Key Management for Transcrypt

by

Abhijit Bagri

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

May 2007

Key Management for TransCrypt

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Abhijit Bagri

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY,KANPUR

May 2007

CERTIFICATE

It is certified that the work contained in the thesis entitled "*Key Management for TransCrypt*" by *Abhijit Bagri* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

Dr. Rajat Moona
Department of Computer Science
& Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016.

[illegible]

Dr. Dheeraj Sanghi
Department of Computer Science
& Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016.

[illegible]

Abstract

With data storage and processing snowballing into a necessity from being an efficient part of any business process or organization, the need for securing storage at various degrees of granularity is gaining considerable interest. The challenge in designing an encrypted filesystem stems from balancing performance, security perception, ease of usage and enterprise level deployability. Often, the most secure solutions may not even be the best solution either due to hit on performance or due to decreased usability. Further, narrowing the trust circle to exclude even hitherto trusted system administrators makes creating an encrypted filesystem a huge engineering exercise.

In this thesis, we talk about key management issues in TransCrypt[21], an encrypted file system design with smallest trust circle to the best of our knowledge. We provide an entire architecture with utilities like secure key stores, and their management through libraries inside and outside the kernel space. We provide enhancement of kernel CryptoAPI to include asymmetric cryptography, filesystem and file metadata management tools, and a communication framework to authenticate genuine users through user-space key stores. We present a design that incorporates modularity, flexibility while providing a transparently operational encrypted filesystem.

Dedicated to

My lovely niece, Pari

Acknowledgements

I would like to express my gratitude to my thesis supervisors Dr Dheeraj Sanghi and Dr Rajat Moona, without whose able guidance this work would not have been possible. I thank Dr Manindra Agarwal for his insights in the project. I would like to thank all the people who have been a part of TransCrypt group - Mohan Dhawan, Satyam Sharma, Deeptanshu Shukla, V Bhanu Chandra, Arun Raghuraman and Sainath Vellal. Intense discussions during TransCrypt meetings with these people have been a great learning experience, besides being instrumental in the conceptualization and implementation of this work.

I would also like to thank Prabhu Goel Research Centre for Computer and Internet Security for partially supporting my thesis and providing me with excellent facilities and a highly flexible and open environment.

Any of my endeavours at IIT Kanpur would not have been possible without the support of my friends, especially my wingmates, who have made my stay at IIT Kanpur a memorable one. I thank them for being with me at all times.

Finally, I would like to thank my parents and my family for their affection and encouragement to succeed in all my ventures.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Data security has become a very important issue with growing dependence on storage systems and increasing reliance on the internet for communication. Millions of dollars have been reported to be lost due to security breaches [16]. The key factors of interest for a deployment of a storage solution are security, performance and usability.

Further, administering networks and storage over thousands of systems spread over employees in companies with large headcounts is an increasingly difficult task. With growing number of outsourcing industries, such usage conditions are becoming very common. These industries are usually attributed with high attrition rates. Getting system administrators who may be trusted with proprietary data is a difficult task. Theft of data by employees who may later cross-over to competitors is another concern. While companies commonly impose restrictions like not allowing USB thumb drives in offices, these restrictions have been ineffective in preventing data thefts and are generally inconvenient.

Security is a greater concern in defense establishments, where secrecy of data is of paramount importance. Most data thefts are found to be rooted at collusion with system administrators. Similar examples are present in other Government organizations. Most users in this kind of environment are non-experts who require ease of usability of any new solution provided.

The pressing need is to develop storage solutions which may provide a heightened security barrier. While several cryptographic filesystem designs are available, as detailed in Chapter 2;

all trust the system administrator. We present implementation of a cryptographic filesystem, designed earlier by our group [21] that minimizes trust on the administrator. The other key issue is that of transparency for regular users who should be able to migrate to the new solution with minimal learning curve. Other issues that of interest for an enterprise deployment are data recovery and compatibility with existing systems.

1.2 Scope of this Work

The TransCrypt filesystem was designed and a proof of concept was presented in our earlier work [21]. We provide the key management scheme for TransCrypt in this work. This involves design of storage for cryptographic metadata in the filesystem, metadata creation, metadata extraction while opening encrypted files and metadata management. We also provide the kernel cryptographic algorithms necessary for implementing this key management scheme. We also detail communication with user space utilities which has been done in association with another work of our group [23]. Some of the salient features of this work are:

- Providing support for handling cryptographic metadata.
- Providing a TransCrypt version with full filesystem and file lifecycle.
- Adding associated cryptographic support.
- Providing user space support modules.
- Communication framework with user space utilities.

The last two above have been done in association with another work of our group[23]

1.3 Organization of Thesis

In the next chapter we discuss other encrypting filesystems and establish the need for a new encrypting filesystem. In chapter 3 we outline the architecture of TransCrypt. We also discuss the security framework for TransCrypt. We provide implementation details in chapters 4 to 6 which form the core of this work. In chapter 7 we take an insight of the future work on TransCrypt and finally provide our conclusions.

1.4 Abbreviations and Definitions

PFK. Per File Key. It denotes the key used to encrypt a file.

FSK. File system Key. It denotes the file system wide key.

PKS. Private Key Store. It denotes the secure module that stores the private key of a user and provides token decryption service.

daemon. It is the user space daemon that communicates with the kernel and facilitates key acquisition for the kernel.

SK. Session Key that is established by the kernel with the PKS.

$\mathbf{PuK}_{UID,CERT-ID}$**.** It denotes the Public Key of the user with user id UID as mentioned in certificate with certificate number CERT-ID.

$\mathbf{PrK}_{UID,CERT-ID}$**.** It denotes the Private Key of the user with user id UID corresponding to the public key as mentioned in certificate with certificate number CERT-ID.

$\mathbf{(A)}_b$**.** Cipher text produced by encryption of plain text **A** using symmetric key b.

$\mathbf{[A]}_{\mathbf{PuK}_{UID,CERT-ID}}$**.** Cipher text produced by encryption of plain text **A** using public key $\mathbf{PuK}_{UID,CERT-ID}$.

Chapter 2

Related Work

2.1 Approaches to Designing Encrypting Filesystems

An encrypting filesystem design takes several approaches. The design of an encrypting filesystem needs to trade between flexibility, efficiency and security. The various encrypted filesystems use a variety of techniques and it is essential, before we venture out to compare them, to lay out the parameters which decide what makes a good encrypting filesystem.

1. **Encryption Layer.** This decides where the actual encryption/decryption operations are performed on the contents of a file during the write/read process from the disk to memory. This would decide where data remains in plain text in the various stages in which it exists on the system, like buffer cache, page cache. The criterion is important because while some of these buffers are per-process, others are per-system and therefore affects other elements of design.

2. **Granularity and Encryption Target.** This refers to the smallest unit which uses the same encryption key. This may be whole filesystems as in the case of Cryptographic File System(CFS) [2], or a per-file key as in eCryptfs [10]. It is also necessary to decide what elements are stored encrypted on the disk. The objects in question are file data, metadata, file system based information etc.

3. **Algorithm.** This refers to the choice of using symmetric key or asymmetric key based algorithms. Symmetric key algorithms provide better speed while asymmetric key based algorithms provide greater security by means of robust key management scheme. A

trade-off between the two also brings up using a combination of these algorithms.

4. **Key Management.** The design needs to specify the format and location in which the keys are stored. This may be the metadata of the file or the filesystem information in the superblock as two examples. Also, an important design decision is whether the keys are per-file, per-user or per-system.

5. **User Interaction.** All encrypting filesystems need to define how the keys will be loaded for authentic users and disallow invalid access to the files. This design detail will have security implications. For example, the author may be hesitant to have a pass-phrase based authentication scheme, susceptible to dictionary attacks. This layer also defines access control for the data. A file may be shared among multiple related users (such as groups) or among multiple unrelated users on an individual basis.

6. **Admin Role.** The role of the administrator and the housekeeping needed to be done by the administrator is another essential point in the design of a filesystem.

7. **Recovery.** Good encrypting filesystem designs also need to address handling of events when things go wrong, for example events such as loss of keys, unavailability of genuine users. There will generally be other non-generic issues evolving out of each particular design.

Integrity is another important criteria for a security-conscious user, but is generally not essential for an encrypting filesystem design and hence is not present in the above list. Some other factors which need consideration during the design process are installation, efficiency, ease of usage, and last but definitely important, backward compatibility with systems, architectures and file-formats.

2.2 Survey of Encrypted Filesystems

Based on the parameters identified in section 2.1 we now survey the existing encrypting filesystems. While this work deals with GNU/Linux based implementation we also discuss the Windows EFS in our survey.

2.2.1 Microsoft EFS for Windows

Microsoft Encrypting File System [5] provides a secure filesystem, as long as user does not gain physical access to the machine. The salient features are the following:

- EFS encrypts files using NTFS file control call-out feature. The files therefore remain encrypted in system-specific buffers.
- Files and folders that are to be encrypted by the file system must be marked with an encryption attribute.
- EFS uses a hybrid approach with random per-file keys, which are encrypted with a user's public key.
- EFS uses only user-specific keys and no system-wide keys. The keys are stored in the header of each file that is encrypted.
- The public keys are stored on the system itself. In Microsoft's new Vista edition it provides authentication using a smart card module.
- The administrator functions as the default recovery agent. EFS provides a strong design for data recovery.

While EFS restricts itself to Windows platform, it does provide several good design attributes. It however does not provide protection against physical access, and the provision of explicitly setting the encryption attribute decreases its transparency and possibility of security lapse due to carelessness.

2.2.2 Cryptoloop and dm-crypt

The original patch of Linux CryptoAPI [6] in the 2.2 series kernel was designed to provide filesystem encryption. Cryptoloop [11] provides an indirection at mount time whereby system calls are intercepted to provide encryption and decryption of filesystem data.

- Cryptoloop uses the loop device as a pseudo device that allows each file system calls to be intercepted for encryption/decryption. This implies that all system and process buffers remain encrypted.

- Cryptoloop encrypts the entire filesystem using a common mount-time pass phrase.
- It uses a symmetric key algorithm which is provided during mount operation.
- The authentication is solely based on the pass-phrase and very susceptible to dictionary attacks.
- No recovery is possible if the user forgets the password as there is no specific recovery agent.

The Cryptoloop project has now been deprecated and is replaced by more actively maintained dm-crypt [8]. dm-crypt [8] provides several additional features and improves upon the cryptoloop with the same base architecture.

- dm-crypt provides setting up IVs, and chaining modes
- It provides greater compatibility with filesystems, and decouples itself from the Logical Volume Manager(LVM)

While a mount time encryption scheme as cryptoloop loop has advantages with compatibility, it falters on various flexibility and security aspects. The absence of recovery agent makes it usage for critical data dangerous. A single system wide key implies re-encryption of entire filesystem if one needs to change the key.

2.2.3 Cryptfs

Cryptfs [29] was developed as an encrypted filesystem which gave way to more robust and now well maintained filesystems eCryptfs[10] and NCryptfs[27]

- Encryption is done by calculating the pages being affected by a write. This leads to encryption of more blocks than that are affected. Encryption layer is at page cache.
- It encrypts at a file granularity. However, multiple files maybe using the same key. Cryptfs decouples encryption with ownership.
- It uses Blowfish algorithm with CBC mode. It does not use asymmetric keys.

- The keys used are per-session. This means the users sharing same keys have to use the same session.
- The keys are derive from a 16-character pass-phrase prompted when a user starts a new session for reading files.
- Administrator alone can mount the encrypted filesystem and is kept totally in the trust circle.
- No recovery module is specified.

Cryptfs claims to be just a proof of concept and not a full fledged cryptographic file system. Its two off-shoots eCryptfs and NCryptfs are more important from a comparison point of view.

2.2.4 eCryptfs

eCryptfs [10] uses some features of Cryptfs notably, per-session based authentication. It has some important changes

- The cryptographic metadata is now stored in the header of files itself. The storage format makes it incompatible for sparse files.
- There is a per-file key, which is encrypted with a session-wide key provided through a pass-phrase. These are called authentication tokens.
- The authentication tokens are per-session. This implies that reading files with different authentication requires a remount.
- eCryptfs uses the kernel keyring infrastructure.
- While pass-phrase mode is suggested, it also provides a pluggable authentication module, which is under development.

eCryptfs provides flexibility and a higher security cover. However, superuser is still in the trust domain, the new file format and session based authentication token are some negative points.

2.2.5 NCryptfs

NCryptfs [27] improves substantially on Cryptfs specifically in providing group access, and improving upon the session specific authentication scheme.

- It adds support for multiple ciphers and authentication methods.
- It provides an architecture for providing access to groups.
- The system administrator is trusted for proper mounting and for proper user space components. It is not trusted with the encryption keys
- It distinguishes between owners and other users, where owners set out policies for other users.
- Provides a user mode mount feature called *attach*
- While authentication is still based on sessions, the access control is managed by using policies, and prompts for keys.
- Keys are prompted on command line after policies allow usage.

NCryptfs provides several desirable features, but its trust on user space processes makes it susceptible to masquerading attacks. Also, request for keys at command line hinders with transparency from a usage point of view.

2.2.6 Cryptographic File System (CFS)

Cryptographic File System(CFS) [2] is an encrypting file system that is implemented as a user-level NFS server. Users create a directory on the local or remote file system to store encrypted data. The cipher and key are specified when the directory is first created. It provides a user space daemon, that is responsible for providing the owner access to the encrypted data through an *attach* command. The daemon, after verifying the user ID and key, creates a directory in the mount point directory that acts as an unencrypted window to the user's encrypted data. Once attached, the user accesses the attached directory like any other directory. CFS runs all its major operations in the user space and so its major drawback is performance due to the several context switches required. Also, due to need of explicitly attaching a file to be used as encrypted, it falters on transparency from a usage perspective.

2.2.7 Transparent Cryptographic File System (TCFS)

TCFS [13] is a improved form of CFS, which eliminates need of *attach* and *detach* commands required in CFS. It allows setting up encrypted attributed associated with files and directories instead. It uses the UNIX authentication system to authenticate users. The file keys are stored in a directory. TCFS is very low on security because of its reliance on login-passphrase based system and storage of file keys on the disk itself.

2.2.8 BestCrypt

BestCrypt [1] is a commercially available encrypting filesystem. Its features are the following:

- The encryption is provided by intercepting system calls using a loopback device.
- It defines *containers* which are huge files loaded as a loopback raw block device. *Containers* are backing stores that may be formatted as any filesystem
- Each container is associated with its own symmetric key.
- Containers need to be created, formatted, mounted by the administrator.

BestCrypt is quite unsuitable for multi-user modes as it involves huge administrative roles and prevents sharing of *containers* amongst users with unequal permissions.

2.2.9 Steganographic File Systems

Some other important filesystem with greater security cover use steganography along with encryption like StegFS [14] and Magikfs [24]. These not only prevent access to hidden data but also what is hidden on the filesystems. StegFS provides several security levels, each with its own key. It modifies EXT2 kernel driver to maintain a separate block allocation table for files on StegFS. Besides, these pack in some other security features to prevent data recovery after deletion. The goal of these file systems are more oriented towards hiding data, rather than providing transparency and trading with performance.

2.3 Summary

Most filesystems maintain page cache in encrypted mode. While this provides easier implementation, encrypting/decrypting at buffer cache is faster. However, maintaining page cache unencrypted is susceptible to memory screening attacks. Apart from Microsoft EFS, no encrypting filesystems uses a hybrid approach of symmetric and asymmetric keys. The superuser is fully trusted in most implementations except in NCryptfs. Minimal trust on superuser which is an important goal of TransCrypt. All filesystems including NCryptfs have ignored masquerading attacks by which the user space utilities can be faked. Such attacks are other considerations dealt with in TransCrypt. TransCrypt offers per-user keys for access control and authentication. and uses a novel scheme for thwarting super-user based attacks by including a system-wide key. While only some designs provide full transparency to users, TransCrypt states making file operation totally transparent as its basic goal. The recovery framework dealt earlier [21] provides the key feature for enterprise deployment.

Chapter 3

TransCrypt Architecture

TransCrypt filesystem described in this chapter is essentially an extension of work done earlier [21, 22]. A number of improvements have been made in this design, and this chapter attempts to provide a complete picture of the current TransCrypt design.

3.1 TransCrypt Features

We discussed in section 2.1 the parameters that are considered while designing and encrypted file system. We now present a TransCrypt feature set based on those criteria.

3.1.1 Per-file Encryption, Per-user Access Control.

TransCrypt encrypts each file with a unique per-file key(PFK). The PFKs are randomly generated at each file creation. The security granularity for access is extended to per-user by using a special cryptographic scheme outlined in section 3.1.3

3.1.2 Block Level Encryption.

TransCrypt encrypts data at granularity of filesystem blocks. The encryption/decryption may be performed during the block write/read level or page write/read level depending on a user's threat perception. The former maintains an unencrypted page cache while the latter keeps the page cache encrypted covering a greater threat model. This option is configurable by a compile time option.

3.1.3 Novel Cryptography Scheme.

The PFKs are generated randomly to protect data. However, access control is provided for a user. This is achieved by combining symmetric key based PFKs with asymmetric key cryptography. The PFK for a file is encrypted using the public key of a user to whom access is to be provided. This is known as ***token*** and is part of the metadata of the file. In order to get access to a file, the user must be able to process the token using his private key. This functionality is completed in the user space. This however has a problem where the naked PFKs travel in plaintext back to the kernel after the token is decrypted using the private key of the user. Also, it is not desirable for even the legitimate users to have access to the PFKs. TransCrypt uses a novel scheme by including a filesystem-wide key, the **File System Key(FSK)** to blind the PFKs. The actual format cryptographic metadata, the ***token***, is the following.

$$token = [(PFK)_{FSK}]_{\mathbf{PuK}_{UID,CERT-ID}}$$

The notation mean that PFK is first encrypted using symmetric key FSK. This is then encrypted with the public key of user with user-id as UID and corresponding certificate id as CERT-ID. A list of these tokens is stored in the *Access Control List* (ACL) of a file. $\mathrm{PuK}_{UID,CERT-ID}$ denotes the public key of the user. The meaning and need of UID and CERT-ID are explained when we talk about cryptographic metadata storage in more detail in section 3.2.1.

3.1.4 Multiple Transparent Authentication Options

User authentication in TransCrypt is achieved by a decryption with private key and this operation is performed in the user space. TransCrypt defines an independent authentication module which services kernel requests. This module is known as the Private Key Store(PKS). This authentication module may be a smart card based module, a secure authenticating server or anything else. Once the smart card is inserted or the server is configured, the user can work transparently without any special intervention by him, such as entering a pass-phrase each time a file is opened. This provides a higher degree of convenience compared to many other systems. We use a user-space daemon to communicate with the kernel, but we do not impose any security assumptions on it. Such a daemon acts simply as a conduit between the kernel and the PKS.

3.1.5 Minimum Admin Trust

TransCrypt keeps the super user outside the trust model which means even if an attacker gains superuser privileges through some compromise, the attacker will be unable to gain access to a user's file. It may be noted that the design does allow partial trust in the superuser and is not resistant against attacks like kernel image forging. Use of Trusted Platform Module in future will further restrict the Trust Model. This design is outlined in section 3.3.

The File System Key(FSK) discussed above is stored with a hash in the superblock of the filesystem. The FSK is obtained through a mount-time pass-phrase, which needs to be supplied by the superuser while mounting a TransCrypt filesystem.

3.1.6 Data Recovery Agent

For purpose of data recovery in case of lost keys, we use a data recovery agent. The superuser in TransCrypt is unable to read files. A special user called the *Data Recovery Agent* **(DRA)** is given read access to each file at the time of creation. The **DRA** is an additional hierarchy created by TransCrypt with the sole responsibility of recovering lost files. The private keys of the **DRA** is broken and stored with multiple entities. In a real life scenario, recovering files should require multiple people to come together and provide authentication to recover files.

3.1.7 Integrity

TransCrypt design [21], provides an integrity framework for TransCrypt. Keyed hashes of each block will be created, and stored with the metadata. A “hash of hashes” will also be signed with the user's private key and stored in the metadata. The metadata would be also hashed and signed to prevent altering permissions and tampering with cryptographic metadata. The current implementation lays smaller focus on the integrity of data and relies on the current OS implementation for the same.

3.2 The TransCrypt Architecture

3.2.1 Cryptographic Metadata

TransCrypt uses the *Access Control List* [9] data structure in the filesystem for storing its cryptographic metadata(token). We disallow the usage of un-named users in the ACL. Only named users are allowed. The named user entry in the ACL is augmented to contain two more fields in addition to the uid and permissions. These fields are *cert-id* and the *token.* Since each user may have several key pairs, the unique public key used is identified by the *(uid,cert-id)* pair. *token* is the cryptographic metadata as described earlier in 3.1.3.

3.2.2 Filesystem Creation, Mounting

At the time of creation of the filesystem, the user is prompted for a pass-phrase, which is then used to create the FSK. The FSK is then hashed and stored in the superblock of the filesystem.

The filesystem to be mounted as a TransCrypt partition is mounted with the ‘acl’ and ‘tcpt’ option switched on. The pass-phrase needs to be provided at the prompt in order to construct the FSK and verify it with the hash stored in the superblock. The FSK is then stored in the in-memory image of the superblock.

3.2.3 File lifecycle

In addition to the normal operations certain additional cryptographic operations need to be performed during creation, opening, reading and writing of the file. In addition to this ACL based commands(setfacl, chacl) also require certain changes as described later.

3.2.3.1 Creation

- A random PFK is generated for the file.
- PFK is blinded by performing a symmetric encryption using FSK as the key creating the blinded PFK.

- The certificate of the user is obtained by requesting the user space daemon for it. The user space daemon returns the most current certificate and the corresponding cert-id is later stored in the ACL entry.
- The certificate is verified by the kernel and public key of the user is extracted.
- The token is created by performing a RSA encryption on the blinded PFK using the public key of the user.
- The token is stored in the ACL entry along with cert-id and uid.

3.2.3.2 Opening

- The token is extracted for the user using the UID.
- The token is sent over to the PKS for decryption. After decryption with the private key, the daemon returns the blinded PFK.
- The blinded PFK is decrypted by using FSK from super block to retrieve the PFK.
- The PFK is stored in the in-memory metadata of the file.

3.2.3.3 Read/Write

- Symmetric key encryption is performed whenever a block is written back to the disk. If page cache is to be encrypted, encryption takes place whenever a page write is committed.
- Symmetric key decryption is performed to provide plaintext data to the application. If page cache is to be kept in encrypted form, decryption takes place whenever a page is requested. Otherwise, decryption takes place when a block is read from the disk and plaintext is stored in the page cache.

3.2.3.4 Changing Permissions

- **Providing Access:** Access may be provided to a user only by the owner of the file or users having access to the file. In such a case, when the kernel updates the ACL entry, it requests the PKS to decrypt the user's token to get the blinded PFK which is then encrypted by the public key of the new user to create a token. The token is then used to create a new entry for the named user in ACL of the file.

- **Revoking Access:** Access may be revoked by simply deleting the ACL entry.

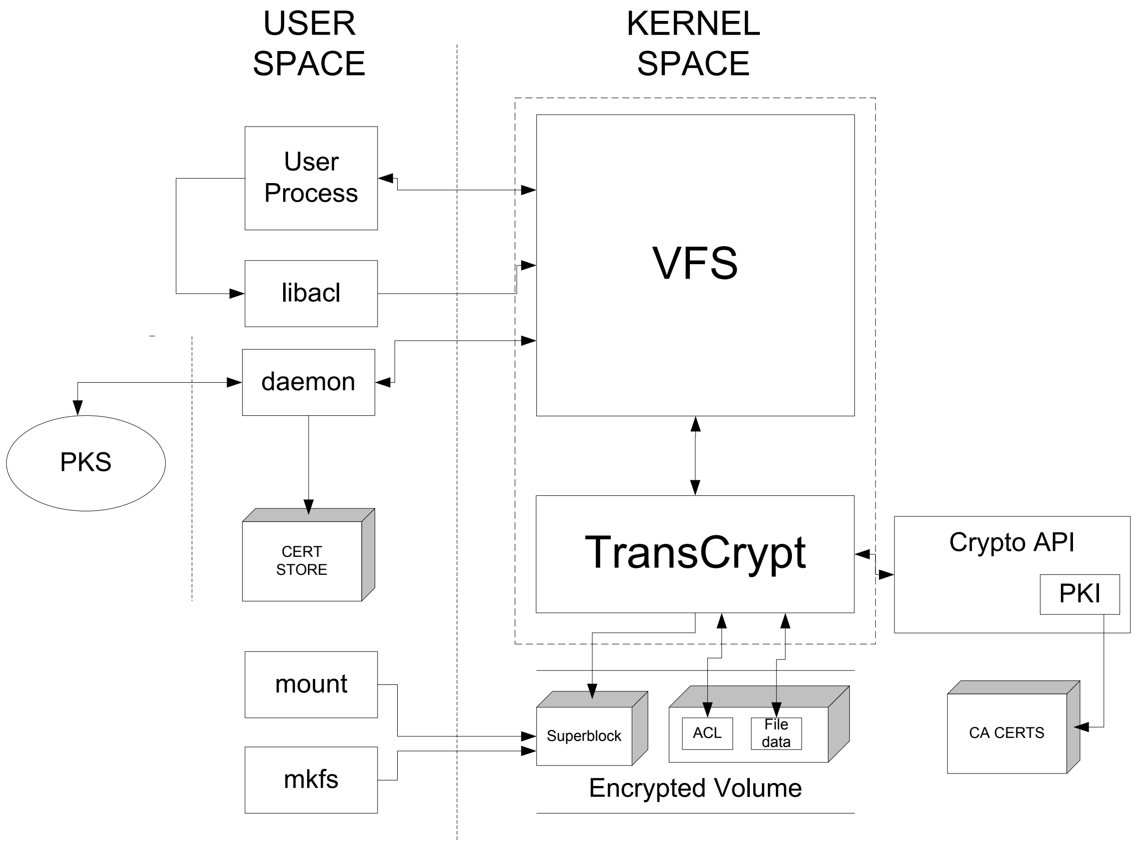

Figure 3.1: Schematic TransCrypt Architecture

3.2.4 User Space Modules

Two user space modules are used by TransCrypt: (a) The **TransCrypt daemon** for communication between the kernel and the user space; and (b) The **PKS** for providing the private key operations. The user space functionalities have been split between the daemon and the PKS as the daemon needs to be present on the system while the PKS may be at a remote location. Further, the PKS is trusted while no such assumption is made about the daemon. The TransCrypt daemon provides two services.

- Provides the kernel with user certificates
- Forwards kernel's token decryption requests to the **PKS**.

The **PKS** decrypts the token using the private key of the user corresponding to the cert-id and provides the blinded PFK to the kernel when requested. This operation is conducted over a secure channel created before each such token decryption request.

Now that we have defined each of the modules of TransCrypt we can outline the complete TransCrypt architecture. A schematic representation is shown in the Figure 3.1. This figure is an adaptation from the earlier work [21] and has been augmented to represent the changes in TransCrypt. The Crypto-API of the kernel is augmented with certificate parsing and validation routines as outlined in Chapter 4. The user space communication and implementation of the TransCrypt daemon and PKS is detailed in Chapter 5. Finally, the augmentations in the GNU/Linux kernel and utilities such as *mount, mkfs, libacl* is detailed in Chapter 6.

3.3 Security Framework

TransCrypt provides a unique security framework by minimizing the trust on the superuser. The aim is to avoid all forms of online and offline attacks. Some of the salient security features and attacks avoided are outline here.

3.3.1 Physical Access to the System

Physical Access to the system does not compromise security in TransCrypt. TransCrypt does not allow anyone to see any files when a hard disk or a laptop is stolen or someone has access to a system physically. This is because of the fact even if someone gains access to a session, without the PKS module to perform key acquisition, he cannot access the files of a user. A scan of the disk data will provide metadata in plaintext and file data in ciphertext. The metadata is unable to provide the PFKs without the PKS and using asymmetric cryptography ensures that even dictionary attacks or brute force are infeasible.

3.3.2 Superuser Under Minimal Trust

A user may gain superuser privileges through loopholes or backdoors in softwares using techniques such as buffer overflow attacks [15]. Without PKS authentication, the superuser will not be able to read a user's file. PKS authentication is not possible unless the superuser is provided access to the file through ACL entries. If the page cache is encrypted, even a memory screening will be unable to gain access to any plaintext.

We however trust the superuser for booting a non-malicious kernel and loading non-malicious modules. The user-space daemon runs under superuser privileges is however, not trusted. A malicious masquerading daemon may launch a denial of service attacks but it will not be able to gain access to the files as all communication over the PKS is secured by session keys established between the kernel and PKS. As we discussed in Chapter 2, no existing filesystem provides this security framework.

3.3.3 Enhanced Security through FSK

FSK provides enhanced security when some user may have been provided access in the past, but is no longer considered legitimate. Through FSK based blinding, even legitimate users do not have access to PFKs used to decrypt the files.

It should be noted that FSK just provides blinding and is not required to be very safe as FSK itself does not encrypt any data. Individual access to FSK will not provide attacker with any illegitimate access. Therefore, we choose to store the hash of FSK in the superblock itself and use passphrase to obtain FSK at mount-time.

3.3.4 Concurrent Access by Multiple Users

Though TransCrypt encrypts data at buffer cache layer where context of who requested a read or write is not available, it does not allow the following attack. A user Alice is reading a file. At that time a malicious user Marvin tries to open a file. Since we do not authenticate the PKS, the key acquisition may result in a garbled key being returned by the daemon. This key is first compared with the key stored in the keyring through the inode context and only then is the user allowed access. If this step was not performed, Marvin would be able to launch a denial of service attack as future read/write operations of Alice would use incorrect PFK.

Another attack scenario is when Marvin could open a file and have an incorrect PFK through a masquerading daemon installed in the inode context. It could then go to sleep and wait for Alice to open the file. However, since Alice's key will not match the installed PFK, the real PFK will not be stored in the inode context. If this comparison was not done, then Marvin could have gained access to the files after opening the file and then waiting for Alice to open the same file. However, while this does launch a denial of service attack, it prevents data theft which is our primary aim.

3.3.5 Secure PKS

We assume that PKS is extremely secure. This is one of the basic assumptions of TransCrypt. However, by providing secure communication setup with the PKS, we include attacks like middleman listening on communication channel to the PKS, especially important when the PKS is a remote authentication server. Even in case of smart cards, this would handle cable snooping attacks.

While we understand that TransCrypt is still susceptible to sophisticated attacks by the superuser where online access to the memory may leak certain data, we aim TransCrypt for an enterprise deployment where such online attacks by the superuser is considered infeasible or unexpected. However, it completely protects against malicious users and illegitimate physical access. There are other denial of service attacks identified by us like overwriting user certificates, daemon masquerading etc., but we decided to ignore such denial of service attacks in favour of simple design and improved performance.

Chapter 4

Kernel Crypto API

This chapter focuses on the augmentation of kernel crypto API. We summarize the crypto API additions made in the previous work [21], [4] and outline the augmentations made by us. While the kernel crypto API additions are essentially designed to serve the requirements of TransCrypt, the implementations are towards developing a generic API. However, omission of certain features from this generic API are either due to the lack of their necessity for TransCrypt or for the sake of a simplistic implementation. We also talk about the keyring infrastructure used by our implementation.

4.1 TransCrypt's Cryptographic Requirements in the Kernel

While several standard user space libraries exist for both symmetric and asymmetric cryptographic operations in the user space [25, 12], the kernel cryptographic functionality has so far been restricted to symmetric key operations [6]. TransCrypt requires support for following cryptographic functionalities.

1. **Symmetric Key Operations.** These operations are already implemented in the kernel.In TransCrypt these operations are for blinding, data block encryption/decryption, securing communication with the PKS, extracting FSK from super block.

2. **Public Key Operations.** TransCrypt requires encryption with public keys for token creation and establishing secure session establishment with the PKS. We also need compatibility with PKCS#1 [19] standards while creating secure cipher text.

3. **Certificate Parsing and Verification.** As public keys are stored in the user space, there authenticity is verified in the kernel by using certificate chains signed by trusted root CAs. We therefore need certificate parsing and verification routines in the kernel.

The earlier works [21, 4], had added RSA encryption and decryption services for a fixed modulus size 1024. The implementation [21] took as input a 1024-bit integer A and created a cipher text C by doing the modular exponentiation.

$$C = A^e \ mod \ n$$

Here e is the public key while n is the modulus. In our case, n is taken as 1024. However, the input A is usually smaller than 1024 bits. For our requirement, the plaintext is usually the symmetric keys of length 128 bits. When dealing with input strings smaller than the modulus size in RSA, PKCS standards [19] lay out specific padding schemes which have now been incorporated in the asymmetric crypto API of the kernel. The private key based decrypt, sign and verify functions have similarly been implemented as outlined PKCS#1 standards. These functions are henceforth referred to as `RSAEP, RSADP, RSASP1` and `RSAVP1` as stated in the PKCS#1 standards.

Certificate parsing and verification functionality was added earlier [3] by porting user space implementation of *xyssl* [28].

4.2 PKCS#1 compliance

Asymmetric cryptographic operations require implementation of four routines - encrypt with public key of recipient, decrypt with private key of receiver, sign with private key of signer and verify with public key of signer. While TransCrypt's current design requires the public encrypt and verify operations, our implementation includes all four functions to provide a generic functionality. We implement two padding schemes as laid out in version 2.1 of PKCS#1 [19]. We detail the padding schemes implemented by us. The exponentiations required for each operation is achieved using the implementation of [21, 4] and is therefore not detailed here.

4.2.1 Encryption using Public Key

We implement two padding schemes as outlined in PKCS#1 version 1.5 and the *OAEP* padding detailed in version 2.1 of the PKCS#1 document [19]. Once the padding is done we perform a `RSAEP` to create the cipher text.

4.2.1.1 Version 1.5 Padding

The plain text block(encoded message), EM corresponding to a message M looks like the following:

$$EM = 0x00 \parallel 0x02 \parallel PS \parallel 0x00 \parallel M$$

Here, PS is a padding string of non-zero octets of length $k-$ *message length* -3. k is the length of modulus in octets. In our case, this is equal to 128(or 1024 bits). The function which performs this padding is called `EME_PKCS_V1_5_ENCODE`.

4.2.1.2 OAEP Padding

PKCS#1 version 2.1 lays out a more comprehensive padding scheme called the Optimal Asymmetric Encryption Padding(OAEP). The scheme is shown in figure 4.1. The figure has been borrowed from the version 2.1 of PKCS#1 draft [19].

The function is called `EME_OAEP_ENCODE`. The implementation of the encoding scheme is as follows.

- *lHash* is generated from label L using SHA-1 hash.
- *DB* is generated by the following concatenation:

  $$DB = lHash \parallel PS \parallel 0x01 \parallel M$$

  Here, M is the message to be encrypted. PS is a zeroed padding string of length $k -$ *(message length)* $- (2 * \textit{hash length}) - 2$. Again, k is the length of modulus in octets. In our scheme, this translates to $86 -$ (*message length*)
- A random *seed* is generated of length equal to the length of hash.

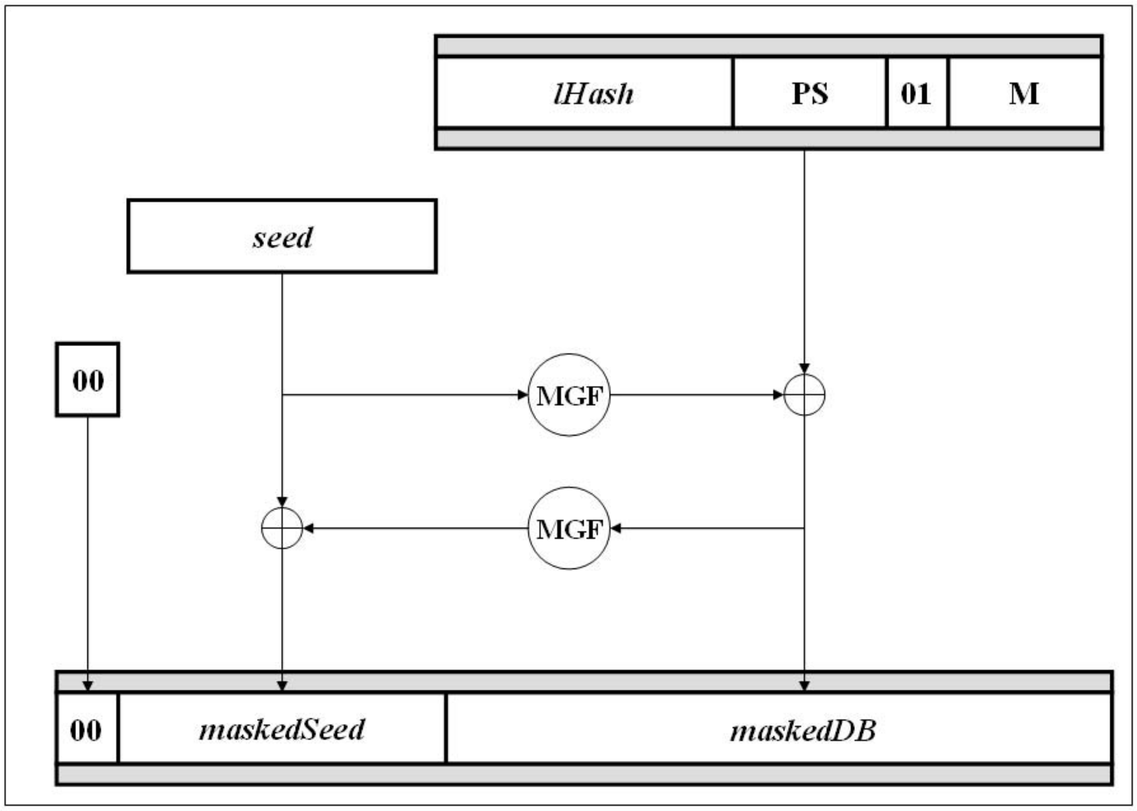

Figure 4.1: Schematic description of OAEP Encoding Scheme. *lHash* is hash of label L. *PS* is padding strong of zero octets. *Figure adapted from [19]*

- A mask of length equal to DB is generated using the mask generating function. The mask generating function is outlined later. We call it *dbMask*. The mask generating function generates mask using seed as the input. We create masked DB by XORing DB with the mask generated above.

$$maskedDB = DB \oplus dbMask$$

- We generate a mask of the length of seed with *maskedDB* as input called the *seedMask*. We then create *maskedSeed* by XORing with *seedMask*.

$$maskedSeed = seed \oplus seedMask$$

- Finally, the encoded message is created by concatenating the above two and appending a zero octet in the front.

$$EM = 0x00 \parallel \mathit{maskedSeed} \parallel \mathit{maskedDB}$$

4.2.2 Decryption using Private Key

The message is first decrypted using `RSADP` operation. The obtained block is then decoded to retrieve the message.

4.2.2.1 Decoding Version 1.5 Padding

The function is called `EME_PKLCS_V1_5_DECODE`. The steps are the following.

- Assert that the first two octets are 0x00 and 0x02 respectively.
- Ignore octets till a zero octet is found.
- The octets beyond the zero octet constitute the original message.
- Return the decrypted message and its length.

4.2.2.2 Decoding OAEP padding

The function is called `EME_OAEP_DECODE`. The decoding function is also provided the label L used during encoding of the message. The deciding takes place in the following sequence of operations.

- Generate Hash of the label using SHA-1.
- Assert that the first octet of decrypted message is 0x00.
- The remaining octets are parsed as *maskedSeed* and *maskedDB* where *maskedSeed* is of length equal to 20 bytes corresponding to SHA-1 hash and the rest form the *maskedDB*.
- The seedMask is generated by using the mask generating function to produce mask of length equal to 20 bytes corresponding to the length of SHA-1 hash and input as maskedDB.
- The seed is retrieved by XORing *maskedSeed* with *seedMask*.

$$\mathit{seed} = \mathit{maskedSeed} \oplus \mathit{seedMask}$$

- dbMask is retrieved using the mask generating function with *seed* as input and output length equal to length of *maskedDB*
- DB is retrieved by following XOR operation.

$$DB = maskedDB \oplus dbMask$$

- The first 20 bytes of DB is parsed as hash of the label and verified against the hash of the label provided.
- The zero octets are ignored till we reach an octet with a value 0x01.
- The remaining octets are the original message
- The message and its length are returned.

4.2.3 Signing Using Private Key

The Sign and Verify operation uses modular exponentiation with private and public keys respectively. The earlier implementation [4] simply performed modular exponentiation for signing and verifying. We implement the full signing-verification module in this work. The message to be signed is first encoded using either the scheme outlined in version 1.5 of PKCS#1 standard or the Probabilistic Signature Scheme(PSS) scheme laid out in version 2.1 of the document.

4.2.3.1 Version 1.5 Encoding

The function takes as input the message and the output length, which is equal to modulus of RSA. The hash function used is SHA-1. The function is called `EMSA_PKCS#1_V1_5_ENCODE`. The encoded message, EM has the format

$$EM = 0x00 \parallel 0x01 \parallel PS \parallel 0x00 \parallel T$$

The algorithm for hash is encoded with the hash value in the block T as an ASN.1 value with Distinguished Encoding Rules(DER). The block T is a value of type `DigestInfo` which has the digest algorithm followed by the digest octet string. PS is sequence of octet strings with values *0xff* and large enough to make the length of the encoded message equal to size of RSA.

The steps to encode the message just involve generating SHA-1 hash of the message, creating the DER encoded block and concatenating them. The modular exponentiation is then performed with private key to produce the signature using the `RSASP1` operation.

4.2.3.2 PSS Encoding

The PSS encoding is illustrated by the figure 4.2 from PKCS#1 standard [19]. The function encoding using this technique is called `EMSA_PSS_ENCODE`. The function takes as an argument the length of salt to be used. The steps used to create the encoded message are the following.

- Calculate length of encoded message as *emLen* octets. Here *modbits* is equal to length in bits of modulus.

$$emLen = \lceil (modbits - 1)/8 \rceil$$

- Generate SHA-1 hash of the message to be signed. Call it *mHash*
- A random salt is generated of length as provided in the argument.
- Create M' of (length 8 + hash length + salt length) and format:

$$M' = (0x)\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \parallel mHash \parallel salt$$

- Generate Hash of above message M'. Call it H.
- Generate DB of length (*emLen* - *hash length* - *1*) octets and format

$$DB = PS \parallel 0x01 \parallel salt$$

  The padding string is all zero octets and has length *emLen* - *hash length* - *salt length* - 2

- A *dbMask* is generated using the mask generating function of length equal to length of DB and input as H.

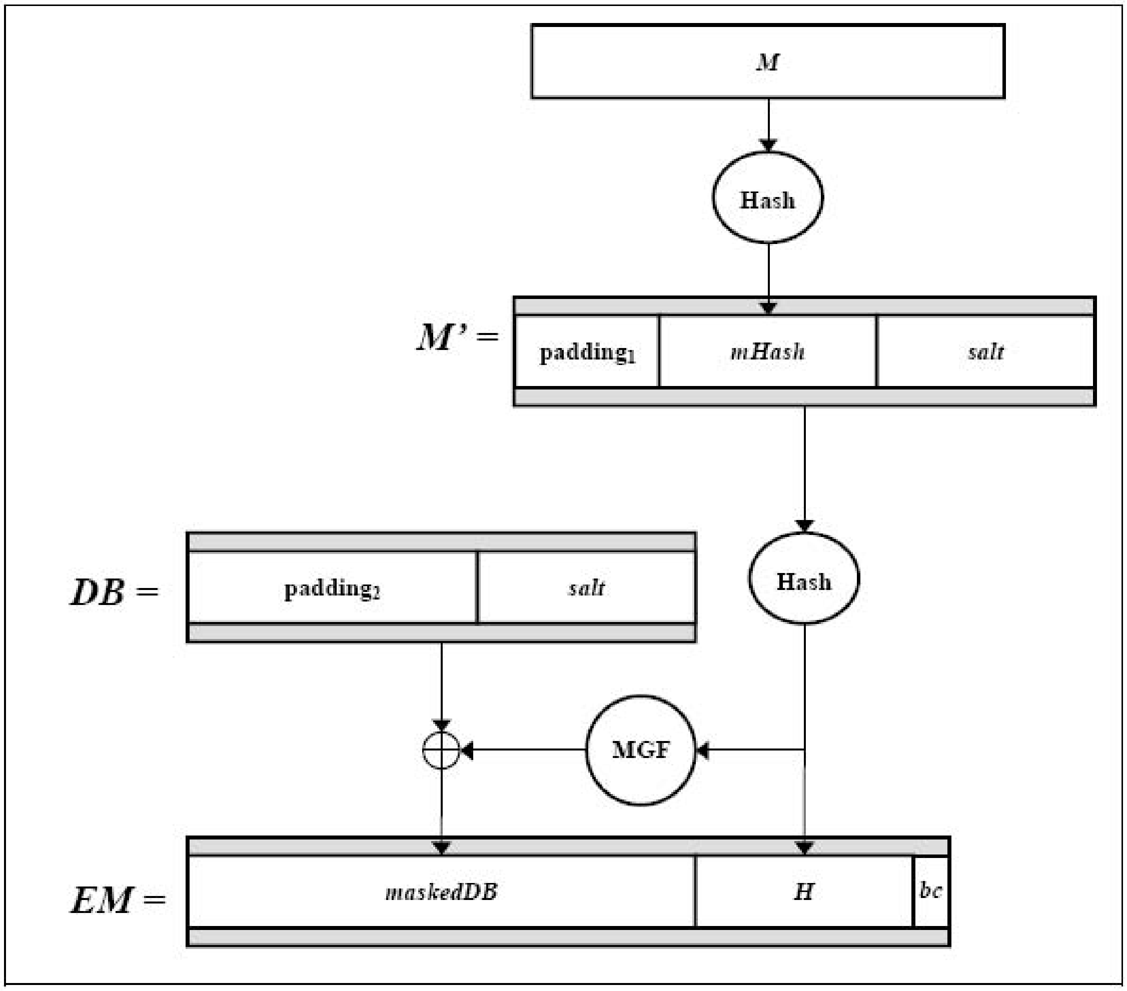

Figure 4.2: Schematic description of PSS encoding scheme. *Figure from [19]*

- The *maskedDB* is now created using

$$maskedDB = DB \oplus dbMask$$

- Most significant (8**emLen* - *emBits*) bits of the leftmost octet in maskedDb are set to zero.

- Finally, encoded message is created by concatenating *maskedDB*, *H* and followed by a special octet of value 0xbc.

4.2.4 Verification Using Public Key

The verification arguments take as input the message to be verified and the signature. In case of PSS encoding it also takes the length of salt used as an argument. The function returns success or failure. As there are two encoding schemes, they have their corresponding verification routines involving respective decoding schemes.

For verification, first the modular exponentiation of signature is computed using public key as outlined in `RSAVP1`. This produces the encoded message *EM*. The message whose signature is to be verified is then converted into another encoded message, EM' using the encoding schemes defined above. The two encoded messages *EM* and EM' are then checked for consistency. If they are consistent then the verification returns a success else a failure.

4.2.4.1 Verification for Version 1.5 Encoding

For version 1.5 the two encoded messages are verified for consistency by simply comparing their contents. A total match implies a valid signature else verification returns a failure. The function is called `RSASSA_PKCS_V1_5_VERIFY`

4.2.4.2 Verification for PSS Encoding

For PSS encoding, a simple match does not suffice due to inclusion of random values in its encoding. The verification routine is therefore done using the function `EMSA_PSS_VERIFY`. The function takes the message whose signature is to be verified and the encoded message produced by the `RSAVP1` operation. It also takes as input the length of salt used.

The `EMSA_PSS_VERIFY` function consists of the following steps:

- Calculate emLen, the length of the encoded message. Here, *modbits* is equal to length in bits of modulus.

$$emLen = \lceil (modbits - 1)/8 \rceil$$

- Generate SHA-1 hash of the message M. Call it *mHash*.

- Check that the rightmost octet has value *0xbc*
- Parse the encoded message to have *maskedDB* as the leftmost *emLen -hash length - 1* and *H* as the next *hash length*octets.
- Check that the leftmost *8 * emLen - emBits* bits are zeroes.
- Retrieve *maskedDB* using the mask generating function as a mask of size equal to length of *DB* and input as hash *H*.
- Calculate *DB* using

$$\mathit{DB} = \mathit{maskedDB} \oplus \mathit{dbMask}$$

- Leftmost *8 * emLen - emBits* of leftmost octet in maskedDb is set to zero.
- Check that the leftmost *emLen - hash length - salt length - 2* octets are zero and the next octet is $0x01$
- Retrieve *salt* as the last *salt length* octets of *DB*.
- Create M' as

$$M' = (0x)\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \parallel \mathit{mHash} \parallel \mathit{salt}$$

- Generate hash H' as the hash of M'.
- Check that $H = H'$. If all the checks above pass, then the signature is returned valid, else it is returned invalid.

4.2.5 Options used in PKCS#1 Version 2.1 Implementation

PKCS#1 allows options of hash functions as per the implementation requirement. Further the mask generating functions also have variable features. We use SHA-1 hash built in the kernel crypto API for all hash generation. The mask generating function used is the recommended mask generating function as outlined in PKCS#1 standards. We outline the mask generating function called `MGF1` here.

4.2.5.1 Mask Generating Function

The function takes an input string, the *mgfSeed* and length of generated mask as argument. The mask generating function also creates hash values. We again use SHA-1 hash for our purposes. The algorithm used is the following.

- Initialize a empty octet string T of length T_len such that

$$T\_len = \lceil (masklength)/(hashlength) \rceil \times (hashlength)$$

- For $counter = 0$ to $T\_len/(hash\ length)$
  - Convert integer *counter* to base 256, and create a 4 octet string C
  - Generate hash of $mgfSeed \parallel C$ and concatenate it to the octet string T
- Return leading *mask length* octets of T.

4.3 Crypto API

The asymmetric functionalities added have then been augmented to be made usable with the current symmetric crypto API [17]. We summarize some key point here.

1. We introduce `asymmetric_tfm` similar to `symmetric_tfm` in the kernel.
2. `asymmetric_tfm` contains the following functions in addition to other structures to define options used and to store public/private keys.
   - `ast_setprivkey/ast_setpubkey`: To set public and private keys.
   - `ast_encrypt/ast_decrypt`: Encryption and Decryption routines
   - `ast_sign/ast_verify`: Signing and verification routines.
3. The following publicly exported mechanisms are added:
   - `crypto_alloc_asymmetric()/crypto_free_asymmetric()`
   - `crypto_asymmetric_setprivkey()/crypto_asymmetric_setpubkey()`
   - `crypto_asymmetric_encrypt()/crypto_asymmetric_decrypt()`

- `crypto_asymmetric_sign()/crypto_asymmetric_verify()`

While the latter three correspond to the three functions defined above the first is used to allocate/free cryptographic transforms for asymmetric operations.

4.4 Keyring Infrastructure

A keyring infrastructure is present in the kernel to help user space utilities to maintain keys inside the kernel. We adapt this infrastructure to maintain keys of different files and different users in the kernel.

The kernel provides several keyrings with variable permission sets. These are thread specific keyrings, process specific keyrings, user specific keyrings and session specific keyrings. Once a reference to a keyring is obtained one may add or update keys in it using the function `key_create_or_update`. The function takes keyring reference as an argument in addition to a key payload and key descriptor. The key descriptor is a string value used to index the key.

The same descriptor is later used to retrieve the key by calling `lookup_user_key`. In using the keyring infrastructure one key decision was what keyring to use and another what should be used to index the keys in the keyring.

We add keys during `open/creat` system call while keyrings are retrieved at time of committing buffers to be written or while reading it from the disk. However, the read/write operations are asynchronous to the `read/write` system calls. This means that we do not have any context to the process which initiated the read or write. We, therefore need to use a keyring which maintains state over multiple users. Also, any process specific state may not be used for indexing the keys.

We use the session keyring which serves our purpose and use the string equivalent of inode pointer values to index our keys in the keyrings. Specific details of where the keys are updated, searched are discussed in chapter 6.

Chapter 5

Daemon and Private Key Store

Private Key Store(PKS) is the entity which stores the private keys of users. The PKS maybe per-user as in the case of smart cards or per-group as in the case of secure authentication servers. The kernel communicates with the PKS for decryption of tokens into Blinded PFKs. The communication with the PKS is done over a secure channel. This chapter outlines the various possible options for a PKS, the communication protocol and details of implementation.

5.1 PKS Options

The module which may act as the PKS is governed by the services that it needs to provide and limits imposed on it. The limitations are governed by security requirements and a model of implementation to ensure usage of a wide variety of entities as PKS.

5.1.1 Services Offered by the PKS

The PKS shall provide the following services to the kernel.

1. Session Establishment. The PKS should be able to establish a session as per the communication protocol laid out later. The session key(SK) once established needs to be remembered by the PKS for the ensuing key acquisition phase.

2. Key Acquisition. The PKS is required to decrypt the given token using the private key of the user. The decrypted token(the blinded PFK) is then sent over to the kernel, encrypted with SK.

The above two services require that the PKS be capable of performing asymmetric and symmetric cryptographic operations. It also needs to maintain state parameters like the session key, once a key acquisition is initiated. This will require some memory at the PKS where the session key is stored by the PKS while the key acquisition is taking place. It also needs to have non-volatile memory where the private key of the user is stored securely.

5.1.2 PKS Limitations

The choice and implementation of PKS is limited by certain properties of the communication.

1. As the PKS stores the private keys of user, it should be a reliable storage. Compromise of this storage shall imply the compromise of the TransCrypt filesystem.

2. While TransCrypt aims at different forms of PKS, it is imposed that PKS is able to deal with one session at a time, in order to make PKS compatible with per-user security stores like smart cards. For non-smart cards, it must be one session per user to avoid reflection attacks.

While the first limitation is a hard requirement for security, the second is a design decision that aims to provide a generic interface for PKS and to mitigate reflection attacks. Any batching of requests or attempts at parallelization needs to be implemented by the conduit daemon instead of PKS itself. This is discussed in further details later in this chapter.

5.1.3 Candidates for PKS

The PKS may be implemented a variety of schemes including the following.

1. **Smart Cards.** A smart card based PKS will have a smart card reader attached to the user system. The private key should never leave the smart card, which requires the smart card to provide cryptographic operations including session key establishment.

2. **Authentication Servers.** For large organizations, a secure server may be used to store the private keys of all users. It is recommended that such a server be dedicated one, with no open ports except the one listening for key acquisition requests.

3. **Trusted Platform Module(TPM).** TPM provides secure decryption in a hardware module. Together with a secure driver, TPM may also be used as PKS.

4. **E-token.** These are devices that provide functionality similar to smart cards.

While there are several other candidates for PKS, for example based on biometric identification, the schemes described above and those similar to these are the focus of the current implementation.

5.2 TransCrypt Daemon

Before we discuss the communication protocol for the PKS, we outline another entity of TransCrypt architecture, the TransCrypt Daemon. The TransCrypt daemon is a non-trusted entity, which facilitates communication between kernel and the PKS. It acts as a conduit for transfer of packets between the kernel and the PKS.

The daemon also has the additional responsibility of providing the kernel with latest user certificates from the user space certificate store. The certificates are indexed by user ids and a certificate id is returned along with the certificate.

The daemon communicates with the kernel over a netlink socket [20]. All packets other than the certificate acquisition packets are forwarded to the PKS by the daemon. For certificate acquisition requests, the daemon reads the certificate files, and returns it to the kernel. The daemon is also responsible for finding a route to the PKS which may be over some other conduit daemons. The route is decided by the uid and cert-id.

While we do not impose any security critical assumptions on the daemon, a masquerading daemon will be able to launch a denial of service attacks as discussed in section 3.3.

5.3 Communication Protocol

The kernel may request one of the following operations to be serviced by either the daemon or the PKS through the daemon.

1. **Certificate Acquisition.** The kernel requests the certificate of a certain user. The request is serviced by the daemon. The daemon looks for the latest certificate in

the certificate store, and returns it to the kernel along with the certificate-id. This communication takes place over unsecure connection. The certificate is acquired and verified by the kernel before extracting and using the public key.

2. **Session Establishment.** The kernel provides a session key which will be used by the kernel for secure key acquisition requests. The session key is encrypted using the public key of the user, hence retrievable only by a valid PKS. The session key is valid till the kernel kills a session or when the session is lost by the PKS because an alternate session is established, possibly by another TransCrypt file system.

3. **Key Acquisition.** The kernel provides the token (i.e. blinded PFK encrypted with the user's public key). The communication is secured using the session key established during a previous session establishment. The PKS looks up the user's private key, decrypts the token, and returns the blinded PFK encrypted with the session key. It may return an error packet indicating that the session has been lost by the PKS.

4. **Session Nullification.** The kernel requests nullification of an established session. The request is forwarded to the PKS. The session key is no longer usable after this, and any subsequent Key Acquisition request requires a preceding Session Establishment request. A session automatically expires when the user removes his PKS or kernel discontinues the session key even without informing the PKS or when another kernel may contact the same PKS.

5.3.1 Communication Interfaces

On booting, the kernel creates TCPT_MAX_FREE number of netlink sockets registered with the same number of NETLINK_TRANSCRYPT protocols. The daemon, which is started through the *inittab*, on startup creates TCPT_MAX_FREE number of threads and starts listening on each of these sockets. This provides TCPT_MAX_FREE number of possible parallel communications. These multiple netlink sockets enables multiple `open` requests to be processed, avoiding the need to wait for an ongoing key acquisition or certificate acquisition to finish. The number of TCPT_MAX_FREE in the current implementation has been fixed to 4. The need for parallel communication has been further addressed in a more efficient manner in [26].

Each daemon thread further binds itself to a IP socket with the PKS. The PKS may have another conduit daemon or driver which communicates with the PKS. This conduit daemon opens an IP socket and listens on it for a request from the daemon and transfers all requests to the PKS. There are several enhancements to the basic protocol described here, which are described in detail at the end of this chapter.

5.3.2 Packet Headers

Each packet sent and received by the kernel is pre-pended with the TransCrypt Header. The format of the header is shown in figure 5.1. The payload may be encrypted with the session key if the connection is secure. The header is never encrypted. The *type* field specifies what kind of request is being sought by the kernel. The *length* specifies the length of the packet including the header length. The *identifier* field is used to maintain sessions, and indicates who is supposed to receive the reply packet inside the kernel. A response to a packet is expected to have the same identifier. To elucidate further, suppose at a point in time, two threads inside the kernel are performing `open` corresponding to which two key acquisitions have been initiated. The identifier field is used to determine which decrypted token goes to which thread. Currently, the identifier field is indexed by the pid of the calling thread inside the kernel. Therefore, before sending a request, the kernel fills up the *identifier* field with the *pid* of the calling process. Now, when the blinded PFK is received, its *identifier* is checked against the thread *pid*. Also, the *identifier* is used at the PKS to identify which session key needs to be used. The *uid* and *cert-id* field is used to identify which PKS is to be contacted by the daemon.The *uid* is the user-id of the user for which the request is made by the kernel. As each user may have multiple certificates which maybe located in different PKS, the *cert-id* field is used to match the correct PKS.

5.3.3 Packet - Structure and Handling

Corresponding to each of the service requests stated above, the kernel sends to the daemon the following packets. While the first is meant for the daemon, the other three are meant for

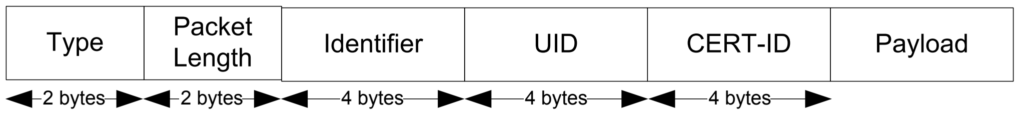

Figure 5.1: TransCrypt Header

the PKS. The last two packets are encrypted using the session key.

1. **TCPT_PKT_GET_CERT.** Certificate acquisition request
2. **TCPT_PKT_EST_SESS.** Session establishment Request
3. **TCPT_PKT_KEY_ACQ.** Key acquisition request
4. **TCPT_PKT_KILL_SESS.** Kill Session Request.

The following response packets are sent to the kernel in request to the above packets. While the first is sent by the daemon, the other two are sent by the PKS and are encrypted using the session key.

1. **TCPT_PKT_REPLY_CERT.** Certificate request response.
2. **TCPT_PKT_KEY_RESP.** Blinded PFK from the token.
3. **TCPT_PKT_KILL_RESP.** Confirmation of session invalidation.

In addition we also define a **TCPT_PKT_ERROR** which is sent by the PKS, or the daemon, in case of error in a received packet or during its processing.

During processing of packets at each end, sanity checks are made. If these checks fail at the kernel, then a corresponding error condition is returned to the calling function. If such an error is detected at the PKS or at the daemon, corresponding error packet is sent. Also, when kernel receives an error packet it returns an error to the calling function and sends a session invalidation request, TCPT_PKT_KILL_SESS. Session invalidation request is not send in the case of certificate acquisition request. The packets are processed as described below.

1. **TCPT_PKT_GET_CERT.**

   Structure: The payload has no additional data. The *uid* and *cert-id* from header is used to match the correct certificate. When the certificate acquisition is done during create, the *cert-id* is fixed by the kernel to 0.

   Processing: This packet is processed by the daemon itself. On receiving the packet, the daemon fetches the certificate after parsing the *uid* and *cert-id* field. If the *cert-id* field is 0 it fetches the latest certificate. The *cert-id* field is updated in the response packet. If the certificate is not available, the corresponding error packet is constructed and returned.

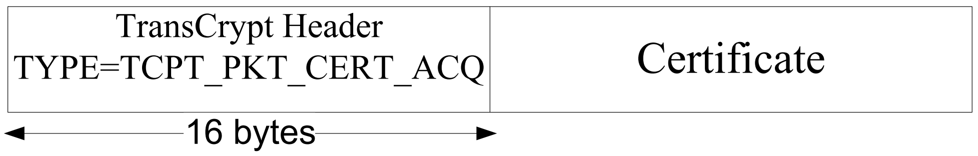

Figure 5.2: TCPT_PKT_REPLY_CERT Packets

2. **TCPT_PKT_REPLY_CERT.**

   Structure: The payload has the structure shown in figure 5.2. The certificate is the appended as a *char* buffer.The *cert-id* field indicates which certificate of the user is being sent.

   Processing: On receiving the certificate acquisition response, if the request had sought a particular *cert-id*(corresponding to an *open* request), and a mismatching *cert-id* is received, error is returned. If the certificate acquisition request was corresponding to the token creation call, then the *cert-id* field is updated in the metadata of the file. The certificate is then parsed and verified by the kernel. The parsed certificate is then used either to establish session or to create tokens depending on whether the request corresponds to `open` or `creat` system call respectively.

3. **TCPT_PKT_EST_SESS.**

   Structure: The payload has the structure shown in figure 5.3. The *uid* and *cert-id* of the header is used to establish session with the correct PKS. The rest of the payload contains a random session key encrypted using public key of the user.

Processing: The daemon forwards the packet to the PKS. The PKS decrypts the payload by using its private key as identified by *uid* and *cert-id* from the header. If it does not find a corresponding private key, an error packet is sent when the next token acquisition is requested. Otherwise, the PKS uses the session key till a session kill request is received or till another TCPT_PKT_EST_SESS packet is received for the same session.

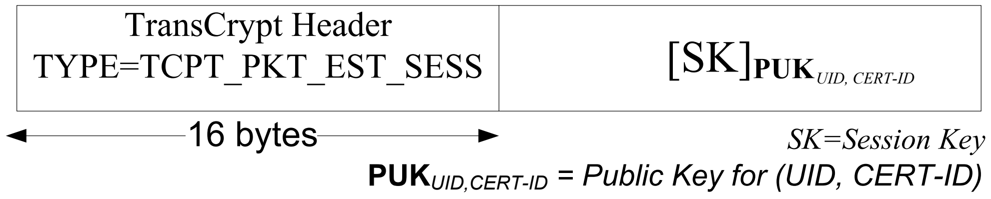
TransCrypt Header
TYPE=TCPT_PKT_EST_SESS
$[SK]_{\mathbf{PUK}_{UID,\ CERT\text{-}ID}}$
16 bytes
SK=Session Key
$\mathbf{PUK}_{UID,CERT\text{-}ID}$ = Public Key for (UID, CERT-ID)

Figure 5.3: TCPT_PKT_EST_SESS Packet

4. **TCPT_PKT_KEY_ACQ.**

   Structure: The payload contains the token encrypted using the SK established during the preceding session establishment request, TCPT_PKT_EST_SESS. The structure is shown in figure 5.4.

   Processing: On receiving the packet, the PKS uses the SK corresponding to the UID to decrypt and to retrieve the token. The private key identified by the *uid* and *cert-id* from the header is used to decrypt the token and obtain the Blinded PFK. The blinded PFK is then encrypted using the SK for the session and returned to the kernel in the TCPT_PKT_KEY_RESP packet.

5. **TCPT_PKT_KEY_RESP.**

   Structure: The payload contains the blinded PFK encrypted using the SK established between the kernel and the PKS. The structure is shown in figure 5.4.

   Processing: On receiving the packet, the kernel uses SK to decrypt the payload and retrieve the blinded PFK. The blinded PFK is then decrypted using FSK and the PFK is stored in the in-memory metadata. If there had been another PFK stored due to an earlier `open` call, this PFK is only checked for equality.

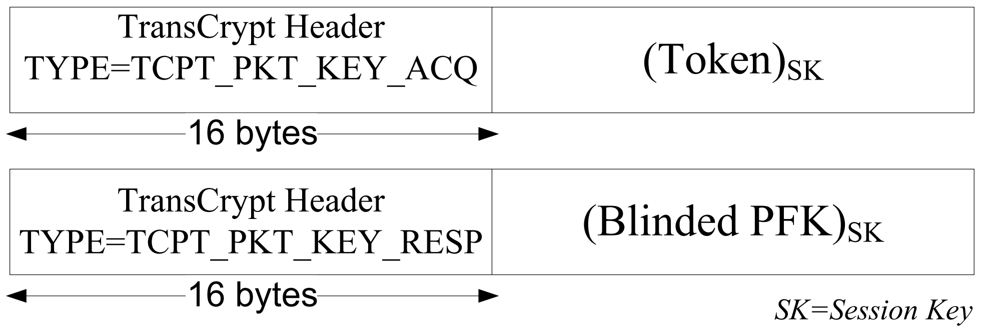

Figure 5.4: TCPT_PKT_KEY_ACQ and TCPT_PKT_KEY_RESP Packets

6. **TCPT_PKT_KILL_SESS and TCPT_PKT_KILL_RESP**
   These packets just have the header to indicate end of session. TCPT_PKT_KILL_SESS has a nonce value encrypted with session key to avoid someone in the middle from killing a session and launching a Denial-of-Service. On receiving a TCPT_PKT_KILL_SESS, the PKS invalidates the session key. Even if there is no corresponding SK entry, a TCPT_PKT_KILL_RESP is sent to confirm there is no stale SK entry.

The overall communication consists of a certificate acquisition, session establishment, key acquisition followed by session nullification. The steps are summarized in figure 5.5

5.4 Protocol Enhancements

The above protocol describes the current implementation of TransCrypt. However, some enhancements of the protocols are mentioned herein to make the protocol more efficient. For the rest of this discussion, we shall denote certificate acquisition packet by CA, Session Establishment packet by ES, Key Acquisition packet by KA and Session Nullification by SN.

Thus, currently while a token creation requires just a CA, a token decryption requires $CA \rightarrow ES \rightarrow KA \rightarrow SN$. However, since we do not expect any response from the server after ES, ES and KA may be packed in one packet. Further, we propose that the Session Key be stored for a certain lifetime instead of per key acquisition. The whole token decryption phase may then be summarized as $CA \rightarrow (ES\|KA)+ \rightarrow SN$. We also maintain a certain lock to allow batching of multiple KA, so that whenever we are sending off an $(ES\|KA)$ request, any

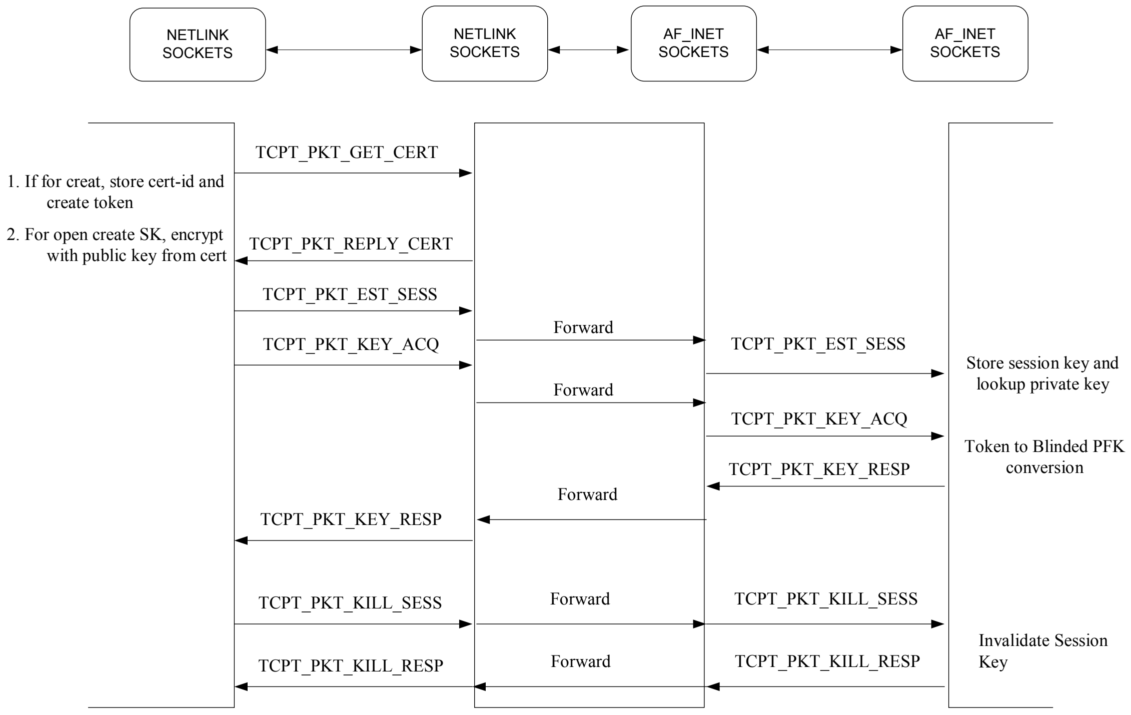

Figure 5.5: Communication Protocol between kernel, daemon and PKS

further KA requests for the same UID, may be batched together.

The final protocol can then be expressed by the following grammar. Here, '*' implies 0 or more occurrences and '+' implies one or more occurences.

$$CA \rightarrow (ES \| KA*) \rightarrow (KA+)* \rightarrow SN$$

Thus, some examples of valid protocol messages are

- $CA \rightarrow ES \rightarrow KA \rightarrow SN$
- $CA \rightarrow (ES \| KA) \rightarrow SN$
- $CA \rightarrow (ES \| KA \| KA) \rightarrow SN$
- $CA \rightarrow (ES \| KA \| KA) \rightarrow KA \rightarrow SN$
- $CA \rightarrow (ES \| KA \| KA) \rightarrow (KA \| KA) \rightarrow SN$
- $CA \rightarrow (ES \| KA \| KA) \rightarrow (KA \| KA) \rightarrow KA \rightarrow SN$

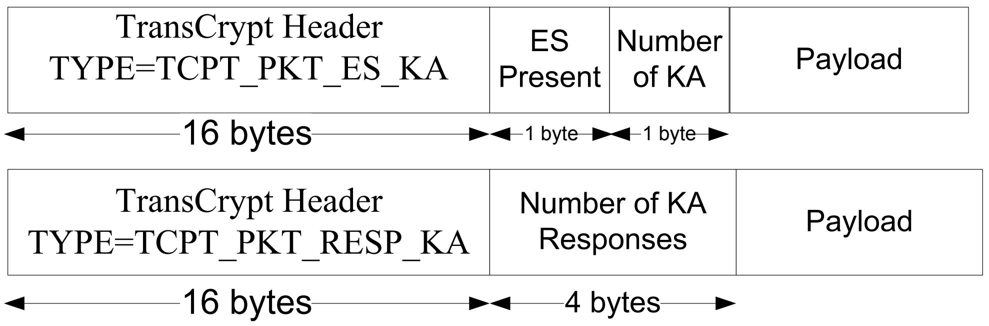

Figure 5.6: Unified Key Acquisition Packet, TCPT_PKT_ES_KA and unified reply packet, TCPT_PKT_RESP_KA

The current implementation supports only the first sequence above. In light of these enhancements, TCPT_PKT_EST_SESS and TCPT_PKT_KEY_ACQ packets. The structure of the unified packet is shown in figure 5.6. We call this packet TCPT_PKT_ES_KA. The payload is read as per the values of ES present and number of KA's in the packet. If ES is present, it should be at the beginning of the packet. It is then followed by each KA payload. The ES payload and KA payload is similar as the payloads of TCPT_PKT_EST_SESS and TCPT_PKT_KEY_ACQ packets respectively. In case of TCPT_PKT_KEY_ACQ each KA will be preceded by its own identifier to match it to the correct kernel thread. ES present being set to 1 and number of KA set to zero is similar TCPT_PKT_EST_SESS. Similarly, ES present being set to zero and number of KA being one is same as TCPT_PKT_KEY_ACQ.

Similarly, replies may be batched together for the same UID. The structure of the packet is shown in figure 5.6. The payload is the set of Blinded PFKs, appended with the identifiers to match them with the correct kernel thread. The entire payload is encrypted using the Session Key, SK.

Chapter 6

Putting Things Together

Our system has been built for GNU/Linux kernel version 2.6.20. WE have carried out certain changes in the kernel as outlined earlier. We also outline the changes implemented in the user space library for ACL manipulation *libacl* and the *mkfs* utility. The changes in *libacl* have been carried out by Mohan Dhawan [7].

6.1 ACL Based Changes

As stated in Chapter 3, we use the Access Control Lists (ACLs) to store the cryptographic metadata for TransCrypt. The current implementation of TransCrypt modifies the ext3 filesystem to work as a TransCrypt volume. The ACL data structure is therefore augmented for an ext3 volume to include, two additional fields called *e_certid* and *e_token* for each named user. The *e_certid* stores the certificate number which was used to create the token for the user, and *e_token* stores the actual cryptographic metadata.

As ACL entries may also be changed by user space utilities through *setfacl* and *chacl* commands. The user space library, *libacl* used has been updated by [7] for this purpose. ACLs are implemented inside the kernel as Extended Attributes (EAs). *libacl* makes system calls *setxattr* which takes a predefined structure as input. The changes made in *libacl* ensure that the structures passed have the modified ACL data structure.

The following changes are carried out in the *setxattr* system call to support the modified ACL behaviour and structure.

- If a new ACL is being created, it implies that the file has no cryptographic metadata.

The call shall then take the following path.

  - The kernel generates a random PFK
  - The PFK is blinded by using the FSK in the in-core copy of the super block
  - For each named user, the kernel acquires the user certificate from the user space and verifies it.
  - If verification succeeds, the public key from the certificate is used to encrypt the PFK to create tokens for each user.
  - The cert-id as returned by the user space daemon, and the token are updated in each ACL entry.

- If the ACL already exists, then the PFK has to be first extracted from the ACL entry of the owner. The following sequence of action is performed in this case.
  - If no new user is being granted access, the kernel just updates the permissions.
  - If there is a new user being added, the kernel sends the token to the PKS to extract the blinded PFK.
  - For each new user being added, a token is created using blinded PFK and a new ACL entry is created.

6.2 Changes in the Superblock

The superblock must provide mechanism to acquire FSK to be used for blinding and unblinding of PFKs during various filesystem operations. In addition, it contains information regarding the algorithms to be used for encryption and blinding among other details. Currently, the filesystem superblock has the following additional TransCrypt specific parameters.

- TransCrypt flag. Used to test whether the filesystem is TransCrypt enabled
- Symmetric Key algorithm used for PFKs, *pfalgo*. This contains a string specifying the algorithm, the chaining method and the key size, for example “aes-128-cb” denotes AES encryption algorithm with key size 128 bits and using cipher-block chaining(CBC) mode.

- Symmetric Key algorithm used for FSK, *fsalgo.* This has the same format as the *pfalgo.*
- IV generation method used, *ivmode.* Valid values are zero at present. Other intended values are *essiv* to be provided in future.
- Hash of the FSK. We use SHA-1 hash algorithm to compute the hash of FSK.

6.2.1 Utility for Filesystem Creation

As the current TransCrypt is an extension of ext3 filesystem, we change the e2fsprogs user space utility to create a TransCrypt enabled filesystem. It takes from the command line the following options.

- *tcpt* **flag**. The "–tcpt" option indicates that filesystem created should be TransCrypt enabled.
- *pfalgo.* The option takes the algorithm for encryption/decryption of files. Currently, only "aes-128-cbc" is supported. If the option is not specified, pfalgo defaults to "aes-128-cbc"
- *fsalgo.* The option takes the algorithm for blinding PFKs. Currently, only "aes-128-cbc" is supported. If the option is not specified, fsalgo defaults to "aes-128-cbc".
- *ivmode.* Currently only zero mode is supported for IV generation, i.e. IVs are zero.
- *FSK generation arguments.* These maybe either of the two options.
  - Passphrase.
  - Hash of FSK and salt value. These options are primarily intended for wrapper program which may take passphrase through other means.

When passphrase is entered, we use the key derivation function `PBKDF1` outlined in PKCS#5 standards [18]. We use the counter value as 1. The sequence used to derive FSK from passphrase is the following.

- The salt S generated randomly as an 8-octet string.

- A new passphrase, P' is created by concatenating the salt S to passphrase P. $P' = (S\|P)$.
- Let T_0 = $\mathbf{SHA1\text{-}Hash}(P'\|S)$.
- FSK is the first 16 bytes(or 128 bits) of T_0.

The SHA-1 hash of the FSK and the salt used is stored in the superblock.

6.2.2 Filesystem Mounting Utility

While mounting the filesystem, options are passed which are directly sent to the kernel. These options are used to determine whether the filesystem is being mounted with TransCrypt enabled or not. This does not require any changes in the mount utility. The options to be passed are the Passphrase, ACL enabled option and that the filesystem is being mounted as a TransCrypt volume.

6.3 Modifications in the Kernel

The modifications in the rest of the kernel essentially deal with reading cryptographic metadata, updating the in-memory data structures, and using them to encrypt/decrypt files. We, therefore discuss the changes under the headings of the file and filesystem specific operations.

6.3.1 Mounting

When a file system is mounted, the tasks required are: (a) to update the in-memory data structure with FSK and other parameters; (b) To determine whether the file system is TransCrypt enabled; and (c) Whether it is mounted as an encrypted volume or not. If the filesystem is being mounted as an encrypted volume, but the TransCrypt flag is not enabled in the superblock, then an error is flagged. A TransCrypt enabled filesytems may, however, be mounted as an unencrypted volume. The filesystem should also be mounted with the ACL option. If the ACL option is not set then the filesystem is considered to be mounted as an unencrypted volume.

The flags ensure that no TransCrypt operation take place if the filesystem is not a TransCrypt volume or if it is mounted unencrypted.

If the filesystem is TransCrypt enabled and is mounted as an encrypted volume, then the FSK is extracted from the passphrase from command line and salt from superblock by the same sequence as in section 6.2.1. The hash of FSK is then verified against the hash in the superblock to verify that FSK has not been tampered with. This prevents denial of service attacks that may be launched very easily. If the verification succeeds, the FSK is updated in the in-memory superblock. Also, information regarding algorithms, IV generation method is updated in the in-memory superblock after reading it from disk.

6.3.2 File Creation

When a file is being created, the following sequence of steps are carried out after the legacy permission checks have been performed while creating the ACL.

- A random PFK is generated.
- The PFK is blinded using the FSK from the in-memory superblock data structure.
- The named users and their permissions is borrowed from the default ACL of the directory in which the file is being created.
- For each named user ACL entry in the default ACL, tokens are generated as following.
  - Certificate for each named user is acquired by contacting the `daemon`.
  - The certificate is verified for correctness, and public key extracted.
  - Tokens are created for each user by encrypting the blinded PFK using the public key.
- The token and cert-id used are entered into each ACL entry.
- Since a `creat` system call is also considered as an `open` system call, the PFK is entered into the kernel keyring.

6.3.3 Opening a File

Opening a file requires extracting the token from cryptographic metadata, and extracting PFK from it. After legacy permission checks, the following sequence of steps are taken.

- The token is extracted from the user's ACL entry.
- The token is sent to the PKS and blinded PFK is obtained. The *cert_id* from the ACL is used to indicate which private key should be used by the PKS.
- The PFK is retrieved by unblinding using the FSK.
- If the file has already been opened by some other user the PFK retrieved is compared to the PFK stored in the kernel keyring. If a match is found access is granted, else access is denied
- If the file is not open, the PFK is entered into the keyring.

6.3.4 Reading/Writing a File

When a read/write is being committed, if the filesystem is TransCrypt enabled and is mounted as a TransCrypt enabled partition, the keyring is searched for a corresponding PFK for the inode. The PFK is then used to encrypt/decrypt blocks using the algorithms specified in the

6.4 Using TransCrypt

6.4.1 Setting Up

Setting up a system for TransCrypt requires setting up the right kernel image and updating user space utilities.

1. **Kernel image.** The TransCrypt kernel should be compiled with TransCrypt option on while kernel is being configured. This will also enable several cryptographic libraries, RSA crypto API, X509 module and libraries used by them. The kernel should then be compiled in the usual way and setup for being run at boot.

2. **Update libacl, mkfs.** The libacl and mkfs that comes with TransCrypt should be compiled and installed to be used instead of the corresponding utilities present in the kernel.

3. **Setting up *daemon* and *PKS*.** The *daemon* should be compiled and setup for running from */etc/inittab* with the *respawn* flag. The authserver may be setup either on the same system or a remote system. The corresponding IP addressed should be setup in the configuration of *daemon* and *PKS*.

6.4.2 Creating a TransCrypt Partition and Mounting

After creating a partition for TransCrypt through any partition utility like *fdisk*, the partition should be formatted by using TransCrypt's mkfs utility. The partition should then be mounted as with encrypt flag switched on and ACL enabled. For the first time, the ACL entries of the root of the TransCrypt should be setup, by using the *setfacl* command.

```
mkfs -t ext3 --tcpt --fsalgo=aes-128-cbc --pfalgo=aes-cbc-128 \
      --ivmode=zero --pass=<Pass-phrase> <device>
mount -o acl,tcpt,pass=<Pass-phrase> <device> <mount directory>
setfacl -d -m <ACL entry> <mount directory>
setfacl -m <ACL entry> <mount directory>
```

The TransCrypt partition is now setup for usage. It may be mounted as above in any directory as required. During the setfacl, it should be ensured that both the *daemon* and *PKS* are up and running.

6.4.3 File Lifecycle

In future, the filesystem should always be mounted with the above options if it is to be used as an encrypted volume. Also, the *daemon* and *PKS* should be running. If all of these are setup, then files may be transparently opened, read and written without any other action from the user. It should be noted that only installation and setup of TransCrypt requires a few additional actions by the user, while the actual file system operation has been kept totally transparent from a usage point of view.

Chapter 7

Future Work and Conclusions

7.1 Future Work

TransCrypt is currently at an advanced implementation stage. Some future goals of the project are enlisted.

- **Integrity Support.** Adding integrity support for files to ascertain if files have been tampered with or not. The metadata may also be signed to ensure that user tokens have not been changed.

- **Data Recovery Agent.** For TransCrypt to be enterprise deployable, it should provide recovery agent. The design of the recovery agent has partially been outlined in [21]

- **Backup Support.** TransCrypt, though implemented over ext3 filesystem, does not handle journaling modes. Investigating and designing backup and recovery is another goal for an enterprise deployable encrypting filesystem

- **Smart Card based PKS.** While TransCrypt has from its inception talked about smart card based key acquisition. Smart Card based PKS is a very important future work.

- **Group Support.** TransCrypt's design has no provision of Groups, an essential and useful feature of GNU/Linux and has wide ranging usage in enterprises. Group support addition should be done to TransCrypt keeping in view the issues of scalability while still maintaining the features of security, transparency and flexibility.

- **Integration with Trusted Platform Module.** The recent advent of Trusted Platform Module (TPM) should be incorporated in TransCrypt. These may be used to avert attacks like malicious kernel image or modules being loaded.

7.2 Conclusions

In this work, we have presented an enhanced architecture of TransCrypt, providing a comprehensive key management scheme. TransCrypt walks over the fine line between usability and security. The design is an attempt to provide flexibility, transparency with a heightened security paradigm. We identify certain attacks, notably daemon masquerading which TransCrypt is susceptible to. These attacks are of denial of service nature as opposed to attacks which may lead to data theft. Attacks leading to data theft is only possible through sophisticated techniques like changing the kernel image, screening the whole memory. We do not consider these attacks a threat for our aim which is to provide an enterprise class encrypting filesystem.

The major contribution of this work is implementation of a full file and filesystem lifecycle. It also presents implementation of user space utilities to support TransCrypt operations. Another contribution is cryptographic additions in the kernel which were needed for TransCrypt. However, these additions have been aimed at providing a common asymmetric API and public key infrastructure in the kernel space.

Bibliography

[1] Mick Bauer. Paranoid penguin: Bestcrypt: cross-platform filesystem encryption. *Linux J.*, 2002(98):9, 2002.

[2] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[3] V Bhanu Chandra. PKI for transcrypt. Technical report, Indian Institute of Technology Kanpur, Kanpur, 2006.

[4] V Bhanu Chandra. Transparent encrypted filesystem. Technical Report, Indian Institute of Technology Kanpur, Kanpur, 2006.

[5] Microsoft Corporation. Encrypting file system for windows 2000. Technical report, www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp, 1999.

[6] CryptoAPI. The GNU/Linux CryptoAPI, 2003. http://www.kernel.org.

[7] Mohan Dhawan. libacl for transcrypt. http://www.security.iitk.ac.in/home/transcrypt.

[8] dm crypt. A device-mapper crypto target for linux. http://www.saout.de/misc/dm-crypt/.

[9] Andreas Grünbacher. POSIX access control lists on linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 259–272, 2003.

[10] Mike Halcrow. ecryptfs: a stacked cryptographic filesystem. *Linux J.*, 2007(156):2, 2007.

[11] Ralf Hlzer. Cryptoloop. http://www.tldp.org/HOWTO/Cryptoloop-HOWTO/, 2004.

[12] LibTomCrypt. http://libtom.org/.

[13] Ermelindo Mauriello. Tcfs. *Linux J.*, 1997(40es):3, 1997.

[14] Andrew D. McDonald and Markus G. Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 462–477, 1999.

[15] Aleph One, Elias Levy, and Phrack Magazine. Smashing the stack for fun and profit. 8 Nov 1996, issue 49 article 14.

[16] R. Power. Computer crime and security survey. Computer Security Institute, VIII(1):1-24. www.gocsi.com/press/20020407.html, 2002.

[17] Arun Raghavan. Porting transcrypt from linux kernel 2.6.11 (fedora) to 2.6.20.1 (vanilla). Technical Report, Indian Institute of Technology Kanpur, Kanpur, 2007.

[18] RSA Data Security, Inc. *PKCS #5: Password-Based Encryption Standard*, June 1991.

[19] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, June 2002.

[20] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an ip services protocol. Network Working Group, RFC3549, http://www.ietf.org/rfc/rfc3549.txt.

[21] Satyam Sharma. Transcrypt: Design of a secure and transparent encrypting file system. M. Tech Thesis, Indian Institute of Technology Kanpur, Kanpur, 2006.

[22] Satyam Sharma, Rajat Moona, and Dheeraj Sanghi. Transcrypt: A secure and transparent encrypting file system for enterprises. In *Proceedings of the 8th International Symposium on Systems and Information Security, SSI 2006*, Sao Paulo, Brazil, November 2006.

[23] Deeptanshu Shukla. A daemon for secure smart card support in the encrypted file system transcrypt. B Tech Project Report, Indian Institute of Technology Kanpur, Kanpur, 2007.

[24] Varun Suresh, Shibin.K, Anoop.S, and Vivek.K.P. Magikfs. the steganographic filesystem on linux. http://magikfs.sourceforge.net/index.html.

[25] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/.

[26] Sainath S Vellal. Design and implementation of a kernel-userspace communication framework for transcrypt. Technical report, www.security.iitk.ac.in/home/transcrypt, Indian Institute of Technology Kanpur, Kanpur, 2007.

[27] C. Wright, M. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system, 2003.

[28] XySSL. http://www.xyssl.com.

[29] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system, 1998.