Operating an Escrow Document Storage and Secure Signing Registry

Dr Craig S Wright

Abstract

We propose a system, method, server processing system, and computer-program product for operating an escrow document storage and secure signing registry. In one aspect, the server processing system is configured to: receive, from a user processing system in data communication with the server processing system, data that is to be securely stored and maintained on the server for the user. Such data will be encrypted in a manner that escrowed keys can be used to access data in the event that a third party is to access the data without the individual's signing and encryption key (e.g. for access from the executor of an estate or a liquidator for a corporation). The data will be stored in a time-stamped and digitally signed format to prove the integrity of the document in a manner that cannot be altered. The document will be able to be signed by external parties who can validate the authenticity of the document without having to read its contents.

Table of Contents

Description of Art

The system allows for the decryption of documents on a set of triggered events based on the commonly held properties of a secure escrow system. At the same time, the integrity of all documents is maintained with documents being digitally signed and the digital signature attached to the document being maintained with a system signature in a database (following a timestamp).

The escrow system should have the following properties:

Property 1: Each user in the system has sufficient control over his or her secret key to be sure that the key is chosen securely. The user maintains a secret key that they control. It can be stored in a number of formats (less to more secure):

- The encryption key is stored in a database run by the registry. It is encrypted using a 256-bit AES key, and the encrypted encryption key is maintained to be decrypted by the user, who can then load it into a temporary memory cache to encrypt his data.
- The user has a secure store on a USB or another semi-offline device to store his keys.
- The user has a smart card containing his cryptographic keys.

Property 2: The central authority ensures that the secret key for each user is chosen securely even if the user doesn't have access to a good random-number generator or if the user fails to use the random-number generator properly. The password/passphrases they choose are analysed, and the user is not allowed to select a poor passphrase that could be easily cracked.

Property 3: Each user is guaranteed that his or her secret key will remain secret unless a sufficient number of trustees release their shares of the key to the central authority.

- Once the central authority releases such details, the data maintained by them cannot be used to remove or alter the signed document.
- All document versions are maintained in a database (even if the original document is removed).

Property 4: The central authority can assure the users that it can only obtain the secret key for a user who is suspected of using his or her escrowed public key for encryption in the context of illegal activities or when a trigger event occurs by retrieving shares of the key from a certain number of trustees.

Property 5: The central authority can ensure that the escrow system will not be abused by criminals in a way that helps them to communicate without fear of court-authorized wiretapping. More precisely, if two criminals abuse the FKE by using the information contained in their public keys to communicate using any published public-key encryption algorithm, and the central authority is provided knowledge of the criminals' escrowed secret keys by the trustees, then one of the following two cases should hold:

1. It should as easy (at least on a probabilistic basis) for the central authority to decrypt the message traffic between the criminals as it is for the criminals themselves to decrypt the traffic; or

2. the criminals already had a way to communicate that could not be decrypted by the government.

The System

The system is a secure escrowed document store that allows documents to be recovered but not to be altered in the system.

The user can save files in a number of folders. Such folders can be set with separate permissions. For instance, a user can create a set of folders such that it creates the following:

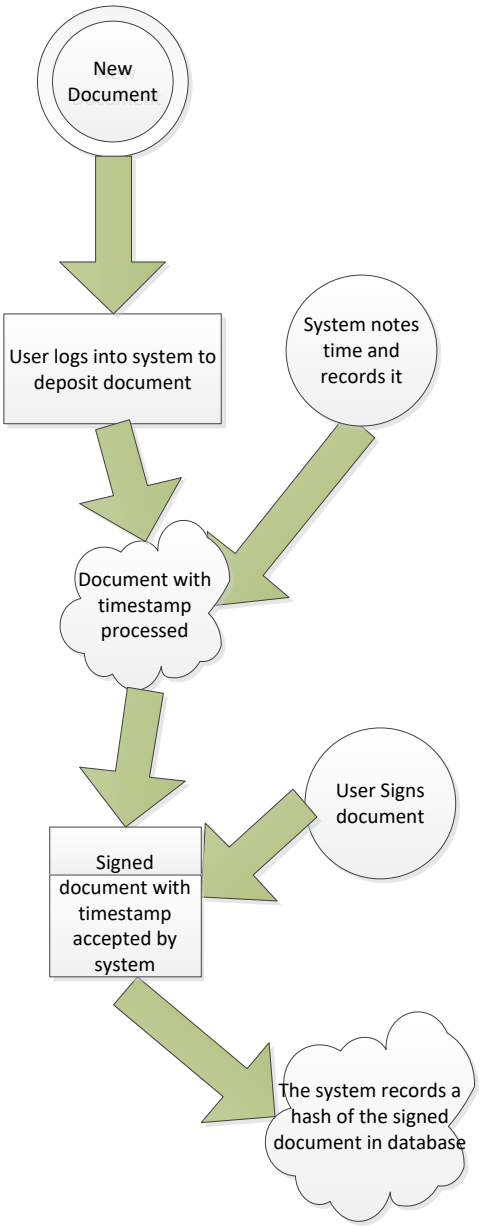
1. A folder that cannot be deleted or changed by the user (e.g. for tax and receipts):
    a. Such a folder is a store for data that must be maintained and cannot be changed.
    b. Files are time-stamped and maintained.
    c. The integrity of all files in the folder can be proven without having been seen by any other individual.
    d. The signing process is secret.
2. A folder that cannot be altered, only to the extent that a new version of a document is placed into the folder:
    a. Such a folder could be used as a registry-file holder.
    b. Users could create documents (such as a will), sign them, and have the documents witnessed without having them read.
3. A shared-document folder:
    a. Here, multiple parties can sign and maintain a document (such as a contract).
    b. All changes will be recorded and noted (including any following the application of a signature).
    c. Documents can be selectively escrowed.

In such a system, the private encrypting key of the user is signed and escrowed. The method of escrow is to use the private key of the user and the escrow party to protect the keys.

A database of both signing keys, the user's public signing key for the verification of the document and the public encryption key of the user, such that he can decrypt his own documents at will is maintained.

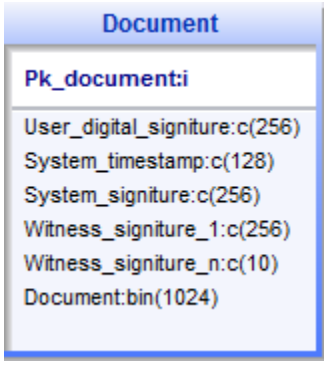
Users can maintain multiple key identities with separate signing keys based on their respective roles (individual, company director, tax authority, etc.).

Signing Documents



Documents loaded as signed (but not encrypted) are stored in the database as follows:

- A field will be recorded for any signatures (hash values) attached to the user and any witnesses.
- The timestamp of the document is included.
- The document as a binary blob (which can also include the encrypted document) is stored.



Recoverable and Certifiable

The system will be a recoverable and certifiable crypto system. Users of the system will be able to act in a manner that allows a user to generate certifiable keys efficiently. The following is the formal definition:

The system is a Recoverable Certifiable Cryptosystem based on a $5-tuple$ (GEN, VER, REC_1, REC_2, $REC3$) such that:

1. GEN is a publicly known poly-time probabilistic Turing machine that takes no input and generates the triple (K_1, K_2, *P*) which is left on the tape as output. Here, K_2 is a randomly generated private key, and K_1 is the corresponding public key. *P* is a poly-sized certificate that proves that K_2 is recoverable by the escrow system using *P*.
2. VER is a publicly known poly-time deterministic Turing machine that takes (K_1, *P*) on its input tape and returns a Boolean value. With very high probability, VER returns *true* if *P* can be used to recover the private key K_2.
3. REC_i, where $1 \le i \le m$, is a private poly-time deterministic Turing machine that takes *P* as input and returns share *i* of K_2 on its tape as output, assuming that K_2 was properly escrowed. The Turing machines REC_i for 1 ≤ i ≤ m can be used collaboratively to recover K_2.
4. It is intractable to recover K_2 given K_1 and *P* without REC_1, REC_2, REC_3.

The certification authority (CA) will not publish a public key unless it is verified that the corresponding private key is escrowed properly. Let EA_i denote Escrow Authority i. It is also assumed that EA_i knows only REC_i, in addition to what is publicly known. Our system is used as follows:

To publish a public key, user *U* runs GEN and receives (K_1, K_2, *P*). *U* keeps K_2 private, and encrypts the pair (K_1, *P*) with the public key of the CA. *U* then sends the resulting cipher text to the CA. The CA decrypts the value, and recovers (K_1, *P*). The CA then computes $VER(K_1, P)$, and publishes K_1 in the database of public keys if the result is *true*.

The certificate P is not published.

Where *U*'s public key is accepted and K_1 appears in the database of the CA, given *P*, the escrow authorities can recover K_2 as follows: EA_i computes share *i* of K_2 by running $REC_i(P)$. The authorities then pool their shares and recover K_2.

Public Escrow Verification

VER takes ((*y*, *g*, *p*), *P*) on its input tape, and outputs a Boolean value. VER verifies the following things:

1. $C^{b_i} C_i = z_i^{\,3} \bmod (2tn) \quad for \quad 1 \le i \le N$
2. $v_i = \left(y^{1-b_i} g^{b_i}\right)^{z_i} \bmod p \quad for \quad 1 \le i \le N$

VER returns *true* if both criterions are satisfied. Note that sceptical verifiers may also wish to check the parameters supplied by the escrow authorities (e.g., that *n* is composite, *p* is prime, etc.).

Key Recovery

REC_i recovers share *i* of the user's private key *x* as follows: REC_i takes *C* from *P*. It then recovers share *si* using the private share *di*. It outputs *si* on its tape. The systems then pool their shares, and *x* is computed. Criterion 3 of definition 1 is therefore met.

Recovering Plaintext Data

The system will recover the plaintext of users' files based on a variety of triggers (such as the user's death in an estate registry, or if an individual is suspected of criminal activity and a warrant is issued from an authorised state authority) without recovering the user's private key itself.

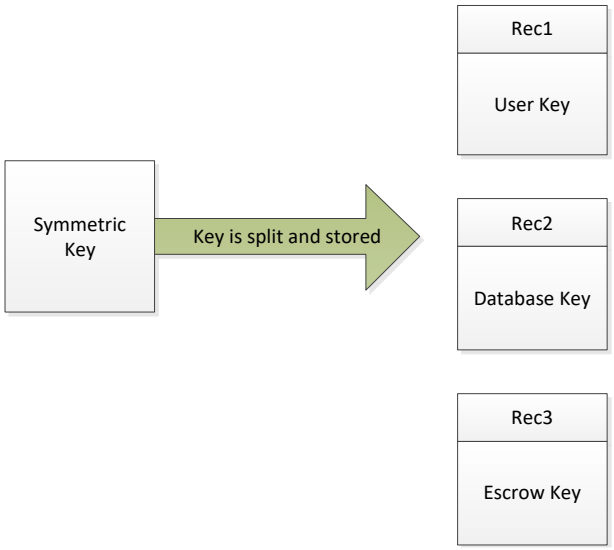
In the system, the private decryption exponent is $d = \sum_{i=1}^{m} d_i \bmod \phi(tn)$, and d is the inverse of $3 \bmod \phi(tn)$. To decrypt the El Gamal cipher text (a,b) of a user U, the escrow system will process the request as follows:

1. Each of the m escrow authorities receives C corresponding to U.
2. Escrow authority 1 computes $s_1 = a^{C^{d_i}} \bmod p$.
3. Escrow authority $i+1$ computes $s_{i+1} = s_i^{C^{d_{i+1}}} \bmod p$.
4. Escrow authority m decrypts (a,b) by computing $b \Big/ \left(s_{m-1} C^{d_m}\right) \bmod p$.

No one can recover x unless an event has triggered the release of the values C^{d_i} from the escrow system.

The process of lodging a document (user *U* saves a document in a protected folder) proceeds as follows:

1. *U* signs a document using their signing key,
2. The system appends a timestamp to the document, and signs the document using the knowledge-free system.
3. The document is encrypted by *U* who uses his escrowed public key to encrypt the document.



REC1 Stored either in the user system or database encrypted using the user's private key.

REC2 Stored by the system and protected using the system key.

REC3 Activated for escrow purposes. REC2 and REC3 can be used in conjunction to decrypt the document but not to alter the signature.

REC1, REC2, and REC3 hold overlapping sections of the symmetric key with at least 2 of the three keys being required to decrypt the document.

Signed without Knowledge

Using the system, the document is encapsulated in the database with a timestamp and the digital-signature value of the user *U* (computed cryptographically).

The system also allows for witnessing from untrusted parties. Such parties can be used to validate the signing of a document and to ensure that a document remains unchanged. They can incorporate their signatures into the database, validate the document, and test if any changes have occurred to the document without having any knowledge of the data contained within the document.

It is a zero-knowledge system.

Identification

A scheme such as one proposed by (Leighton & Micali, 1989) can be implemented on a smart card.

Such a card can maintain the user's identity, keys, and other data.

Safeguards for Escrowed Keys

The KEC employs safeguards to protect against the compromise or loss of keys. Such can include a combination of technical, procedural, and legal safeguards. Examples are auditing, separation of duties, split knowledge, two-person control, physical security, cryptography, redundancy, computer security, trusted systems, independent testing and validation, certification, accreditation, configuration management, and laws with penalties for misuse.

Check Triggers

Database triggers are procedural code that is automatically executed in reaction to selected events of a particular table, row, or field in a database. The system triggers to be set to fire when events that are defined in the policy occur. Such triggers are defined below.

System Triggers

System triggers allow the activation of controls that start when system events take place. Such events can include:

- the start-up and shutdown of the database;
- the logon and logoff from users;
- privileged access; and
- the creation, altering, and dropping of schema objects.

Autonomous transactions will allow a log to be written for the above system events. The database security will check what (if any) system triggers exist and ensure that they are aligned with the policy of the user's store. An example trigger would be sending an alert if a user with administrative access has been added to the database.

Update, Delete, and Insert Triggers

Defence in depth requires an understanding of the user's actions at multiple levels. It covers not just access to the database but access at the detailed row level for selected events and where there is sensitive data. Database triggers will capture changes at the column and row level. All data changes will be recorded. The database will be configured to write entire rows of data detailing a change to the data (who, what, where, and why). It can be done both ahead of and subsequent to the modification of data being made with a write of information to a log table in the database and to an alternate location.

Fine-Grained Audit and Review

A fine-grained audit will be based on internal triggers that react when selected SQL code is parsed. Such an approach allows a reviewer to perform access reviews at the row and column level—for both changes and read statements.

Validate Access

The user will be able to check out who has access to the database (and even what tables, rows, and fields someone has access to) and files he maintains. Checking access requires that the audit verify

access location and time (where and when). Logon failures should also be checked with seemingly legitimate access at out-of-the-ordinary or anomalous times (such as for access to a local payroll system at 3 am on a Sunday morning).

Auditing Changes to the Database Structure

Production databases should **NEVER** allow **ANY** user to alter the schema structure. Changes should only be done (such as for upgrades) at definite times (that are logged and approved through change control). All other changes should be regarded as suspicious. Any privileges allowing so must be reviewed carefully. An examination of the database logs for evidence of structural changes can uncover evidence of invalid or unauthorised use of the database.

Monitor any Use of System Privileges

It is one thing to check the configuration of a database; it is another altogether to validate that access has been corresponding to a configuration file over time, or indeed if the database is reacting as it should. Logging to a separate system is therefore critical. If the DBA and system-administration function lie with the same person, it is possible to remove evidence of changes to the system.

Separate logs provide the capacity to check if either an attacker or a rogue DBA has made any authorised changes to the database.

Log and Record Data Changes to Objects

Such requirements are very application and installation specific. Here is where the security tester needs to know what he is doing, and why. Such a type of review needs to be purposeful and objective. It is easy to exceed the scope of an object-access audit, and in such an event, it is also possible for the testers to breach the law themselves (for instance, in gaining an unauthorised view of health information).

Failed Log-On Attempts

Check for attempts to gain unauthorised access the database (and ensure the logs are available).

Attempts to Access the Database with Non-Existent Users

Such could be an attempt to bypass the controls in place over the system.

Check for Users Sharing Database Accounts

Non-repudiation hinges on not sharing accounts and access. Shared accounts are the anathema of a secure system, and there is no compliance regime that allows such practice. Checks will disable shared access to the system requiring a reset of the account.

Multiple Access Attempts for Different Users from the Same Terminal

Check if multiple database accounts have been used from the same terminal—which can indicate compromised access or shared access.

Appendix

Encryption

Hashing and encryption are similar and related but not the same thing. Hashing is a one-way function that takes data and provides a cryptographic fingerprint of the data that cannot be reversed and uniquely identifies the information to the fingerprint. Encryption is reversible. The use of a key will either lock or unlock the data, protecting it from prying eyes.

What Is a Hash

With conferences and the like, I have been behind in writing up a load of material on topics such as NAP etc. I have not forgotten them. For now, I will start by detailing some terms I have seen being used poorly.

To start, I will look at what a hash function is.

Formally, a hash function *H* is defined as a transformation that takes a variable-size input *m* and returns a fixed-size string. The fixed string is what we term the hash value *h*.

We can express the function as: ***h = H(m)***

There are some things we need to know in developing our function *h*:

- The input *m* can be of any length (which includes being smaller than the resulting hash output *h*, larger than the output, or even of the same size).
- The size of the output *h* remains the same no matter what the input is. In other words, if the output of the hash function returns a length of L bits for a given input *m*, it must return an output of length L bits for ANY input *m*.
- The hash function *H(m)* is 'one way.' If we have a value *h*, we cannot use it to determine the initial input *m*.
- The function, *H(x)*, must be simple and computationally inexpensive to compute, such that making a table of mapped values and indexing calculated hashes against inputs must be expensive relative to making the hash.

Hashing is used primarily in digital signatures and for integrity checks. They also aid in time stamping.

Collisions

It is said that a hash needs to be "collision-free." Which is wrong. If we think that an 8-bit hash has only 256 possible values, we see that there is an infinite number of collisions as we can have an infinite variability in input. Collisions abound!

Collisions always EXIST IN A HASH FUNCTION. They are NOT the issue. The issue is not the existence of a collision, but the fact that we may be able to calculate the collision and predict it.

A hash function *H(x)* will have collisions, but the distribution of such collisions should be unpredictable.

If we constrain the length of the input *m* to a certain value, the number of collisions can be said to be in the order of:

$$\frac{\text{no. input messages possible}}{\text{no. hashes by hash length}}$$

where:

- no. input messages possible = 2^(length *m*); and
- no. hashes by hash length = 2^(hash length).

Collisions exist.

References

Leighton, F. T., & Micali, S. (1989). USA Patent No. 4879747. USPTO.