A Linux kernel cryptographic framework:
Decoupling cryptographic keys from applications

Nikos Mavrogiannopoulos
COSIC/ESAT – IBBT
Katholieke Universiteit Leuven
nikos@esat.kuleuven.be

Miloslav Trmač
Red Hat
mitr@redhat.com

Bart Preneel
COSIC/ESAT – IBBT
Katholieke Universiteit Leuven
preneel@esat.kuleuven.be

Categories and Subject Descriptors
D.4.6 [Operating systems]: Security and protection—Cryptographic controls

ABSTRACT

This paper describes a cryptographic service framework for the Linux kernel. The framework enables user-space applications to perform operations with cryptographic keys, while at the same time ensuring that applications cannot directly access or extract the keys from storage. The framework makes use of the higher privilege levels of the operating system in order to provide this isolation. The paper discusses the relevant security requirements and expectations, and presents the design of the framework. A comparison with alternative designs is also provided.

1. INTRODUCTION

There are many types of attacks against software applications, such as code injection, control flow modification [23, 1], etc., attributed to software vulnerabilities. Their elimination has been proven to be a Sisyphean task and today, software vulnerabilities are mitigated using a combination of defensive programming [21] and retroactive patching. As a result, there is a risk of an attacker ‘taking control’ of an application exposed to a public network (such as a web server), maybe for some limited time until the vulnerability is known and the system is updated.

The cost of such compromise may be significant for the stakeholders. One example are on-line shops today that typically use TLS [10] to authenticate to the client and setup a secure communications channel. The secure channel is setup using a server certificate and a corresponding private key. Because in typical web server applications the private key resides in the same address space as the server, its compromise is equivalent to a compromise of the private key. Thus, an adversary with the ability to compromise the web server could retrieve the private key and obtain the ability to masquerade as the original shop to its clients. This would be possible even after such a compromise is no longer possible (e.g., caused by an automatic software upgrade, or by installing a newer operating system). This might be catastrophic for the reputation of the shop.

A defense mechanism for the above attack is decoupling cryptographic keys from the software that uses them. This is typically achieved using hardware security modules or smart cards [22], that store the cryptographic keys in a medium that allows operations without exposing them. That approach effectively raises the protection of cryptographic keys from the cost of an application compromise, to the cost of compromising a hardware module. However, there is a possibility to increase the protection levels without involving new hardware. Operating systems (OS) are typically designed with security in mind [14], and provide protection mechanisms such as segregation of the OS with user applications and isolation of independent processes. These mechanisms can be used to decouple cryptographic keys from applications, providing an additional layer of security to a system where hardware security modules are not practical or are too costly (e.g., low-cost servers, or mobile phones).

Today, frameworks offering isolation of cryptographic keys from applications exist [19, 18], based on the separation of processes and the access control mechanisms offered by the OS. In this paper we present a cryptographic framework that decouples cryptographic keys from applications based on the separation between the OS and its processes. Apart from the description, we discuss the difficulties this approach imposes as well its advantages and disadvantages comparing to an alternative design. Moreover, in order to ensure the safety of the stored keys, we base the design of the framework on principles that delegate the protection of stored keys to the strength of the supported algorithms.

The next section provides a background on existing work. In Section 3 we formalize the threats on current operating systems and set the basic requirements for our framework. In Section 4 we describe our framework on a high level and present its architecture. A security analysis is provided in Section 5. In Section 6 a performance analysis is provided and the following section discusses the differences between our approach and alternatives. Section 8 concludes.

2. BACKGROUND AND RELATED WORK

Hardware security modules (HSM) and smart cards are the main medium used today for decoupling cryptographic keys from software and are often required by governmental stan-
3. MODEL AND REQUIREMENTS

Our model of an OS is based on the Linux kernel design. In that, software is comprised of an OS running on a different protection ring\(^4\) than normal applications. This requirement ensures that normal applications cannot access data available to the OS.

A user called administrator, is able to directly control the OS executing in the highest protection ring and there is at least one user that operates solely on a lower protection ring. The applications running on a system are assumed to be running with the privileges of a normal user. We use the common terms user-space to reference the lower privilege rings and kernel-space for the higher privilege ring the OS operates.

### 3.1 Threats

\(^2\)A cryptographic API that provides logical separation between keys and operations. It is mainly used to access operations on hardware security modules and smart cards.

\(^3\)Software that provides emulation to a security module but provides no isolation between cryptographic keys and the application was not considered in our study.

\(^4\)We use the term protection ring to assume a strict separation of data and code between different access control levels.

In this model of an OS, applications operate in user-space and utilize cryptographic keys. We assume that application vulnerabilities might exist for some time until the system is patched. Those give adversaries a time window to execute malicious code at the same ring as the applications running in the system. Thus our main threats are:

- An adversary that has taken control for limited time of a user-space application and is trying to recover the protected keys; the adversary might be collaborating with another legitimate user;
- An adversary that is a legitimate system user and wants to extract raw keys; in contrast to the former adversary, this one is not restricted by time.

We want to protect keys in the presence of such adversaries. It should be noted that in this model, there is no distinction between operating system access and administrative access. This is due to the design of the modelled system.

### 3.2 Requirements

<table>
<thead>
<tr>
<th>Feature</th>
<th>Supported Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key generation</td>
<td>Symmetric keys and Private/public key pairs</td>
</tr>
<tr>
<td>Encryption and</td>
<td>RSA PKCS #1 v1.5, RSA PKCS #1 OAEP, AES-CBC, AES-CTR,</td>
</tr>
<tr>
<td>Decryption</td>
<td>AES-ECB, 3DES-CBC, 3DES-CTR, 3DES-ECB, CAMELLIA-CBC,</td>
</tr>
<tr>
<td></td>
<td>CAMELLIA-CTR, CAMELLIA-ECB, RC4</td>
</tr>
<tr>
<td>Signing and veri-</td>
<td>DSA, RSA PKCS #1 v1.5, RSA PKCS #1 PSS, HMAC-MD5,</td>
</tr>
<tr>
<td>fication</td>
<td>HMAC-SHA1, HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384,</td>
</tr>
<tr>
<td></td>
<td>HMAC-SHA-512, MD5, SHA1, SHA-224, SHA-256, SHA-384,</td>
</tr>
<tr>
<td></td>
<td>SHA-512</td>
</tr>
<tr>
<td>Key wrapping</td>
<td>AES-RFC3394, AES-RFC5649</td>
</tr>
<tr>
<td>Key derivation</td>
<td>Diffe-Hellman as in PKCS #3</td>
</tr>
</tbody>
</table>

Table 1: Features and supported algorithms.

A list of high level requirements for the framework concerning the cryptographic keys to be protected, follows.

1. Users should be allowed to perform cryptographic operations using the keys, but the key material should not be extractable;
2. Keys should be protected from all known side-channel attacks;
3. All cryptographic material such as keys must not be accessible from applications that use them, or from any other application, either directly or through a process memory dump;
4. Keys should be transferable to other systems in a secure way.

Our requirements for the features on the implementation were set to include the required functionality for TLS 1.2 \(^1\) protocol implementation, and for its interface to be translatable to the PKCS #11 API. The operations and algorithms implemented are summarized in Table 1.

### 4. ARCHITECTURE

Our framework, named NCR, is implemented as a self-contained Linux kernel module that provides access to cryptographic...
operations. It supports a synchronous API for communication with the user-space based on the `ioctl()` system call. We emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the OS and user-space as shown in Figure 1. Our goal is to completely decouple cryptography from the processes that use it.

![Figure 1: High level architecture](image)

### 4.1 Usage of keys

Applications should be allowed to perform operations using a set of keys in a way that only the outcome will be available to them. For this reason keys should be accessible and referenced as abstract objects, without exposing their contents. However, not all keys should be treated the same. Security protocols, typically operate using long term keys, used for a key exchange that outputs short term keys valid only for the current session (session keys). Typical cryptographic protocols such as the Internet security protocols [10, 15], satisfy the known session keys property of [4], which ensures that the knowledge of past session keys does not affect protocol security. Thus, in order to avoid protocol specific key derivation algorithms being present in the framework, we explicitly distinguish keys as ‘long term’ and ‘session’ keys. This distinction strikes a balance between pragmatism and need for key protection. Operations on long term keys will be protected by the proposed framework, while session keys will be generated and used in user-space. The distinction is not arbitrary. Session keys are valid for limited time and protect data in the application address space. Hence, if one is able to extract those keys during their validity period, it is not unrealistic to claim that the data they protect could also be extracted.

### 4.2 Framework components

Our design consists of two basic components, one is the ‘cryptographic backend’ and the NCR component that provides operations to user-space applications. Each component is described in detail in the following paragraphs, and a high level interplay of the components is shown in Figure 2.

#### 4.2.1 Cryptographic backend component

This component of the design is based partially on the existing Linux kernel asynchronous symmetric cipher and hash API, and partially on libtomcrypt and libtommath. The latter libraries were used to implement the asymmetric cryptography backend that is not part of the Linux kernel cryptographic API. The libraries were modified for the Linux kernel and were used for the implementation of RSA, DSA and Diffie-Hellman algorithms. The Linux random number generator was used to provide randomness.

The main purpose of the cryptographic backend component is to provide the main NCR component with a set of algorithms to support the required operations as in Table 1.

![Figure 2: Component view of the framework. The entry and exit points for cryptographic keys are marked with double arrows.](image)

#### 4.2.2 NCR component

The NCR component contains the main implementation of the provided features. It operates on key objects that are only available within the component. The key objects are marked on generation or import with policy flags that indicate exportable status, wrappable status, or wrapping status. This approach allows for key objects of a gradual security level. The highest being the keys that have no policy flags set and thus cannot be exported nor wrapped with another key, and the lowest being keys that are marked as exportable. We refer to keys with no policy flags as safe keys.

To enforce the security flags throughout the component, key objects can enter and leave the component through specific points only, the ‘Key storage’ and possibly the ‘Key wrapping’ subsystem as shown in Figure 2. Keys marked as exportable can also use the ‘Key export’ subsystem. The security flags are checked and enforced by the above subsystems, which also ensure that keys cannot be decoupled from their flags.

**Key objects.**

A mandatory access control mechanism is used to restrict access to cryptographic keys. Each key object on creation is

---

5 `ioctl()` is a primitive communication interface between kernel and user-space. It typically works by allowing the caller to specify the operation and a structure with the operation input data.

6 Key wrapping refers to the technique of encrypting a key with another.
given an owner and a group and the access policy is being applied by the subsystem that is performing the operations, the abstract session interface. The usage of a mandatory access control instead of the familiar in Linux discretionary has the advantage of preventing the transfer of keys from users under some outsider's control to other unauthorized users of the system.

Abstract session interface.
This interface provides a generic way for the diverse supported algorithms to be used. The algorithms operate on input data and provide encryption, decryption, signature generation, signature verification as well as message digest operations. These operations are mapped to three actions, or ioctl() calls, being NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL. The NCRIO_SESSION_INIT action initializes the algorithm in the proper mode, i.e., encryption mode or decryption mode, and associate the operation with a key object. Subsequent NCRIO_SESSION_UPDATE actions operate on input data, i.e., by hashing or encrypting, and might generate output data depending on the operation. The NCRIO_SESSION_FINAL will terminate the operation and if required output data (e.g. in signature generation). To avoid the need of multiple system calls for simple operations, i.e., encryption of a given data block we introduced NCRIO_SESSION_ONCE, which combines the functionality of the above operations in a single system call.

Key wrapping and unwrapping.
The key wrapping subsystem allows transferring keys securely to other systems by encrypting a key with another. The subsystem accepts key encryption keys if they are marked as wrapping keys. The marking of a key as wrapping or unwrapping is a privileged operation (requires administrative access). An integrity check is performed by all the algorithms supported by the subsystem to ensure that random data cannot be unwrapped to a valid key. The type and associated algorithm with the key are included with the wrapped data to distinguish between different keys. The calls for wrapping and unwrapping keys are NCRIO_KEY_WRAP and NCRIO_KEY_UNWRAP. Because this operation separates the keys from their associated flags, it is a privileged operation.

Key storage.
A difference with user-space security module frameworks is that kernel-space provides no ability of data storage. For this reason we adopted a design where the storage is delegated to the user-space applications. The protection of safe keys is being done by encrypting and authenticating everything with a system-wide master key. The keys, meta-data such as flags and an access control list are encrypted and authenticated, and handed out as binary data to user-space. Hence, the key storage facility – provided by the framework’s user - is agnostic on the actual keys it stores. It is not able to recover any key, and can only pass it to kernel-space for loading. No naming scheme is imposed by the framework and the user can adopt any convenient scheme for the stored key objects. The corresponding calls for storing and loading keys for storage are NCRIO_KEY_STORAGE_WRAP and NCRIO_KEY_STORAGE_UNWRAP.

System-wide master key.
As previously discussed a master key is required to allow the key storage subsystem to operate. The only requirement for this master key is that it has to be loaded by a privileged user – the administrator. In a typical deployment we expect that the key will be provided either as part of the boot process or protected using the filesystem access control mechanism

It should be noted that the overall system security is only as strong as this master key.

Key derivation.
Unlike the ciphers and other algorithms supported by the Abstract session interface, the Diffie-Hellman key exchange, cannot fit the NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL formality of operations. To solve that issue, PKCS #11 defines a way to operate it in software by using a key derivation API, which we adopt. This key derivation API is a single action NCRIO_KEY_DERIVE that accepts an input key, an output key and parameters.

Key generation.
This subsystem supports two actions NCRIO_KEY_GENERATE and NCRIO_KEY_GENERATE_PAIR. The former action generates a random key of a given bit size to be used in MAC and symmetric ciphers. The latter generates a public and private key pair suitable for an algorithm such as RSA, DSA or Diffie-Hellman. All generated keys are tied to a single algorithm, and special parameters are allowed to fine-tune the key generation, i.e., by specifying bit lengths for symmetric keys, or prime and generator for Diffie-Hellman keys etc. Special flags can be given to keys during generation time. They are split in operation and policy flags as in Table 2. However, not all combinations are allowed, to prevent interactions between different algorithms (e.g., prevent a key to be used for RSA signing and encryption).

Key Import and Export.
Keys that are explicitly flagged as exportable can be exported using the NCRIO_KEY_EXPORT action. Symmetric keys are exported in raw format, and public and private keys encoded using ASN.1 DER encoding rules.

5. SECURITY ANALYSIS
In this section we discuss how this implementation defends against attacks on cryptographic key isolation frameworks and side-channel attacks. In Appendix A we demonstrate through a human-directed proof how a particular set of security objectives, based on our requirements, are met, in the context of a TLS server application.

5.1 Cryptographic keys protection
Several cryptographic APIs that enforce isolation of keys, have issues that result in modification of key attributes [7, 6] by an adversary. We counter those issues by enforcing

Storing the system key in a hardware security module, when available, might seem to increase the defenses of the framework even against an adversary with administrative access. However, although such protection would prevent him from recovering the storage key, it would allow him to recover all the locally stored long-term keys. Because we expect long-term keys to be locally stored, the framework would not protect against this adversary.
Table 2: Flags used to mark keys. More than one flag may be applicable to a key.

<table>
<thead>
<tr>
<th>Operation flags</th>
<th>Policy flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>Wrapping</td>
</tr>
<tr>
<td>Decryption</td>
<td>Unwrapping</td>
</tr>
<tr>
<td>Signature</td>
<td>Wrappable</td>
</tr>
<tr>
<td>Verification</td>
<td>Exportable</td>
</tr>
</tbody>
</table>

- **Operation flags**
  - Encryption: The specified key can be used only for encryption.
  - Decryption: The specified key can be used only for decryption.
  - Signature: The specified key can be used for signing only.
  - Verification: The specified key can be used for verifying signatures only.

- **Policy flags**
  - Wrapping: The specified key can be used for wrapping other keys.
  - Unwrapping: The specified key can be used for unwrapping other keys.
  - Wrappable: The key can be wrapped by other keys.
  - External: The key was not generated within the framework.
  - Exportable: The key can be exported from the component.

Keys and attributes to be handled as a single unit in all operations. When for example, keys are exported from the framework, i.e., for storage, the exporting function will authenticate the data to prevent modifications. This authentication depends on the system key, preventing attacks as in [6].

5.2 Safe keys
Safe keys have no policy flags and only allow cryptographic and storage operations. The supported ciphers are known to be secure against chosen-ciphertext and chosen-plaintext attacks, and the signature algorithms against adaptive chosen-message attacks [11]. This ensures that cryptographic operations alone will not reveal the key. The storage operations as discussed above encrypt and authenticate the key itself, as well as all metadata using the system’s master key. This way unauthorized modification or decryption by all types of adversaries is prevented. In addition to the file system’s access controls that will be used to store the key, the storage key format includes and access control list that specifies the users allowed to unwrap it. Thus, an adversary that has successfully attacked one user account can not use it to load another user’s keys.

5.3 Wrappable keys
Wrappable keys cannot be exported in raw but have to be encrypted by another designated keys available to the framework. It is used as a mechanism to allow transfer of keys to other systems in a secure way, i.e., without revealing them in the process. In [7] Clulow describes several attacks to PKCS #11 wrappable keys which are countered by making wrapping and unwrapping privileged operations. To defend against a (stronger) adversary that might obtain and modify such keys, we do not allow arbitrary wrapping mechanisms but only algorithms that include an integrity check [12].

5.4 Exportable keys
Keys marked as exportable cannot be protected by this framework. They should be considered as having the same security level as any keys available to user-space.

5.5 Side channel attacks
Adversaries of such a framework have incentive to take advantage of side-channel attacks such as the ones described in [3, 17], to use timing and other information obtained by operations in order to extract the raw keys. For the purposes of this project we implemented the counter measures for the RSA algorithm, known as RSA blinding [17]. To counter timing attacks in the Linux kernel symmetric cipher subsystem, constant time ciphers should be added. This was not performed as part of our prototype but would be required for a real-world deployment.

6. PERFORMANCE ANALYSIS
In this section we show that the performance cost of decoupling of cryptographic keys from user-space processes. We compare the NCR framework (/dev/ncr) against a key isolation framework based purely in user-space. Moreover we include performance results of non-isolation frameworks such as the Linux port of the OpenBSD framework [16] (/dev/crypto), and a user-space implementation of the AES. The latter two would reveal the performance “loss” by the usage of a key isolation framework.

For the benchmark we created a utility that submitted requests for encryption on each framework of chunks of data with different size and a fixed key. The AES cipher in CBC mode, and the NULL (dummy) cipher were used in our tests. The same implementation of AES was used in all frameworks under test, and all were tested on the same hardware and Linux kernel version 3.0.0.

For a submission of a chunk of data for encryption, the /dev/crypto interface requires 3 system calls, the same as the NCR framework. However NCR allows for submission of requests using a single system call, the NCRIO_SESSION_ONCE. We called the latter ‘/dev/ncr single’ variant and included it to our benchmark to assess the optimizations obtained by reducing the number of system calls.

The only existing user-space key isolation framework that was available for the same system was LSM [18], but its design is based on the sockets API, requiring TCP/IP processing and data copying between the processes. A comparison with the NCR or the /dev/crypto frameworks that are zero-copy would be negatively affected by such overhead. For that reason we implemented a user-space benchmark that would operate using the zero-copy principle. It involves two applications sharing memory and using semaphores for synchronization of operations.

In Figure 3 we show a comparison of all frameworks using the NULL cipher and in Figure 4, a comparison using the AES cipher. Since the NULL cipher performs no operations, the former results reveal the overhead of the framework and its maximum performance limit. Comparing to the /dev/crypto interface we see that the NCR implementation has no noticeable performance difference. This illustrates the fact that decoupling of cryptographic keys from processes does not introduce any performance cost to a kernel framework. We can also see from the performance of NCR with a

---

8The Linux port of the OpenBSD framework was selected because of its performance advantage [8] over the native Linux cryptographic API (AF_ALG).

9By zero-copy we mean that no data are copied from the memory of the requesting application to memory of the serving application or kernel.
7. COMPARISON WITH ALTERNATIVE APPROACHES

The differences between different framework designs, such as based on the separation of processes or the separation with the OS are not limited to performance. A separation framework might have additional goals from the requirements we set in Section 3. For example the CNG framework [19] abstracts the usage of hardware security modules, smart cards or software isolation using the same interface. This is not easily implemented in a framework residing in the OS, since functionality such as access to smart cards is not typically available. Portability is another concern. A cross-platform solution cannot rely on a particular kernel interface.

On the other hand, there are advantages of an OS based framework. This is mainly performance as illustrated in the previous section, as well as efficient access to cryptographic accelerators [16] and the secure bootstrapping property [2]. Cryptographic accelerators, when not implemented in the CPU instruction set are only available through a kernel-space driver. A kernel-space cryptographic framework is in advantage of the driver directly without the cost of context switches. In a secure bootstrapping architecture each system module verifies the modules it loads (or uses). For example, the system loader would verify the OS and the OS would verify the loaded applications and so on. Having the cryptographic framework in the OS allows for a clear and simple design since the OS would already contain the functionality needed to perform signature verification, i.e., the algorithms, and this functionality has already been verified by the system loader as part of the OS. We provide a summary of the advantages and disadvantages in Table 3.

8. CONCLUDING REMARKS

In the previous sections we have presented the implementation of a cryptographic framework for the Linux kernel. The main goal of the framework is to decouple cryptographic keys from the applications using them, by confining the keys to kernel-space. This confinement is achieved through the enforcement of rules on the way keys are generated and utilized in order to protect against known attacks on cryptographic APIs. The API is practical, in the sense that it can be used to fully implement cryptographic protocols such as TLS 1.2 [10], and can be used to provide services under the PKCS #11 API. It is available to all system users and the

---

Table 3: Comparison of the different approaches for cryptographic key isolation.

<table>
<thead>
<tr>
<th></th>
<th>Kernel-space framework</th>
<th>User-space framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Combination with HSMs</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Secure bootstrapping</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Efficient access to hardware cryptographic accelerators</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

---

For example parsing PKCS #15 structures or PKCS #11 modules requires an addition of large subsystems to the OS.
administrator role, although required, is limited to maintenance aspects of the framework, such as loading the master keys, and the generation of wrapping keys.

Our performance evaluation showed that the throughput of the framework is equivalent to another kernel-based cryptographic framework that does not decouple keys from applications, indicating that there are no issues in the design that hinder performance. Moreover we show that alternative designs consisting of user-space only components will have performance degradation, caused by the inter-process communication overhead. Thus we believe that our software implementation provides a practical additional layer of protection for cryptographic keys.

9. ACKNOWLEDGMENTS

The authors would like to thank Steve Grubb for the inception of this project, Jan Filip Chadima for implementing a user-space interface to the NCR framework, Stephan Müller, Tomáš Mráz, Óscar Repáraz, Andreas Pashalidis, Nessim Kisserli, Jan Cappaert and the anonymous referees for valuable comments and feedback. This work was supported in part by the Research Council K.U.Leuven: GOA TENSE, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project.

10. REFERENCES


APPENDIX

A. APPLICATION TO TLS KEY PROTECTION

This section describes in detail how the proposed framework could be used to protect the private keys used to identify a TLS [10] server. The approach is inspired by the security target format defined by the common criteria standard [13].

A.1 Problem definition

The TLS 1.2 protocol uses a public-private key pair to identify the server and avoid man-in-the-middle (MITM) attacks: the public key is included in a certificate signed by a trusted third party and provided to the client as part of the TLS connection establishment process. The protocol can use either a RSA key pair, or a Diffie-Hellman key pair. Although the algorithm and key exchange method choice affects internal operation of the TLS handshake, for our purposes we can treat both kinds of keys in the same way: the private key must be available to the TLS server, and disclosure of the private key to an adversary allows him to successfully mount a MITM attack impersonating the legitimate server.

As Section 4.1 describes, protection of session key material does not provide a meaningful improvement in security, and an adversary is necessarily able to mount a MITM attack, or an equivalent information disclosure attack, during the time the TLS server is under the adversary’s active control. We therefore consider the following components of the problem of protecting a TLS server.

T1 Threat to be mitigated: An adversary could impersonate the TLS server for an unbounded time. This is based on the capabilities of the first adversary described in Section 3.1.
T2 Necessary functionality: The TLS server must have enough access to the private key identifying the server (referred to simply as “the key” from now on) to be able to serve legitimate requests.

We also assume that the TLS private key is managed by the administrator.

A.1.1 Security objectives

To handle the above-described problem, we have chosen a set of specific objectives that we want the TLS server deployment to meet fully.

O1 The TLS server process can remain compromised only for limited time.
O2 Only the TLS server process and processes started by the administrator can perform operations using the key.
O3 An adversary who controls the TLS server can not export the key to a different computer.
O4 Processes of the administrator are trusted not to reveal the key.
O5 The TLS server has enough access to the key to be able to perform RSA decryption, RSA or DSA signing and Diffie-Hellman shared secret establishment, depending on TLS key exchange method.

These objectives constitute one possible approach to handling the problem set out in Section A.1: O1–O4 are together designed to mitigate threat T1, and O5 to provide functionality required by T2. Whether, and how effectively, the chosen objectives handle the problem set, is a question that must be assessed by human judgement.\(^{11}\)

A.2 Requirements and policies

Finally, we design a set of requirements on the cryptographic module described in this article and policies that must be enforced by the server’s administrators by system configuration or setting up human processes. The requirements and policies were chosen in a way that allows us to demonstrate that all objectives were met in a semi-formal way.

We use the following requirements which are met by the module and were verified by manual inspection of the source code:

R1 The only ways to make a pre-existing key available as a /dev/ncr key object is to load it using the key storage component, unwrap it, and to import it from a plaintext representation.

R2 The wrapping and exportable flag restrictions are correctly enforced.

R3 No other user than administrator is able to set the wrapping or exportable flag of an existing key.

R4 No user other than administrator is able to make a copy of an existing non-exportable key with one or both of the flags set if they were not set on the original without having access to a key with the wrapping flag set.

R5 A key wrapped for storage identifies users allowed to unwrap or to perform operations using the key, and this restriction is correctly enforced.

The following policies are the responsibility of system administrators:

P1 Any TLS server compromise is discovered and fixed in a bounded amount of time.

P2 The TLS server runs under a dedicated user account.

P3 The key, wrapped for storage, is stored in a file that is accessible to the user account of the TLS server.

P4 The key is not wrapped, using the storage master key or any other key, in files that are accessible to any other user account, except perhaps for the administrator.

P5 The key wrapped for storage in P3 above identifies the user account of the TLS server, and no other account, as allowed to unwrap the key.

P6 The storage wrapping master key is available at most to the kernel and processes running as administrator on the TLS server computer.

P7 Processes of the administrator are trusted not to reveal the key.

P8 The key wrapped for storage in P3 above does not have the exportable flag set.

P9 The TLS server does not have access to any key with the wrapping flag set.

P10 The key is not stored in any file unencrypted, nor encrypted with a key available to processes not running as administrator.

A.3 Meeting security objectives

We can now demonstrate that the requirements and policies from Section A.2 are sufficient to implement the objectives we have chosen in Section A.1.1:

We start with two auxiliary claims:

L1 If a key does not have the wrapping or exportable flag set, then the TLS server can not wrap using the key, or export the key, respectively: ensured by combination of R2, R3, R4, P9.

L2 An adversary who controls the TLS server can not extract the key material to user space:

- The key is not available exported in plaintext: P8, L1, P10.
- The key can not be extracted through the key wrapping mechanism: P9, L1.
- The key can not be extracted using the storage wrapping mechanism: P6.

Moving on to the objectives:

- O1 is implemented by policy P1.
- To demonstrate O2 is implemented, consider:
  - Only the TLS server process and processes started by administrator can load the key into a /dev/ncr key object: R1, P4, P2, and P10.
  - An adversary who controls the TLS server can not make the key object available to other user accounts on the same computer: R5, P5, P9, L1.
  - An adversary who controls the TLS server can not extract the key material to user space: L2.
- To demonstrate O3 is implemented, consider:
  - The adversary can not extract the key material to user space: L2.
  - The adversary can not use the key wrapping mechanism to transfer the key material: P9, L1.
  - The adversary can not use the storage wrapping mechanism to transfer the key material: P6.
- O4 simply becomes policy P7.
- O5: The key is made available to the server by P3.

\(^{11}\)For example, the above set of objectives ignores ways to mount threat T1 without attacking the computer on which the TLS server runs, such as discovering the private key from the public key by brute force attacks, or obtaining a copy of the key from a key escrow system.