signing with the server's private key, verifying user credentials, *&c.*

This structure is intended to deny the attacker execution of code with `root` privilege on the server; the attacker only interacts directly with the unprivileged process. Provos *et al.* state that "programming errors occurring in the unprivileged parts can no longer be abused to gain unauthorized privileges" [10]. This claim holds because the unprivileged process executes with restricted file system access (enforced with a `chroot` system call), and with unused user and group IDs of `nobody`, which prevent it from tampering with other processes.

The SELinux security extensions to Linux [8], which post-date Provos *et al.*'s work, allow enforcement of flexible mandatory access control policies specified by a system administrator. These policies support finer-grained restriction of a process's privileges than under stock Linux, primarily by checking system call invocations in the kernel against a per-process access control list. We employ these extensions in our cryptographic protocol implementations for OpenSSH and OpenSSL.

2.2 DIFC

Decentralized information flow control (DIFC), as implemented in the research prototype operating systems

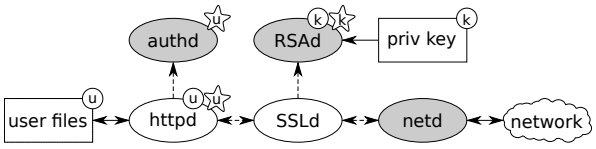

Figure 1: HiStar-labeled SSL web server. We omit *SSLd*'s and *netd*'s labels in the interest of brevity.

Asbestos [13] and HiStar [15], and retrofitted to Linux in Flume [7], offers a different approach to limiting privilege within applications. In these systems, a programmer expresses an information flow policy by labeling data according to its sensitivity level. Should an unprivileged compartment access data labeled as sensitive, it becomes tainted, and at run-time, the operating system prevents it from communicating with compartments tainted with lower levels of sensitivity, or with the network or console. This way, an unprivileged compartment cannot convey sensitive data out of the application. To allow output, *trusted compartments* perform privileged operations on sensitive data: they own sensitive labels, and are thus allowed by the operating system to *declassify* sensitive information, stripping it of its sensitivity label(s).

Building on these DIFC primitives, Zeldovich *et al.* present a state-of-the-art privilege-separated SSL web server [16], shown in slightly simplified form in Figure 1. Ovals represent code: shaded ovals are trusted, privileged compartments, while white ovals are untrusted 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. Circles annotating data items and compartments indicate labels; in the latter case, a compartment is tainted with the label in question. Finally, a label within a star denotes that a compartment owns that label (and may declassify data labeled with it).

The HiStar-labeled SSL web server is partitioned into several untrusted compartments to mitigate the effect of a compromise of any single compartment. The major compartments are a per-connection *SSLd*, per-connection *httpd*, and *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 trusted *authd* daemon. Label ownership allows *httpd* to read the user's data and declassify it for transfer over the network. The trusted *netd* serves as a barrier between the application and the network. It passes only declassified data (with no label) to the network.

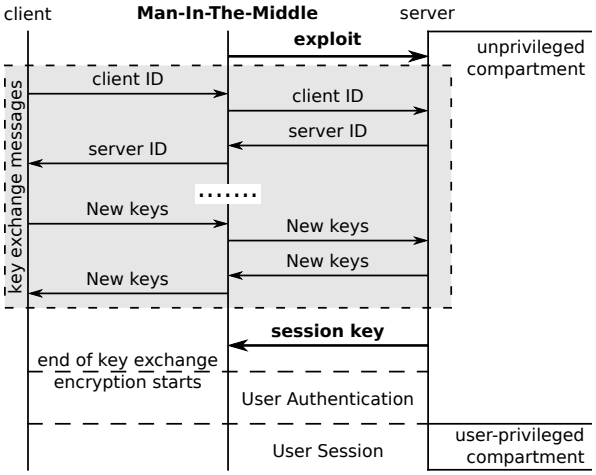

Figure 2: Session key disclosure attack against privilege-separated OpenSSH server.

3 Attacks on Protocol Implementations

The designers of cryptographic protocols like SSH and SSL aim to provide end-to-end confidentiality and integrity for users' data transferred during a session. When applied correctly, both privilege separation and DIFC can ensure that exploits of unprivileged compartments in a cryptographic protocol's implementation will not lead to violations of these properties. In this section, we present two attacks that violate the confidentiality and integrity of sensitive user data in two state-of-the-art privilege-separated systems: one in privilege-separated OpenSSH, and one in a HiStar-labeled Apache-derived SSL web server.[1]

3.1 Session Key Disclosure Attack

The partitioning goal stated by the designers of privilege-separated OpenSSH was to prevent attackers' executing code with `root` privilege. However, as we will see, that goal is not sufficient to preserve the confidentiality and integrity of the *user's* sensitive data.

Bittau *et al.* describe an active man-in-the-middle attack against an SSL-enabled Apache Web server [2] that is also valid against a privilege-separated OpenSSH server. We term this attack the *session key disclosure attack* (SKD attack). While Bittau *et al.* narrowly discuss this attack in the context of an SSL implementation, we now demonstrate that this attack applies against any protocol in which the two parties share a symmetric secret key.

In the SKD attack, an active man in the middle compromises an unprivileged compartment on the server, discloses the user's session key, and can then decrypt the sensitive data transmitted during the session. This attack succeeds because the unprivileged compartment responsible for key exchange and server authentication can read the session key shared between the server and client.

We illustrate the SKD attack on Diffie-Hellman key exchange in OpenSSH in Figure 2. Here an unprivileged compartment processes key exchange messages and invokes a privileged monitor to sign a session ID with the server's private key (the privileged monitor is not shown in the figure). The user-privileged compartment executes with the authenticated user's UID and provides a remotely accessible shell.

The attacker begins by exploiting the server's unprivileged compartment. He relays all key exchange messages from a legitimate user. The server and the user compute a shared session key, which the attacker's injected code sends to the attacker from the compromised compartment. After user authentication, the user transmits sensitive data encrypted with the compromised session key. Using the session key, the attacker can reveal the user's sensitive data, as well as inject her own commands and obtain further sensitive information stored on the server. Moreover, the session key also provides secrecy for the user authentication interaction, so the password of a client using password authentication will be compromised.

We note with interest that the state-of-the-art, HiStar-labeled SSL web server [16] aims to safeguard users' sensitive data from disclosure to other users. But because the designers of this cryptographic protocol implementation did not consider the SKD attack when structuring their code, this server is vulnerable to the SKD attack in the same way that the above-described privilege-separated OpenSSH server is. Specifically, the untrusted *SSLd* compartment computes a session key for a user's connection, but if an active man-in-the-middle attacker compromises this compartment, she may disclose the session key.

3.2 Oracle Attack

Another attack violates the confidentiality of a user's private data in the HiStar-labeled SSL web server shown in Figure 1. Depending on the key exchange protocol in use, *RSAd* signs either the ephemeral RSA key or the public Diffie-Helman components supplied by the untrusted *SSLd* with the server's permanent private key. This signature authenticates the server to the client. It is possible, however, to abuse the signing operation exported by *RSAd*. Although a compromised *SSLd* cannot directly read the private key, it can sign any data chosen by the attacker; the attacker controls the *SSLd* compartment, and can invoke *RSAd* with any arguments she chooses. Thus, the attacker can use a compromised *SSLd* to produce valid signatures using the server's identity. This example demonstrates that simply putting sensitive data beyond direct reach of untrusted code does not provide sufficient isolation.

We name such attacks against a cryptographic protocol's partitioning *oracle attacks*. Any trusted compartment or sequence of trusted compartments isolating sensitive data and exporting privileged operations to untrusted code can be an oracle. An oracle takes untrusted input from untrusted code and returns the result of a privileged operation. By choosing its inputs appropriately, an attacker can obtain sensitive information by invoking the trusted compartment. *SSLd* is meant only to pass *RSAd* an ephemeral key or the Diffie-Helman components for *its own current session* for signing. But if an active man-in-the-middle attacker compromises *SSLd*, she can sign arbitrary keys and DH components and present them to other users, and so impersonate the server.

We have further identified oracle structures in the "baseline" privilege-separated OpenSSH server [10]. The trusted monitor process exposes a private key-signing operation to the unprivileged compartment for authentication of the server during key exchange. The unprivileged compartment thus has an oracle for the server's private key, and an attacker who compromises that compartment can impersonate the OpenSSH server, just as was described for the SSL web server above.

While partitioning the SSH and SSL/TLS protocols, we identified still other oracle attacks. Digital signatures suffer not only from signing oracles, but also signature verification oracles, in which an attacker can pass a signature verification operation by supplying chosen inputs to a trusted compartment performing this privileged operation. There also exists an oracle where an attacker forces a set of trusted compartments generating a session key to produce the same key used in a past user's 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.

3.3 Discussion

The SKD and oracle attacks are *independent* of the low-level system primitive used to limit privilege; they appear equally in applications built with privilege separation and DIFC. These attacks are made possible by *weakly structured cryptographic protocol implementations*. The implementation of a cryptographic protocol should guarantee the same properties provided in the middle of the network: data confidentiality, data integrity, and robust authentication of the peers, even if untrusted compartments in its implementation are compromised. Avoiding SKD and oracle attacks requires subtle structuring of the implementation of a cryptographic protocol.

The SKD and oracle attacks target *building blocks* of

cryptographic protocols. Risk of an SKD attack exists in many cases where a session key and key exchange protocol are used. Similarly, oracle attacks are associated with basic cryptographic operations such as encryption, decryption, signing, signature verification, message authentication, *&c*.

We next propose guiding principles for defense against SKD and oracle attacks. Just as these attacks arise in building blocks for cryptographic protocols, these principles concern how to implement these building blocks safely. We thus believe both the attacks and defenses apply to many cryptographic protocols.[2]

4 Principles for Partitioning

In this section, we define principles to guide the programmer when partitioning an implementation of a cryptographic protocol into reduced-privilege compartments. These principles allow preserving the key security properties of the protocol end-to-end, even when untrusted compartments are compromised. Our principles are agnostic to the underlying privilege-enforcement mechanism. Thus, they may be applied in DIFC-based systems, in privilege-separated systems based on Linux processes (with or without SELinux policies), and in other systems. They apply both to the client and server sides of cryptographic protocols.

Throughout, we assume that an attacker can compromise untrusted code, and can execute arbitrary code in its compartment, though only with the privileges allowed in that compartment. In this threat model, if an untrusted compartment acquires sensitive information or an attacker compromises a privileged compartment, we presume she obtains sensitive information.

4.1 Two-Barrier, Three-Stage Partitioning

A cryptographic protocol typically shares a symmetric secret key between two communicating parties, used to compute message authentication codes (MACs) and to encrypt data. A key exchange protocol confidentially shares this symmetric key. In addition, in some applications, the cryptographic protocol must authenticate peers to each other. Any authentication method that does not rely on transferring sensitive data, such as public key authentication, may be performed during the key exchange protocol, before a session-key-encrypted channel has been established. The SSL/TLS protocol fits this model [4]. In contrast, password-based authentication, *e.g.,* as supported by SSH [14], sends sensitive data over the network, and must therefore only authenticate after the session-key-MACed and -encrypted channel has been established. After authentication, an application is assured of the remote principal's identity, and can grant the remote principal access to locally stored sensitive data.

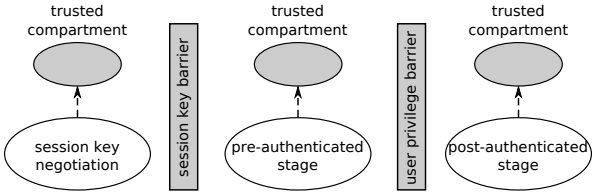

Figure 3: Barriers and stages in protocol partitioning.

We distinguish two attack models. The first is that of the SKD attack described in Section 3.1, where a man-in-the-middle attacker exploits a vulnerability in a client or server application to obtain the peers' session key. The second attack model is that of an *impersonation attack,* where an attacker exploits an endpoint and subverts authentication in order to impersonate one of the peers.

In order to prevent these attacks, a partitioned application should implement structures that we term a *session key barrier* and a *user privilege barrier.* These two barriers partition an application into three stages, as shown in Figure 3. The first such stage, the *session key negotiation stage*, performs the key exchange protocol. The second stage, the *pre-authenticated stage*, conducts peer authentication. Finally, the *post-authenticated stage* processes user requests. Within each stage, one untrusted compartment handles network input and executes without privileges to read or write sensitive data, while multiple trusted compartments execute with privilege to access sensitive data. These trusted compartments export any necessary privileged operations to the untrusted compartment.

Session Key Barrier The session key barrier denotes the killing of the untrusted compartment that completes session key negotiation and the subsequent spawning of a new untrusted compartment (in Linux, a process) to continue execution in the pre-authenticated stage. We now explain why this structure is necessary.

The untrusted compartment performing session key negotiation (before the session key barrier) is the only untrusted compartment in the partitioning of the cryptographic protocol implementation that processes cleartext, unauthenticated messages from the network. These messages (and exploits!) may arrive from an SKD attacker. Thus, while the untrusted compartment in the session key negotiation stage interacts with the remote peer to compute the session key, it should not have read access to the session key. In addition, any data that allows *deriving* the session key, such as a private Diffie-Hellman component (in the case of Diffie-Hellman key exchange) or a pre-master secret (in the case of RSA-based session key establishment in SSL) should be also considered sensitive. All access to privileged operations with such data should be provided via trusted compartments.

Because this compartment only processes messages in

cleartext, it does not in fact need read access to the session key; only the next stage, the pre-authenticated stage, which continues execution after the channel between the two peers is MAC'ed and encrypted with the session key, needs the session key.

> 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 untrusted compartment performing session key negotiation may be exploited, we cannot trust the provenance of the code executing in that compartment at the end of session key negotiation, and rather than allowing that compartment to continue execution in the pre-authenticated stage, where it would have access to the session key, we kill it (*i.e.,* kill the Linux process).

But why can't an SKD attacker exploit the untrusted compartment in the pre-authenticated stage? This compartment only processes input that is MAC'ed using the now available session key. A would-be SKD attacker cannot inject messages with a valid MAC into the channel, and so is precluded from exploiting this compartment. We assume here that the MAC computation function itself, which processes network input, can be audited and trusted not to be exploited.

Thus, both the MAC on the channel and the killing of the untrusted compartment in which session key negotiation has completed effectively erect a barrier between any SKD attacker and the session key.

> 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 untrusted compartment.

Note that the "original" privilege-separated OpenSSH server does in fact destroy the unprivileged compartment after performing user authentication, but we require this to be done after key exchange. The "original" OpenSSH destroys the compartment not for SKD attack-resistance reasons, but because of a programming difficulty. In this implementation, the unprivileged compartment has the user ID of `nobody`, but must change its user ID to that of the authenticated user. Changing a process's user ID requires `root` privilege; therefore, the monitor kills the compartment and creates a new one with the required user ID.

Transitioning to the pre-authenticated stage may require transferring state from the unprivileged compartment of the session key negotiation stage to the unprivileged compartment of the pre-authenticated stage. As this state comes from a compartment that may be controlled by an SKD attacker, the pre-authentication stage should validate this state's sanity to prevent an SKD attacker from passing bad state in an attempt to compromise the pre-authenticated stage. The same problem arises when a privileged compartment accepts arguments to a privileged operation from an untrusted compartment; these arguments should also be verified to prevent compromise of the privileged compartment.

> Principle 4: Any state exported from a compartment performing session key negotiation and any untrusted arguments passed to privileged compartments should be validated.

We do not offer general techniques for verification of untrusted state and arguments. However, in our partitioning of protocol implementations, we employ pipes for inter-process communication. Although marshalling, unmarshalling, and data copies cost in performance, this mechanism provides a recipient with an RPC-like expectation of the format of the data structures it receives. These RPC-like semantics ease state and argument verification.

The session key barrier is enforced when an application switches permanently from communicating with cleartext messages to MAC'ed messages. Some protocols, such as SSL, however, can alternate between these two types of messages. In such cases, the transition between the two stages should be performed after the last cleartext message. However, doing so would require processing messages MAC'ed and encrypted with the session key during the session key negotiation stage, which risks creating session key oracles! We address this problem with Principle 7.

> Principle 5: A cryptographic protocol should not alternate between cleartext messages and MAC'ed messages.

User Privilege Barrier The user privilege barrier represents any authentication method that can be used to authenticate a peer before granting it privilege to access sensitive information owned by a particular user. This barrier prevents impersonation attacks, where an attacker exploits an application to subvert its authentication mechanism. Authentication should be performed by an unprivileged compartment that has no access to sensitive user data. The pre-authenticated stage is protected by the session key barrier, so this stage is not exposed to any SKD attacker. However, it is crucial for the integrity of the session key barrier that there be no unMAC'ed messages processed during the pre-authenticated and post-authenticated stages. Without the SKD threat, the session key is no longer sensitive information in the pre-

authentication stage, and it can be accessed directly by unprivileged code. We allow the impersonator to access the session key at this point because it is his *own* key and does not correspond to any other user's session. Successful authentication transitions the application into the next stage, the post-authenticated stage.

Today's state-of-the-art privilege-reduced applications implement the user privilege barrier as we require. However, monolithic, full-privilege applications perform authentication in a privileged compartment. The privilege-separated OpenSSH server performs user authentication in an unprivileged compartment, and then the monitor creates a new compartment with the user ID and group ID of the authenticated user. The HiStar-labeled SSL web server supports only password authentication, and the unprivileged *httpd* daemon obtains ownership of the user's labels only after the user successfully authenticates with an authentication daemon.

Some protocols authenticate peers without sending confidential data, such as passwords. For example, the SSL protocol's handshake supports only public key authentication methods. Such authentication techniques can be merged with a key exchange protocol or performed in cleartext after the key exchange protocol. Thus, the user privilege barrier can be established within the session key negotiation stage omitting the pre-authenticated stage. This optimization is encouraged, as it reduces the number of stages and compartments, and thus increases the performance of a privilege-separated application.

Authentication that requires passing sensitive data encrypted with the session key cannot be performed during the session key negotiation stage. Otherwise, the partitioning of the session key negotiation stage would require a trusted compartment to decrypt sensitive data, and this compartment would result in a session key oracle, which could be used to decrypt the user's sensitive data. Moreover, other trusted compartments would be needed to process authentication-related sensitive data, because we cannot allow untrusted code to operate with this confidential data.

The post-authenticated stage executes in a compartment with the authenticated user's privilege; it acts for the authenticated user and can access the user's data. When we transition from the pre-authenticated to post-authenticated stage, we need not kill the former stage, as it cannot be exploited, given the MAC'ed channel eliminates SKD attackers, and authentication barrier prevents impersonation attackers. Instead, we can change the privilege of the compartment used in the pre-authenticated stage to that of the authenticated user, and continue execution with the code for the post-authentication stage.

We note that for some applications, the post-authenticated stage may require further privilege separation. For example, an application may require access to a centralized database where sensitive data belonging to many users is stored. In this case, the user-authenticated compartment should be denied direct access to the database, but a trusted compartment should export access to the database. This privilege separation, reminiscent of techniques explored in OKWS [6], prevents a user from accessing other users' sensitive data.

4.2 Oracle Prevention Techniques

In the previous section, we presented a general structure for a cryptographic protocol's implementation to defeat SKD and impersonation attacks. At every stage of the aforementioned structure, there is sensitive data accessible only by trusted compartments, which in turn export privileged operations to unprivileged compartments. As discussed in Section 3.2, in all such situations, there is a risk of granting an attacker an oracle for the sensitive information.

For example, the session key negotiation stage depends on confidential session key sharing. An SKD attacker can use a trusted compartment as a decryption oracle to obtain a secret component of a session key. An impersonator may replay authentication data from another connection as an input to an authentication oracle and pass authentication as a legitimate user. Clearly, we need techniques to mitigate any oracles in these stages.

Entangle Output Strongly with Per-Session Known-Random Input Network protocols employ randomness generated afresh for every session to defeat authentication replay attacks, where an attacker replays messages eavesdropped from a user session to reestablish the past session and repeat a user's past requests. The server generates a random nonce incorporated into the session key (in the case of RSA key exchange) or a fresh private DH component (for DH key exchange) to make the session key different for every session. We can similarly employ this session randomness as a defense to counter oracles.

The output of a trusted compartment should not completely depend on untrusted input, so that an attacker will not be able to replay past input to the compartment and get the same deterministic result. Entangling the output of a privileged compartment with a trusted per-session random nonce solves this problem.

For example, Figure 4 demonstrates an approach to preventing a signing oracle in a privilege-separated OpenSSH server. We restrict the trusted monitor that implements signing with the private key to sign only session IDs that incorporate per-session random bits. A sequence of privileged operations performed by the trusted compartment ensures that the server's private DH component is indeed included in the session ID. This way,

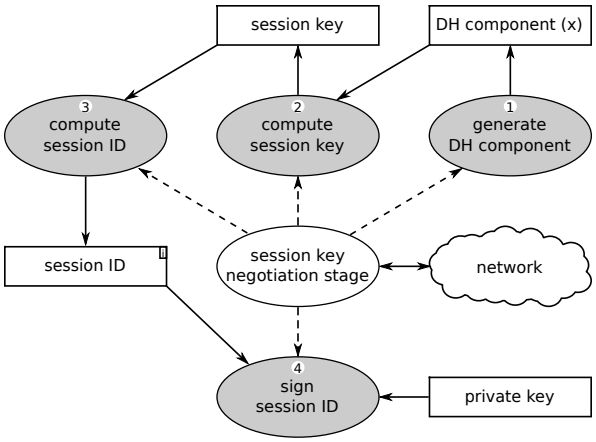

Figure 4: Prevention of private key oracle in OpenSSH server by entangling output with per-session known-random input.

we entangle the output of the RSA signing compartment/operation with trusted, per-session, known-random input. Numbers within trusted compartments in Figure 4 specify the order of their invocation, and this order should be enforced by the application.

With this oracle defense mechanism, the attacker cannot mount an impersonation attack, as every signed session ID will incorporate different randomness contributed by the server, and will thus not be valid in the context of any other session. Similarly, in order to prevent deterministic session key oracles, we make sure that the compartment generating the keys includes randomness generated afresh for every session. Moreover, per-session randomness is crucial in prevention of signature verification oracles; the data for signature verification should also incorporate it.

> Principle 6: To prevent oracles, entangle output strongly with per-session, known-random input.

In RSA key exchange in the SSL/TLS protocol, there is the potential for a deterministic session key oracle attack, where an attacker can produce a deterministic session key by suppling chosen inputs to a privileged compartment generating the key. In particular, a session key consists of two public components, per-session server and client randoms, and a pre-master secret transmitted encrypted in the server's public key [4]. When generating the session key, these components are concatenated together and hashed. The server decrypts the pre-master secret using its private key before hashing it together with the other components. If an attacker controls the server random, client random, and encrypted pre-master secret inputs to the session key generation function, he can feed data eavesdropped from a user session to the privileged compartment generating the session key and produce the key that corresponds to the eavesdropped session. We prevent deterministic session key oracles by ensuring that every server-computed session key includes a trusted server nonce produced and supplied to the compartment generating the session key by a trusted source. This way, an attacker cannot control the generated session key, as each time it incorporates a different random nonce.

Obfuscating Untrusted Input by Hashing The SSL protocol alternates cleartext *change cipher spec* messages with authenticated and encrypted *finished* messages [4]. A *change cipher spec* message signals that the sender is about to enable encryption and authentication, which will be used on all subsequent messages. A *finished* message contains a MAC'ed and encrypted hash of all previous cleartext messages received by a peer during the handshake protocol. The *finished* message ensures that these cleartext messages were not tampered with by an attacker.

To ensure that the session key barrier is enforced, we cannot process cleartext messages in the pre-authenticated stage. Instead we should process the *finished* messages within the session key negotiation stage. However, doing so requires a trusted compartment that performs session key encryption and decryption operations on behalf of untrusted code. This trusted compartment is a session key encryption/decryption oracle which can be used to decrypt user information and validly encrypt an attacker's exploits or requests.

Our oracle mitigation technique provides the required privileged operations (encryption and decryption with a session key) and avoids a session key oracle by obfuscating input data through hashing. As the *finished* message is an encrypted hash, a trusted compartment can be structured in the following way: it obtains data from an untrusted compartment, hashes the data, and then encrypts the resulting hash. A privileged operation that hashes data and then encrypts is not useful for an attacker, as the attacker's requests and exploits for the pre-authenticated and post-authenticated stages will be viewed as hashes.

As for the decryption oracle, we do not return the cleartext *finished* message to untrusted code. Instead, our trusted compartment takes the verification data from an untrusted compartment and performs verification of the *finished* message itself. The result of this verification is returned to the untrusted compartment. However, this mechanism allows dictionary attacks, where an attacker can guess the cleartext message by supplying the verification data. Again, obfuscating the untrusted validation data by hashing before comparing it with the cleartext *finished* message solves this problem. This approach fits the protocol because the *finished* message happens to be a hash of all previous handshake messages. If an attacker attempts to guess the cleartext requests, his guess will be hashed first, then compared with the original message.

The hashing that we apply to prevent both oracles al-

ready is present in the SSL handshake protocol. However, the protocol and our oracle mitigation technique use it for different reasons. The protocol requires the compression and collision resistance of a hash function, but our technique employs the hash function because of its non-invertibility. Happily for us, the hash function provides all of the mentioned properties, and can be used in both cases.

> Principle 7: To prevent oracles, obfuscate untrusted input by hashing.

Last Resort: More Trusted Code The previous oracle mitigation techniques require the availability of a random nonce or a hash function. However, for those cases in which a cryptographic protocol does not specify these functions at a point in the protocol where there is the risk of an oracle, we offer a last resort technique.

For an oracle to exist, a result of a privileged operation must return to an unprivileged compartment. It is possible to avoid the oracle by making the output privileged and restricting access to it in the unprivileged code. Although this technique helps, it is not efficient, as a new trusted compartment is required to process the result, and you may need to process the result of the new compartment in the same way. Our last resort technique may lead to a chain of trusted compartments, which increases the trusted code base and requires more auditing work. Moreover, to terminate this chain, there must be a suitable condition for applying one of the previous oracle mitigation techniques, or the last trusted compartment in the chain must not produce any output.

> Principle 8: To prevent oracles, as a last resort, add more trusted code.

4.3 Degrees of Sensitivity

Cryptographic protocols often operate on sensitive data of more than one class. As an example, one frequently occurring class of sensitive data is that which must be kept secret to ensure secrecy and integrity of data transferred within a *single* session, *e.g.,* the pre-master secret in RSA key exchange, the private DH key in DH key exchange, the session key, the per-session ephemeral RSA private key, *&c.* Disclosure of such sensitive data results in violation of the secrecy and/or integrity of sensitive data within a single session. Yet there is often another class of even more sensitive data that must remain secret in order to preserve the secrecy of user data in *many* sessions. This class includes a server's private key, users' private keys, and passwords that are reused on many servers. The secrecy of such data is vital because an attacker can use it to gain access to user data in multiple sessions by impersonating the server, or by using users' passwords to access many servers.

In a simple scenario like this one involving two classes of sensitive data—that which is critical to one session's secrecy *vs.* that which is critical to ensuring many sessions' secrecy—mixing sensitive data of both classes and code to manipulate data of both classes in the same compartment incurs warrantless risk. To see why, let's deviate from our threat model and assume that an attacker can compromise trusted compartments. Now any vulnerability in code that manipulates sensitive data pertaining to one session's secrecy can disclose sensitive data that could compromise secrecy of all sessions. Creating distinct compartments for data of differing degrees of sensitivity (and the code that manipulates it) mitigates this risk. Similarly, to prevent disclosure of one user's data to another, separate compartments should manage sensitive session-related key data for each user.

> Principle 9: A privilege-separated application should manage a session with two separate privileged compartments—one to operate with data related to secrecy of the current session, and one to manage data that preserves secrecy of many sessions.

We examine the benefits of isolating code and data in distinct compartments according to their sensitivity in detail in the context of a hardened OpenSSH implementation in Section 5.2.

5 Hardened SSH Protocol Implementation

We now demonstrate these principles for preventing SKD and oracle attacks by finely privilege-separating the implementations of the client and server sides of the SSH protocol.

Recent privilege separation and DIFC work focuses on server applications, as they are always online and can be attacked at will. We argue that the SSH client code should be considered as important as the server code. An attacker can set up a public service and provide access to it via SSH. By exploiting vulnerabilities in the SSH client implementation, the attacker can obtain users' private keys, used to authenticate them to other legitimate SSH servers. These keys allow the attacker to obtain or tamper with the user's sensitive information stored at these other SSH servers. Moreover, as the SKD attack is equally valid on both sides, server and client, protection against it is equally needed on the two sides.

Throughout this paper, the baseline OpenSSH server design we refer to is that of Provos *et al.* [10]. While this OpenSSH server implements privilege separation, it allows unprivileged code access to the session key (contravening Principles 1 and 2) and sign a session ID provided by unprivileged code (contravening Principle 6), and thus is vulnerable to SKD and oracle attacks. We show how to

partition the server more finely to prevent these attacks. But first, we finely partition the OpenSSH client, which to date has only existed in monolithic form, and is thus also vulnerable to both SKD and oracle attacks.

5.1 Hardened OpenSSH Client

The OpenSSH client runs under the invoking user's user and group IDs. Because changing the user ID to `nobody` and invoking the `chroot` system call require `root` privilege, they cannot be used here. Instead, we limit the privilege of the trusted and untrusted compartments of the OpenSSH client with SELinux policies [8], and the SELinux type enforcement mechanism in particular. SELinux policies allow us to restrict untrusted processes from issuing unwanted system calls such as `ptrace`, `open`, `connect`, *&c.*[3] Our prototype supports only password and public key authentication, and does not yet implement advanced SSH functionality (tunneling, X11 forwarding, or support for authentication agents).

Our finely privilege-separated 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, create a pseudo-terminal device, *&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 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 (represented by ovals) appears in Figure 5. The use of two distinct monitors is motivated by Principle 9.

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 to read and add public keys for approved hosts; to read/write a UNIX sockets for communicating with the private key monitor and an unprivileged process running untrusted code (described below); and to read/write a terminal device. 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 that confines it with a similarly tight policy, allowing it only to read the user's private key(s), with no access to other files, nor to create any sockets. The private key monitor shares a UNIX socket with the session monitor and only accepts requests from the latter.

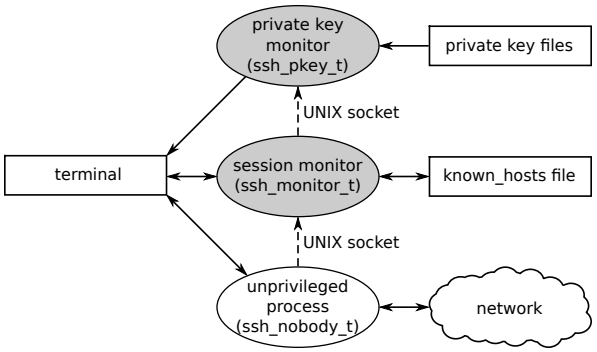

Figure 5: Architecture of privilege-separated OpenSSH client. Shaded ovals denote privileged compartments. Unshaded ovals denote unprivileged compartments. The last line in each oval denotes the SELinux policy enforced.

Session monitor
1) **DH_priv_key** = gen_DH_priv_key()
2) DH_pub_key = comp_DH_pub_key(**DH_priv_key**)
3) **sess_key** = comp_sess_key(**DH_priv_key**, srvr_DH_pub_key)
4) **sess_ID**^{i} = comp_sess_ID(**sess_key**, clnt_version, srvr_version, clnt_kexinit, srvr_kexinit, ...)
5) **sym_keys** = derive_sym_keys(**sess_ID**^{i}, **sess_key**)
6) **srvr_pub_key**^{i} = verify_srvr_pub_key(srvr_pub_key, **known_hosts_file**)
7) verify_ sign(**sess_ID**^{i}, **srvr_pub_key**^{i}, sign)
Private key monitor
1) sign = priv_key_sign(**priv_key**, **sess_ID**^{i}, user_name, service, auth_mode, ...)

Figure 6: Privileged operations performed by the two client monitors. Data in regular font is untrusted parameters provided by unprivileged compartments. Sensitive data appear in bold, and are accessible only by the monitor compartment in which they appear. $\mathbf{x}^i$ denotes that sensitive data **x** is exported to an unprivileged compartment read-only.

After creating these two monitor processes, the original SSH client process drops privilege to the *ssh_nobody_t* domain. Untrusted code runs in this unprivileged process and domain during the rest of this invocation of the SSH client. 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 sockets. The *ssh_nobody_t* domain further denies all access to the file system, allowing the unprivileged process access to the terminal device only.

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 6, 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* (sym_keys)

are the keys derived from the session key for the MAC and encryption/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 signing operations with the key. Only the session monitor may invoke these signing operations in the private key monitor (over a UNIX-domain socket), and it provides the session ID to be signed as an argument. We give more detailed explanation of the private key sign operation in the end of this section.

Session Key Negotiation Stage In a successful SKD attack on the OpenSSH client, the attacker would interpose himself as an active man-in-the-middle between the client and server, exploit the client during the SSH protocol handshake, and forward all handshake messages and post-handshake session data unmodified between the server and client. By injecting code into the client to cause the session key to be disclosed to him, the attacker would thus reveal all sensitive data transferred thereafter between the user and server over the encrypted session.

The first stage of execution for the finely partitioned OpenSSH client is the session key negotiation (SKN) stage, which allows an unprivileged compartment—with the help of the session monitor—to perform Diffie-Hellman key exchange, and thus to negotiate a session key and authenticate the server. As required by Principle 1, we restrict the SKN 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 6)—as they are essential for session key secrecy, and thus for preventing an SKD attack.

We must also prevent a *verification oracle attack* against the client 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 to which the client wishes to connect. 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 signature verification routine verify_sign() in the session monitor in Figure 6. 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 sess_ID with his *own* private key and supply his *own* corresponding srvr_pub_key, or supply a bogus sess_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 srvr_pub_key.

To prevent this verification oracle, we must not allow an unprivileged compartment (at risk for remote exploit) to provide either srvr_pub_key or sess_ID to verify_sign(). We thus perform signature verification in the session monitor, and isolate sess_ID and srvr_pub_key within the monitor, out of reach of untrusted code. In actuality, the untrusted compartment provides srvr_pub_key to the session monitor, but the session monitor validates it against the contents of the `known_hosts` file before verifying the signature. Note that sess_ID is entangled with trusted random bits generated by the client every new session, originating from the client's DH_priv_key via comp_sess_key() and comp_sess_ID(). This construction, specified by the OpenSSH protocol, implicitly applies Principle 6, which further prevents an attacker from forcing sess_ID to match that from a past eavesdropped session.

The above partitioning prevents an attacker who compromises the unprivileged compartment from mounting successful SKD or verification oracle attacks on the client.

We now turn our attention to the next steps 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. 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 re-

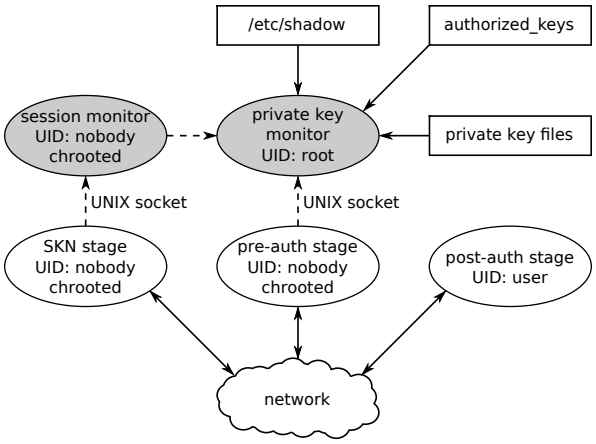

Figure 7: Architecture of finely privilege-separated OpenSSH server.

quires 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 in the client, we do not allow the untrusted compartment to directly invoke signing 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 6. Recall that the session ID has been entangled with trusted random bits generated by the client for the current session. 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 is permitted to invoke privileged rekeying operations implemented by the session monitor.

5.2 Hardened OpenSSH Server

In accordance with Principle 9, we extend the baseline privilege-separated OpenSSH server with an extra session monitor process that handles sensitive data related to a single user's session while preventing an SKD attack and both private key signing and signature verification oracles, as shown in Figure 7. The private key monitor is the original monitor process from the baseline privileged-separated OpenSSH server, which performs operations that require `root` privilege.

The session monitor, the unprivileged SKN process, and the unprivileged process of the pre-authentication stage all run in a `chroot`ed environment with an unused UID, under a restrictive SELinux policy that allows only the system calls implied in Figure 7, and prohibits all others, including dangerous ones such as `ptrace` and `connect`. The process for the post-authenticated stage runs with the UID of the authenticated user and is not restricted with any SELinux policy, as with the baseline OpenSSH server.

Session Key Negotiation Stage The session monitor implements the privileged operations required for the SKN stage, and we ensure that the pre-authenticated stage does not start unless the unprivileged compartment of the SKN stage terminates (in accordance with Principle 2). Because the Diffie-Hellman key exchange protocol is symmetric between the server and client, we implement operations 1–5 from Figure 6 in the server's session monitor just as in the client's. The SKD attack is an equally serious threat for client and server; as both parties share the same session key, an SKD attacker can compromise either party's code to disclose it.

During the SKN stage, the server authenticates itself to the client by signing a session ID. The monitor in the baseline privilege-separated OpenSSH server signs any data supplied by the untrusted compartment, thus allowing an oracle attack. A man-in-the-middle attacker can interpose himself between a client and a bona fide server and employ a signing oracle on the server to impersonate the server by producing valid signatures on session IDs corresponding to the attacker's session with the client. We prevent such attacks by constraining the private key monitor to sign only data provided by the trusted session monitor—specifically, the current session ID entangled with trusted random bits provided by the server, as shown in Figure 4, as suggested by Principle 6. The server's session monitor produces this sess_ID in operation 4 in Figure 6, just as the client's does. This signed sess_ID cannot be used to impersonate the server as it is only valid within the current session. To perform the signing operation, the session monitor calls into the privileged private key monitor and supplies the required trusted sess_ID to sign.

Pre-authenticated and Post-authenticated Stages The baseline privilege-separated OpenSSH server separates the pre-authenticated and post-authenticated stages. It performs user authentication operations such as password verification and signature validation (in public key authentication) in the monitor. However, this architecture allows an SKD attacker to compromise the password during password authentication, as it is encrypted with

the session key obtainable by the attacker. During public key authentication, the untrusted compartment supplies the data used for user signature verification, again allowing oracle attacks against user authentication. The monitor validates the signature against the session ID supplied earlier when the untrusted compartment requested the server's signature on this session ID. Thus the untrusted compartment can control the session ID used in public key authentication of the user. In order for an attacker to impersonate the client, she must provide some session ID signed by the client for the server's verification operation. It is unlikely that the attacker can force a user to sign arbitrary data with his private key. However, an SKD attacker can compromise the user's session and log its session ID and signature pair. She can then replay these data to the server's signature verification compartment. Because the server's signature verification routine does not check whether the provided session ID is valid within the current session, the verification routine will report that the client has authenticated successfully. In this way, the attacker successfully impersonates the user.

In our implementation, we fix this problem by making sure that the session ID used for signature verification is produced by the session monitor, as done in operation 4 in Figure 6, and entangled with trusted random bits provided by the server. Our SKN stage also ensures the secrecy of user passwords by thwarting SKD attacks.

Discussion: Trusted Code Base Figure 8 compares the trusted code bases of Provos *et al.*'s baseline privilege-separated OpenSSH server and our finely partitioned OpenSSH server. The latter implements two monitors, in accordance with Principle 9, and as described in Figure 7: one private key monitor that implements code required for user authentication and accessing the server's private key, and one session key monitor that contains the privileged code for processing the sensitive state for a user's session. Consider operations 1–5 in Figure 6, which are essential to protection against SKD and oracle attacks. In our partitioning, the session monitor implements these five operations, while in baseline OpenSSH, the untrusted compartment implements them.

At first glance, one might remark that our partitioning therefore incorporates *more* privileged code than baseline OpenSSH. But that assessment is flawed. Rather, the sensitive state pertaining to a user's session was incorrectly deemed non-sensitive data in baseline OpenSSH. Hence, we show baseline OpenSSH's untrusted process as shaded—notation for privileged—because it is *already* (albeit inappropriately) privileged to manipulate sensitive per-session data. Following the partitioning principles we have offered leads to the correct treatment of this data as sensitive, the creation of a new privileged compartment that can exclusively manipulate this data

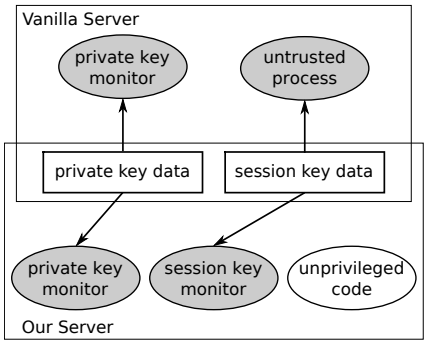

Figure 8: Relationship between privileged (shaded) and unprivileged (unshaded) code in baseline and hardened OpenSSH server implementations.

(the session monitor), and the *reduction* of privilege for all remaining code from baseline OpenSSH's untrusted process (denoted in the figure as "unprivileged code")!

6 Hardened OpenSSL Library

Toward demonstrating the generality of the partitioning princples presented in Section 4, we have also applied them to the SSLv3 and TLSv1 cryptographic protocol implementations in the OpenSSL library. Partitioning in accordance with these principles requires a fair amount of programmer effort. We thus found the OpenSSL library a particularly attractive target because it is used by a broad range of security-conscious applications, and thus allows amortizing one protocol implementation's partitioning effort over many applications. The resulting finely privilege-separated 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.

We finely partition both the client and server sides of OpenSSL. Our implementation supports the RSA, ephemeral RSA, Diffie-Hellman, and ephemeral Diffie-Hellman key-exchange protocols with both client and server authentication. The OpenSSL partitioning is in fact similar in structure to that of SSH, as these protocols protect against similar threat models. When an application invokes the SSL_accept (on the server side) or SSL_connect (as an SSL client) library call, we `fork` a private key monitor, session key monitor, and an unprivileged SKN compartment. 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. Within the SKN

stage, we apply the same principles and mechanisms as we did to OpenSSH to prevent SKD and oracle attacks. 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.

To limit the privilege of the untrusted SKN compartment and the session monitor in applications that do not run as `root`, we apply SELinux policies. Under our SELinux policy, the untrusted SKN compartment is allowed 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 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 an SKN compartment.

Our partitioned implementation of the OpenSSL library supports session caching, crucial to high-performance applications such as SSL web servers. Currently, our implementation fully supports TLSv1 session resumption via tickets [5]. For SSLv3, our implementation support an external inter-process session cache in a file or over a UNIX socket, but not as shared memory.[4]

We have tested our finely privilege-separated OpenSSL library with a number of client-side and server-side applications, including the server and client sides of stunnel, the mutt and mailx mail agents' (for IMAP and POP3 over SSL/TLS), the dovecot IMAP and POP3 server, the client and server sides of the sendmail mail transfer agent (for SMTP over SSL/TLS), and the Apache HTTPS server (versions 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.

Apache, however, required code modifications—not to protect against SKD and oracle attacks, which are entirely covered by the partitioned OpenSSL library, but to protect sensitive data *after* the SSL handshake completes. As noted above, Apache reuses worker processes to serve successive users' requests. We modified Apache to fork a new worker for each connection and kill a worker after it has served one connection.[5] With or without this unrelated application-level change, Apache 1.3.19 runs with our finely partitioned OpenSSL library as a drop-in replacement for the stock OpenSSL library. Apache 2.2.14 does, too, with the addition of a single trivial method to its I/O layer.[6]

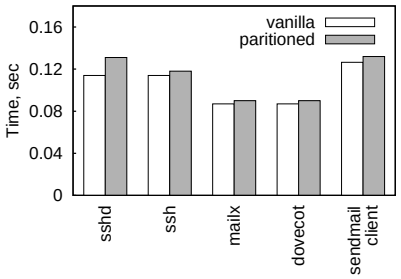

Figure 9: Latency of baseline and hardened OpenSSH 5.2p1 client/server and mailx 12.4, dovecot 1.2.10, and sendmail client 8.14.4 using baseline and hardened OpenSSL 0.8.9k library. Run on Dell desktop with 1.86 GHz Intel Core 2 6300 CPU and 1 GB RAM running Linux 2.6.30.

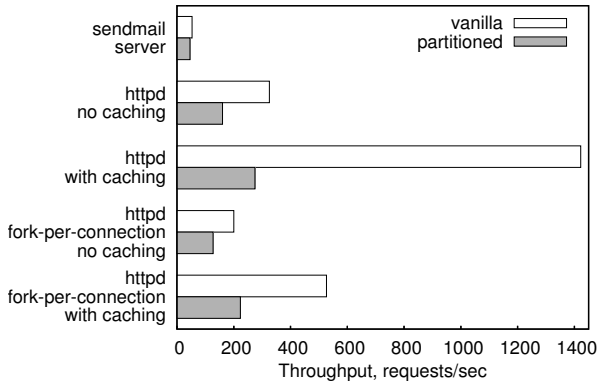

Figure 10: Throughput of sendmail server 8.14.4 and Apache web server (httpd) 2.2.14 using baseline and hardened OpenSSL 0.8.9k library. Run on Sun X4100 server with 2.2 GHz AMD Opteron 248 CPU and 2 GB RAM running Linux 2.6.32.

7 Evaluation

We now consider the cost of defending against SKD and oracle attacks in cryptographic protocol implementations. As the principles given in Section 4 demand additional isolation between code and data, and thus additional processes, performance is a concern: both process creation and context switches incur overhead. To explore the extent of these overheads, we compare the performance of the baseline OpenSSH and OpenSSL-enabled applications with that of the implementations hardened in accordance with the principles we have propounded. We consider in turn the end-to-end metrics of operation latency (important to users) and server-side throughput (important to server operators).

Figure 9 compares operation latencies for a range of applications. Each application is either client-side or server-side; in each case, the complementary remote peer runs the *baseline* cryptographic protocol implementa-

tion. All connections are made over the loopback interface to a locally running server. For OpenSSH, we report the latency of logging into an SSH server using public key authentication and running the `exit` command. The remaining applications use the OpenSSL library. For the mailx email client and dovecot IMAP server, we measure the time required for the client to connect over SSL/TLS, check for new mail, and exit. For the sendmail client, we measure the time required to connect and send a one-line email to a sendmail server over SSL/TLS. For these applications, the latency a user perceives does not increase significantly between the baseline and hardened cryptographic protocol implementations.

In Figure 10, we consider the throughput achieved by an SSL/TLS-enabled sendmail server and HTTPS server, both based on the OpenSSL library. For the sendmail server, we submit emails over SSL/TLS from multiple clients and report the maximum load the server can sustain in requests (emails) per second. For Apache (httpd), we measure the maximum load an SSL server can handle in requests per second by stressing the server with multiple clients requesting a small static page over HTTPS; clients always establish fresh SSL/TLS sessions in these measurements.

As noted in Section 6, apart from adding defenses against SKD and oracle attacks, we further modified the baseline Apache implementation to `fork` a new worker process for each new client connection, to avoid risking disclosure of one user's sensitive data to another when worker processes are reused. To distinguish the cost of additional processes introduced for defending against SKD and oracle attacks *vs.* those introduced to avoid reuse of workers, we measure the throughput of two Apache implementations: one in which workers are reused and one in which Apache `forks` a new worker for each connection. We further consider Apache's performance in two extremes of operation: when no SSL sessions are cached and when all sessions are cached. In this evaluation, we constrain HTTPS clients to use RSA key exchange when establishing an SSL/TLS session because this protocol is less computationally intensive on the server than Ephemeral Diffie-Helman key exchange, and thus better exposes the overhead of hardening.

When session caching is disabled, hardening of the SSL/TLS implementation while reusing worker processes reduces server throughput by a factor of two; this reduction increases to a factor of five when all sessions are cached. The overhead of the SKD and oracle attack defenses is masked in part by the compuational costs of the cryptographic operations required to establish a new SSL/TLS session; when all sessions are cached, though, these cryptographic operations are absent, laying bare the overhead of finer-grained partitioning.

`Fork`ing one worker per HTTPS client request incurs further overhead; throughput drops by nearly a factor of three between baseline Apache without session caching and a hardened Apache that forks one worker per client without session caching. A newly `fork`ed worker incurs many page faults as it modifies copy-on-write pages from its parent. In baseline Apache, these page faults occur only once per worker, for the first client ever served; in this per-client-`fork`ing Apache, though, they occur for every client connection.[7]

These applications based on the OpenSSL library use single-process, monolithic designs. Hardening against SKD and oracle attacks requiers 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 client uses four processes per SSH session *vs.* the two employed by the baseline privilege-separated OpenSSH server. Apart from process creation and page fault costs, anti-SKD and anti-oracle hardening incurs overhead for marshaling and unmarshaling arguments and return values between compartments connected by pipes.

8 Related Work

Provos *et al.* demonstrate a generic approach for privilege separation of applications that aims at preventing privilege escalation in case an application's unprivileged part is compromised [10]. They reduce privilege in the OpenSSH server by partitioning it into an untrusted process and a privileged monitor. Our work tackles the different goal of preventing disclosure of users' sensitive data in cryptographic protocol implementations. This goal incorporates solving the problem of privilege escalation. We extend the partitioning of the privilege-separated OpenSSH server to comply with this goal.

HiStar [15] and DStar [16] offer an approach for enforcing privileges on compartments with labels and DIFC. DStar extends this approach to a distributed environment without fully trusted machines. Zeldovich *et al.* partition an SSL server to mitigate the effect of a compromise of any single compartment and prevent disclosure of user data. However, as we have described, it is possible to disclose users' sensitive data from the SSL server using recent SKD and oracle attacks. The lack of partitioning of the SSL protocol allows these attacks. Our work is complementary to work on DIFC systems, as they are only privilege enforcement mechanisms, and we provide guidance on how to structure code in cryptographic protocols.

Wedge [2] offers another privilege enforcement tool, *sthreads*, for fine-grained partitioning of applications on Linux. An sthread is a thread designed for implementing compartments with restricted memory and file descriptor permissions. When creating an sthread, a programmer explicitly specifies which memory regions and

file descriptors the parent process shares with the child. The other sthread privileges are restricted through an SELinux policy. Wedge incorporates *crowbar*, a development tool for identifying the required memory permissions of a compartment when partitioning a monolithic application.

The work on Wedge was first to identify the problem of partitioning the code for a cryptographic protocol. While partitioning an SSL-enabled Apache Web server, Bittau *et al.* discovered the SKD attack. An ad hoc mechanism was implemented to prevent this attack, but Wedge does not offer a general solution for the SKD and oracle attacks. Rather, the work mainly focuses on the privilege enforcement primitives (sthreads) and partitioning tools (crowbar). By contrast, we offer partitioning principles that defeat the SKD and oracle attacks, and we believe these principles are general enough to use with many cryptographic protocols.

OKWS is a toolkit for building secure Web services [6]. They employ the same privilege enforcement mechanisms as privilege-separated OpenSSH, in particular the `nobody` user ID and a `chroot` system call, to isolate distrusted Web services from harming the system they are running on and each other. Krohn demonstrates that the performance of such privilege-separated Web services can be comparable to that of non-privilege-separated systems.

Our partitioning principles and mitigation techniques might also find applicability in a capability-based systems such as KeyKOS [3] and EROS [12]. These systems can restrict privilege of compartments in partitioned applications; however, they do not define how to partition, or more importantly how to use capabilities to produce secure partitioning schemes.

9 Conclusion and Future Work

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

We have offered principles to guide programmers in partitioning cryptographic protocol implementations to defend against SKD and oracle attacks. In essence, following these principles reduces the trusted code base of an application by correctly treating session key material and oracle-prone functions as sensitive, and limiting privilege accordingly.

We have demonstrated that these principles are practical by newly partitioning an OpenSSH client and extending the partitioning of a privilege-separated OpenSSH server. Further experience with the OpenSSL library suggests they may generalize to other cryptographic protocols; they are broadly targeted at protocols that negotiate session keys and perform common cryptographic operations. While 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 on a busy SSL-enabled Apache server, reducing the new SSL/TLS session handshake rate by half. We intend in the future to apply Bittau's checkpoint-and-restore optimizations [1] to avoid the further cost of `fork`ing one worker process per client request in our current unoptimized hardened OpenSSL implementation.

We discovered the attacks we have presented by manual study of the SSH and SSL/TLS protocols and their implementations. We intend in future to explore tools that use static and dynamic analysis to ease discovery of such vulnerabilities in cryptographic protocol implementations.

Acknowledgements

This research was supported in part by a Royal Society-Wolfson Research Merit Award and by gifts from Intel Corporation and Research in Motion Limited. We thank Andrea Bittau, our shepherd Mohammad Mannan, and the anonymous reviewers for comments that improved the paper. We further thank Andrea Bittau for sharing code for his checkpoint-and-restore server performance optimizations.

References

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

[2] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI*, 2008.

[3] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[4] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, January 1999.

[5] P. E. J. Salowey, H. Zhou and H. Tschofenig. Transport layer security (tls) session resumption without server-side state. RFC 5077, January 2008.

[6] M. Krohn. Building secure high-performance web services with okws. In *USENIX*, 2004.

[7] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[8] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX (Freenix Track)*, 2001.

[9] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, pages 18–18, 2003.

[10] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

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

[12] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *SOSP*, 1999.

[13] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM TOCS*, 25(4):11, 2007.

[14] T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture. RFC 4251, January 2006.

[15] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[16] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.

Notes

[1] While we did not implement these two attacks, we present analysis of the protocols and implementations demonstrating they are possible.

[2] While space limits us to illustrating these attacks and defense principles in the context of SSH and SSL/TLS, we have found they apply equally to IPSec, CRAM-MD5, and other secure protocols.

[3] In case SELinux is not available, it is possible to `ptrace` a process and control its system calls. There are also alternatives to SELinux such as Systrace [9]. Finally, it is possible to isolate untrusted processes with `root`-privileged `chroot` and `setuid` system calls.

[4] SELinux policies allow us to deny the SKN compartment and session monitor access to a file-based or UNIX-socket-based external cache, but 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.

[5] 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.

[6] This method allows the OpenSSL library to learn the underlying file descriptor associated with an active SSL session.

[7] We note that `fork`ing one worker per client request is a naive approach to inter-user isolation, known to perform poorly. One technique that offers such isolation at significantly less performance cost is that of *checkpoint-and-restore* for workers, as proposed by Bittau [1]. In this approach, a snapshot of each worker process is taken in pristine state before handling any user requests, and after each user's request, the memory image of the worker is restored by a trusted monitor process to this pristine state. In this work, we have been concerned with SKD and oracle attack defenses, which are complementary to inter-request isolation within a worker. Combining Bittau's techniques with our own would yield a higher-throughput HTTPS server.