A daemon for secure smart card support in the encrypted file system Transcrypt

A report submitted in partial
fulfillment of the requirements
for the degree of
Bachelor of Technology

by

Deeptanshu Shukla

Under the guidance of

Prof. Dheeraj Sanghi
Prof. Rajat Moona

to the

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April, 2007

Certificate

Certified that the work contained in the report "A daemon for secure smart card support in Transcrypt" by Deeptanshu Shukla has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dheeraj Sanghi 21/5/07

Rajat Moona 3/5/07

Abstract

Transcrypt[1] is an encrypted file system which uses a smart card to store unique private parameters of users. This smart card is needed for all file operations by the user of Transcrypt. For accessing the smart card, a user space daemon is used. This daemon should be able to provide secure access to the smart card without compromising the level of security which Transcrypt aims to achieve. Transcrypt adopts a kernel space only approach and does not trust even the superuser. It prevents against both online and offline attacks. This project aims to explore the possible methods of developing this daemon and implementing the best case possible.

Acknowledgements

I would like to express my deep sense of gratitude to my supervisors Dr. Dheeraj Sanghi and Dr. Rajat Moona for their valuable guidance throughout the course of the project.

I would also like to thank the entire Transcrypt group especially Abhijit Bagri, Satyam Sharma and Mohan Dhawan for their invaluable ideas and constant inputs.

Deeptanshu

Deeptanshu Shukla

Contents

Introduction

Transcrypt is an encrypted file system for Linux which assumes a minimum trust model and provides a secure solution for data storage and sharing in an enterprise environment. It adopts a kernel-space only approach and protects against a wide threat model which includes both online and offline attacks. It does not even trust the superuser. It utilizes a user-space smart card daemon for key management.

Each file has an encryption key (K_{PFK}) associated with it which is generated during file creation. The K_{PFK} is needed during all file accesses. There is also a file system wide key (K_{FSK}). Each user has a key pair which is assigned to him during issuance of his/her smart card. The smart card carries the user's private key (K_{Pr}) while the user's public key (K_{Pu}) is used to create the token $K_{Pu}(K_{FSK}(K_{PFK}))$ which is stored as part of the file's ACL(Access Control List) entry corresponding to the user. The user's public key can be obtained from a public certificate. The ACL contains the list of users which have access to a particular file , the kind of access that they have and their tokens.

All file operations require the K_{PFK}. To obtain the K_{PFK} from the token from the file's ACL entry corresponding to the user demanding access to the file, the public private RSA key pair is needed along with the K_{FSK}. The private key is not present on the system but on the smart card and thus without it no file operations can take place.

Problem Statement

The private key parameters of users are stored on smart cards that act as their trusted tamper-proof hardware tokens. All computations involving the subject's private key are executed within the smart card. In order to prevent malicious users from getting access to encrypted files a secure mode of access to the smart card by the kernel must be devised. In other words, the K_{PFK} blinded by K_{FSK} must be obtained securely from the smart card.

The user's public certificate must be acquired to obtain his/her public key required for token creation. At the same time it must be ensured that the certificate used to obtain the public key of the user which tries to gain access, is genuine i.e., it must be verified by a certificate authority.

A valid token also needs to be created and stored in the file's ACL entry each time a new file is created.

The problem statement can thus be summarized as the development of a scheme of performing the following operations securely.

- Blinded FEK acquisition
- Certificate acquisition
- Certificate validation
- Token Generation

For this, a decision about which of these could be undertaken in the user space and which of these *must* be undertaken by the kernel needs to be made. The scheme should ensure that there are no reply attacks and man-in-middle attacks. Denial of service attacks are not a major concern in Transcrypt and beyond the scope of the current study.

It is later proved that the communication between the kernel and the smart card can be easily performed with the help of a user space smart card daemon and thus the main concern subsequently would be to come up with a schema of smart card access through the daemon which is secure enough for Transcrypt.

Motivation

There are two main kinds of operations which can be performed on files viz., file access and file creation. Interaction with the smart card needs to take place only for file access. In addition to this, smart card interaction is also needed when the owner of an existing file gives access to some other user. The ACL entries for a file decide whether the user has the read/write/execute access to the file(authorization) while the FEK verification from token entry using smart cards is an exercise to ensure that the person trying to access a file or grant its permissions to some other user is who he/she claims to be(authentication).

When a user tries to access a file, first a check is made to find out whether the user has an ACL entry for the file. In case the user doesn't have it, access is denied. If the user has an ACL entry, he/she needs to know K_{PFK} . This is obtained from the token corresponding to the user in the file's ACL entry. The token is sent to the smart card via the user space smart card daemon for decryption using the user's private key K_{Pr} which is stored on the user's smart card. The blinded file encryption key $K_{FSK}(K_{PFK})$ is returned to the kernel via the daemon. The kernel can obtain the K_{PFK} from $K_{FSK}(K_{PFK})$ as it knows K_{FSK} which is stored in the file system's superblock. In case the user is malicious, the smart card will not have the correct K_{Pr} and the K_{PFK} obtained will not be the correct one. Therefore the user will not be able to the original file contents but will see junk data.

Similarly when an owner of a file, say A wants to give file access permissions to some other user say B, A needs to have $K_{FSK}(K_{PFK})$ or K_{PFK} in order to be able to create a valid token for the B to be stored in the file ACL entry corresponding to B. For this, A's token is read from A's ACL entry in the file under consideration and sent via the daemon, to the smart card for decryption. The $K_{FSK}(K_{PFK})$ which is returned to the kernel via the daemon, is then encrypted with B's public key K_{Pu} to create his/her token $K_{Pu}(K_{FSK}(K_{PFK})$ and stored in the file's ACL entry corresponding to B.

During file creation and grant of file permissions by one user to another, additional work needs to be done viz., getting the public key of the user from the certificate which should be verified by a certificate authority and create a corresponding token for the user to be stored in the file's ACL.

Thus secure smart card access and certificate acquisition, certificate verification and token generation schemes is vital for the security which Transcrypt aims to achieve.

Related Work

This is not the first time smart cards are being used to store private user date for authentication; a number of such schemes do exist. Notably among these is the remote user authentication scheme using bilinear pairings suggested by Das, Saxena and others[2]. The user's smart card generates a dynamic login request and sends it to the remote system for login to the system. The login request is computed by the smart card internally. The remote system does not maintain any password or verifier table for the verification of user login request. Thus it allows the users to change their password freely and can protect against ID-theft, replaying, forgery, guessing, insider, and stolen verifier attacks. However Liao *et al* [3] have shown that some attacks are possible and have proposed some modifications to remove the weakness. Their scheme enhances the security and efficiency of Das *et al* without adding any computational costs. Another authentication mechanism proposed by Liaw, Lin and Wu[4] also avoids the use of verification tables and uses a similar mechanism for mutual authentication between the user and the remote system.

The Scheme

The four operations of concern are :

- Certificate acquisition
- Certificate validation
- Token Generation
- Blinded FEK acquisition

It must be decided which of these should be undertaken by the kernel and which in the daemon without compromising the security.

Certificate acquisition is the process of acquiring a user's certificate given his/her userid. Since all certificates are ultimately verified by a trusted entity, this action can safely be done by a user-space process.

Certificate validation is the verification of the authenticity of a user certificate and must be performed before extracting the public key out of it to be used for token generation. This is the most critical cryptographic operation. A malicious daemon can trivially and illegally verify invalid certificates and extract an illegitimate public key that would then be used to create an illegal token instead of a normal token for a user. If no scheme can be found which ensures absolute authenticity of the daemon, it would be safest to perform certificate validation in the kernel.

Token Generation is the production of user tokens to be stored in the file ACL. The user's public key must be known in order to create his/her token. Token generation cannot be done in the user-space because a malicious user-space daemon can trivially use an illegitimate public key to create a token to illegally grant access to an attacker when it was supposed to be created for a different user. Hence, token generation can be done securely only in the kernel.

Blinded key acquisition is the action of decrypting the token using the appropriate user's private key and retrieving the blinded file encryption key from it. It requires the user token to be sent to the smart card or authentication server for decryption and returning the blinded key to the kernel. Because the actual decryption always takes place on a trusted end point (smart card or auth server), the only role to be performed for this activity is routing the token to the appropriate end point, provided an end-to-end authenticated and encrypted session is established between the kernel and the other

trusted end point. Hence, we can safely use an unauthenticated user-space daemon to act as the conduit between the kernel and the smart card (or auth server). Clearly, this requires a key pair (with certificate signed by any CA trusted by the organization) to be associated with the kernel also. A certificate exchange based challenge-response protocol must be used for mutual authentication between the kernel and the smart card (or auth server) followed by session key establishment, such as the following:

1. Kernel sends its certificate to the smart card (along with certificate chain of the intermediate CA hierarchy till the root CA), which verifies it.
2. Smart card sends its certificate (the user's certificate) to the kernel (along with intermediate CA certificate chain) which verifies it and ensures that it is the same certificate as expected for the current user context.
3. Kernel sends a challenge (random nonce encrypted with public key extracted from smart card's certificate) to the smart card.
4. Smart card decrypts the challenge and sends back its response (the plain random nonce) back to kernel, along with its own challenge (random nonce encrypted with public key extracted from kernel's certificate).
5. Kernel sees the response from smart card and if satisfied (successful authentication of smart card, implying that is indeed a genuine user with a genuine smart card) decrypts the smart card's challenge and sends back its response (the plain random nonce generated by the smart card), along with a random session key SK encrypted with the public key of the smart card (user).
6. The smart card verifies the kernel's response (to complete the mutual authentication) and if satisfied decrypts the random session key SK. Both the kernel and the smart card use this SK for encrypting all communication henceforth.

Thus, a mutually authenticated and encrypted secure channel has been established between the kernel and the smart card (or auth server) which can be used to send the token to the other end point to be decrypted and get back the blinded FEK. This removes man-in-middle attacks as the session key is known only at the ends and any intermediate entity cannot listen to the conversion as it doesn’t have the session key and cannot obtain the same. Also there are no replay attacks as the key is temporary for a session. So any messages which are logged by a malicious daemon become useless in the next session (next access attempt on the same file).

A number of other schemes to be followed for secure communication between kernel and smart card were also explored, all of which have possible loopholes in security.

The use of timestamps was explored so that replay attacks by the daemon can be prevented. For this a timestamp had to be added with the per file key each time each time it is obtained from the blinded FEK. This is then encrypted with the file system key and then the user's public key to form the new token. Hence each time a file access takes place, a new token is generated which has a timestamp embedded into it. Thus replay attacks are avoided. However this scheme fails during grant of file permission.

It was also suggested that in order to validate the authenticity of the daemon, a hash of its binary could be taken and then signed with a private key which is known only to the kernel. This private should then be discarded. Whenever the daemon is loaded, the kernel can verify its authenticity by decrypting the signed hash using its public key which will be encoded into the kernel itself. This scheme however totally ignores run-time attacks on the daemon. By signing only the hash of the daemon's binary, all that is being prevented is that the binary doesn't change from below after installation. Man-in-middle and other run-time attacks on the daemon are still possible. These being trivial for an attacker with superuser privileges. Any superuser can trivially subvert any userspace process at runtime (which includes such things as modifying memory space, variables, inserting breakpoints, etc).

Another scheme was discussed where the daemon and kernel share a public and private keypair based security association so that all certificate verification can be performed by the daemon which then sends a modified temporary certificate signed by itself to be then verified by the kernel. However, this scheme is also insecure because it requires the daemon to be trusted which is not possible because it can be easily subverted to do malicious actions at runtime.

Because of the wide range of attacks and ease with which user-space processes can be completely subverted, such schemes were identified to be insecure.

It was thus later decided that certificate verification take place in the kernel itself to avoid the need for an authenticated daemon. Thus in the final scheme, the daemon need not be authenticated as it was realized that the worst a malicious daemon could do (provided we have a secure end to end session and only the intermediary i.e., the daemon is corrupt) was denial of service which is beyond the scope of security which Transcrypt offers. It cannot in any case get access to data it is not authorized to.

The Implementation

The communication between the user space and kernel space has been implemented using netlink sockets.

For the communication to take place, both the daemon and the kernel must know whom to talk to. The daemon knows the process id (pid) of the kernel which is always 0, but the kernel doesn't know which process to talk to. For this reason the messaging was previously implemented as user space/daemon initiated unicast communication . As soon as the daemon is started, it sends a *hello* message to the kernel saying "I am the daemon and this is my pid, now you can talk to me". Having known the pid of the daemon process, the kernel can now communicate with the daemon (send the token, etc). This pid needs to be remembered. Once the daemon has informed the kernel about its pid, all further communications are kernel initiated; this is because, whenever a file is created or accessed, a portion of the kernel code is executed and it is this code which contact's the user process.

An alternate approach was later adopted in which the messaging was a kernel initiated multicast communication. In this case there is no need for the daemon to inform the kernel about its pid. The daemon is registered as a member of a group which listens to kernel's messages specific to file creation/access. The kernel sends all relevant messages to this group only.

It must be noted that the daemon process starts immediately after the kernel boots up and must be running all the time in order to enable file accesses. In case the user tries to access a file before the daemon starts (which is not a common scenario) access will be denied.

To simulate a smart card an auth server has been used. The auth server has been designed in a way that would make the shift to a smart card handler program least painful.

Openssl library has been used to implement the various cryptographic operations performed at the auth server. 128 bit RSA has been used as the assymetric encryption algorithm for encryption-decryption with the user public-private key pair, while 128 bit

AES has been used as the symmetric encryption algorithm for encryption-decryption with session key.
The implementation can be described under three topics:

1. Messaging/Communication
    a. Kernel space and user space daemon
    b. User space daemon and auth server
2. User space daemon
3. Auth server

The exact implementation details being standard netlink socket communication are not described in this report for brevity.

Messaging/Communication

Message Structures

A number of different kinds of messages need to be exchanged between kernel, daemon and the authserver. A generic message structure for all such messages was designed.

The message structure is illustrated below:

ID identifies the kind of message, LENGTH gives the total size of the message, PID is the process id (this will have to be changed to something more unique for each request, for example a file pointer) and DATA is the message payload.

ID,LENGTH and PID fields are of 2 bytes each while the DATA field can be as large as 1600 bytes.

As mentioned earlier, instead of having a daemon initiated unicast communication between the daemon and the kernel, a kernel initiated multicast communication is now being used and thus the hello message (described below) is deprecated.

The following packets are being used.

1. *dpkt_hello* **[deprecated]**
   Hello packet with id TCPT_PKT_HELLO from **daemon to kernel** to initiate communication.

2. *dpkt_cert_acq*
   Certificate acquisition request packet with id TCPT_PKT_GET_CERT from **kernel to authserver** (via daemon) requesting user certificate with userid as the payload.

3. *drpkt_cert_resp*
   Certificate acquisition response packet with id TCPT_PKT_REPLY_CERT from **authserver to kernel** (via daemon) with the user certificate as the payload

4. *dpkt_est_sess*
   Session establishment packet with id TCPT_PKT_EST_SESS from **kernel to authserver** (via daemon) with a random session key as the payload.

5. drpkt_ack_sess
   Session established acknowledgement packet with id TCPT_PKT_ACK_SESS from **authserver to kernel** (via daemon)

6. *dpkt_challenge_auth*
   Packet initiating challenge-response to verify the authenticity of the authserver with id TCPT_PKT_CHALLENGE_AUTH from **kernel to authserver** (via daemon) with a random number (challenge) as the payload

7. *drpkt_response_auth*
   Response packet for the challenge, with id TCPT_PKT_RESPONSE_AUTH from **authserver to kernel** (via daemon) with challenge+1(response) as the payload

8. *dpkt_key_acq*
   Key acquisition request packet with id TCPT_PKT_KEY_ACQ from **kernel to authserver** (via daemon) with the session key encrypted token as the payload

9. *drpkt_key_resp*

Key acquisition response packet with id TCPT_PKT_KEY_RESP from **authserver to kernel** (via daemon) with the session key encrypted blinded FEK as the payload.

User Space Daemon

The daemon forwards all messages from the kernel to the authserver and vice versa. An exception being the certificate acquisition packet which is not forwarded. Instead a reply packet is sent to the kernel with the user certificate as the message data. Communication between kernel and daemon uses multicast netlink while the communication between daemon and authserver takes place using the common c sockets.

Methods to detect a daemon death and take appropriate action were explored. Time outs could be used to resolve this issue. The kernel could wait for a fixed time duration after sending each message within which it expects to receive a reply. However this might sometimes lead to wrong prediction of the daemon's death in case the delays are either at the authserver or due to congestion. It must also be noted that netlink not being a reliable protocol, some packets might even be dropped. Another method using a cron daemon was also explored but finally it was decided to create an entry into the /etc/inittab file (used to schedule processes periodically and take appropriate action when the process gets killed) so that the daemon can be respawned on death. It must be noted that we need not be concerned about a malicious daemon as it can only lead to denial of service attacks which are not our concern. However since we have ensured that the daemon will be restarted the service denial cannot last long. Thus even denial of service attack is taken care of unless the malicious daemon process somehow gets hold of superuser privileges and changes the entry in /etc/inittab.

.

Authserver

The Authserver replies to the messages sent to it from the kernel through the daemon according to the message type. It strips the message and reads the ID to know the message type. It then takes the appropriate action accordingly. It implements the various cryptographic operations which need to be performed at the smart card. This includes decryption of the token using private key accessible to it using RSA and encrypting the reply message data with the session key using AES. In case of unrecognized packet type, it sends an error packet.

Three kinds of packets are handled at the Authserver:

dpkt_challenge_auth

Packet initiating challenge-response to verify the authenticity of the authserver with a random number (challenge) as the payload. The response packet *drpkt_response_auth* with challenge+1(response) as the payload is sent back to the daemon which forwards it to the kernel.

dpkt_est_sess

Session establishment packet with a random session key as the payload. Session established acknowledgement packet *drpkt_ack_sess* is sent back to the daemon. Session key maintenance is implemented at the Authserver using a two dimensional array having the session keys alongwith the PID of the process (to be changed later to file pointer which is unique to each request) and UID of the user accessing the file. The array is indexed on request number, a number which is incremented each time a certificate is acquired. Each time a new request for key acquisition arrives, a check is made to find out whether a session key was established for the PID by searching through the array starting from the current request number (This increases the efficiency of the search as in most cases the key acquisition request will be just preceded by a certificate acquisition request).

dpkt_key_acq

Key acquisition request packet with the session key encrypted token as the payload.
Key acquisition response packet *drpkt_key_resp* with the session key encrypted blinded FEK as the payload is sent back to the daemon. The session key used for encryption is read from the table of session keys.

A new packet "*dpkt_session_end*" was also introduced later as a clean way to end a session. The action performed by the authserver on receipt of this packet is to delete the session key entry corresponding to the request from the session key table(array).

Future Work and Conclusion

Currently, though the authserver has been implemented to take care of multiple requests, the daemon does not have any such provision. Thus multiple file accesses at the same time would give an error. Multiple requests can be handled in a number of ways. Two of which are :

- Use of threads
- Use of a multiple (but limited number) of sockets

The daemon must remember which socket sent which kind of packet and send the reply obtained from authserver to the appropriate socket. Thus a mapping between socket descriptor and the packet received at it will have to be maintained in the form of a table. These sockets can either all be open during daemon start or may be opened on demand (if there are never more than two requests, only two sockets will be open). The latter was decided upon the as the scheme which should be implemented. The implementation was started but as with all software implementations, has its share of issues which couldn't be resolved due to time constraints.

The proposed scheme for communication between kernel and smart card (authserver) has been tested to be free from security loopholes and does not in any way reduce the level of security provided by Transcrypt. The end to end session key establishment method can in general also be applied to similar applications which require a secure access to some device by the kernel without the overhead of having the implementation inside the kernel.

Bibliography

[1] Satyam Sharma, "TransCrypt: Design of a Secure and Transparent Encrypting File System", *MTech Thesis, Department of Computer Science and Engineering , IIT Kanpur,* 2006.

[2] Manik Lal Das, Ashutosh Saxena, and Ved P. Gulati, "A Dynamic ID-based Remote User Authentication Scheme," *IEEE Transactions on Consumer Electronics, vol. 50, no. 2, pp. 629–631,* 2004.

[3] I-En Liao, C. C. Lee, and M. S. Hwang, "Security Enhancement for a Dynamic ID-based Remote User Authentication Scheme," *IEEE CS Press, International Conference on Next Generation Web Services Practices (NWeSP'05), pp.437 440,Seoul,Korea,* 2005.

[4] H.T. Liaw, J.F. Lin, W.C. Wu. "An Efficient and Complete Remote User Authentication Scheme using Smart Cards". *Article "Mathematical and Computer Modeling", Volume 44, Issue 1-2, pp 223-228,* 2006.