

Smart Cards for the FlexiPKI Environment

Michael Hartmann¹ · Sönke Maseberg^{1,2}

¹Technische Universität Darmstadt
{ hartmann | maseberg }@cdc.informatik.tu-darmstadt.de

²GMD-SIT Darmstadt
maseberg@darmstadt.gmd.de

Public key cryptography is based on mathematical problems, which are supposed to be hard to solve, but this is not known for sure. That is the reason, why we already presented a public key infrastructure, which will keep its functionality in the case of failure and which is able to be repaired and where generated digital signatures won't lose their proofability. In this article we will focus on the smart card, which is used in this environment. We define the special requirements and present our proposal to fulfill these specific needs.

The Challenge

The security of a Public Key Infrastructure (PKI) depends on the quality of the used components, that means the cryptographic components like signature algorithms and hash functions and the application specific components like the keys and used parameters. The German “Bundesamt für Sicherheit in der Informationstechnik (BSI)” provides an overview of algorithms and parameters which are qualified for generating signature keys, for hashing of data to be signed or for generation and verification of digital signatures ([BSI]). For example the hash functions SHA-1 and RIPEMD-160 and the signature algorithms RSA with 1024 bit, DSA with 1024 bit and ECDSA with 160 bit key length are qualified for cryptographic purposes, but only recommended until 2004. It could not be excluded that one component is compromised, since no proofable secure signature algorithms and hash functions exist, but there are keys which are not suitable to be used in cryptography. If a failure occurs the security of the PKI could not be guaranteed any longer as a whole, this means the electronic communication may be insecure. An existing mechanism for such a case is the revocation of certificates. The velocity, the revocation can be recognized with, depends on the security policy, which tells if the certificates must be checked only when verifying signatures or additionally while generating a new signature. The security policy must define if Certificate Revocation Lists (CRLs) [HFPS00] are used,

at which time the CRLs have to be updated or if the status of a certificate must be checked via Online Certificate Status Protocol (OCSP) [AAM+99]. Two problems still exist:

The revocation of a certificate will reduce the functionality of the PKI. The reconstruction of the PKI may cause the development, production, distribution and installation of new components, depending on the failure. This efforts time and money.

The argumentative force of a digital signature generated before the failure occurred could be lost if the signature algorithm or the key is compromised. Therefore our aim is to develop a PKI which makes it possible to exchange the compromised components in a secure way and where digital signatures are extended in a way that they do not lose their argumentative force, if a failure occurs. As a basic restriction the technical components must be integrated in the PKI in such a flexible manner, that an exchange could be possible. The flexible public key infrastructure which is developed at the chair of professor Buchmann [BuRT00] in the “FlexiPKI” project is following this philosophy.

The Idea – previous work

The main idea is, to build up a public key infrastructure upon several independent cryptographic and application specific components, where in the case, one component is compromised the other components still can be used to: Exchange the compromised component securely and sign electronic documents multiple (concept of multiple digital signatures). This idea is based on the proposal that the possibility, that the occurrence of two failures at the same time is very small. Since nobody can proof things concerning the future, we have to make the following assumptions and restrictions: It is assumed, that it is impossible, neither to solve all mathematical base problems, to develop new efficient solutions, nor to build computers with such a high performance, that all cryptographic algorithms are suddenly broken without any warning. If a failure occurs, it should be handled securely to avoid any damage – a so called fail safe concept.

The concepts of multiple signatures and exchangeable cryptographic components are explained in detail with cost/profit considerations in [HaMa01]. In this article we focus on how the exchange of cryptographic components is done on the smart cards, which are used in the FlexiPKI environment.

Concept of a Smart Card for FlexiPKI

Now we will describe the requirements of a smart card for this environment, how the exchange is done and which standards should be modified.

Requirements

Our aim is to define and provide a smart card which can be used in the FlexiPKI environment. This smart card has to support at least two sets of cryptographic components, where a set of components denotes a signature algorithm, a hash algorithm and a padding scheme. These components must be replaceable after rollout of the smart cards and this smart card should never disclose any secret information.

The Registry

The smart card has to store and to manage some information to handle the components on the card. This unit is called the Registry. The information managed in the Registry is listed in the tables 1 – 5. ICC-ID is the unique identification number of the smart card. This is necessary to send a message to a specific card, e.g. when only this CH's private key is broken. To avoid overflows when incrementing the sequence counters we introduce two sequence counters. They are called GlobalMessageID and PersonalMessageID. The PersonalMessageID sequence counter is used and incremented if a message is relevant only for this smart card. In all other cases the GlobalMessageID is used. The table Registry.Methods (ref. table 2) maps an AlgoID on the AID which implements this algorithm. The field Security Conditions defines which security conditions have to be fulfilled to remove an algorithm from the card. In detail there are stored several AlgoIDs and Key-IDs which must be found in Registry.SecurityConditions when the algorithm should be removed. Table Registry.Keys (ref. table 3) assigns to a specific key the owner (CA or CH), the related security anchor (reference to the public key of the certification authority), the algorithm this key can be used with and it defines the security conditions which must be fulfilled to remove a key from the card. In the list Registry.Broken (ref. table 4) the AlgoIDs of already broken and deleted algorithm are listed, to prevent from installing them a second time. The list Registry.SecurityConditions is cleared with every received UPDATE_COMPONENT_DATA. During the update procedure this list is filled with the AlgoIDs and Key-IDs of correctly verified signatures (ref. figure 1).

Tab.1: Registry

Name	Length	Description
ICC-ID	Variable	Unique Identification Number
GlobalMessageID	4	Sequence counter
PersonalMessageID	4	Sequence counter
Methods	Variable	Table that assigns AlgoID and AID
Keys	Variable	Table that stores key information
Security Conditions	variable	List that stores AlgoIDs and Key-IDs
Broken	variable	List that stores broken AlgoIDs

Tab.2: Registry – Methods

AlgoID	AID	Description	Security Conditions
...

Tab.3: Registry – Keys

Owner	CH Keys Key-ID	Related Security Anchor	AlgoID	Security Conditions
{CA or CH}	{Key-ID}	{Key-ID of CA's Public Key}	{Algorithm use with}	{sequence of AlgoIDs and Key-IDs}

Tab.4: Registry – Broken

AlgoID / Key-ID
...

Tab.5: Registry – Security Conditions

AlgoID / Key-ID
...

How the Exchange works

The user receives an object of the datatype UPDATE_COMPONENT_DATA (UCD - ref. table 1) which he sends to the smart card (in the following called client, since this is generic) via a sequence of APDUs. The client deletes the compromised components and installs the new components, that are included in the UCD object. The TLV objects that could be part of the UCD object are listed in table 6.

The client receives the UCD object and then it starts to process the enclosed data, as illustrated in figure 1. First the list Registry.SecurityConditions is cleared (1). If UCD.ICC-ID is present (2) the client has to check if it equals its own Registry.ICC-ID (3). If they do not match the process will be aborted immediately (4). Now the client has to validate the sequence counter UCD.MessageID. UCD.MessageID is compared to Registry.PersonalMessageID (5) if UCD.ICC-ID is present and to Registry.GlobalMessageID (6), if UCD.ICC-ID is not present. If UCD.MessageID is not higher than the corresponding value in the Registry, the process will be aborted (7). Now all existing UCD.Signature objects are verified and for each valid signature the used AlgoIDs and Key-IDs are stored in Registry.SecurityConditions (8). Then the number of valid UCD.Signature objects is determined (9) and if less than 2 valid signatures are found, the process also will be aborted (10). In the other case an atomic transaction begins (11): the internal sequence counter Registry.GlobalMessageID or Registry.PersonalMessageID is incremented by one, depending on the presence of

UCD.ICC-ID (12/13). All algorithms stored in UCD.ToInstall are installed now, if they are not listed in Registry.Broken (14). In Registry.Methods the AIDs of the installed components are assigned to the AlgoIDs and their Security Conditions are set (15). The algorithms listed in UCD.ToDelete are removed from the card, the corresponding entry in Registry.Methods is deleted and the AlgoIDs of the deleted algorithms are appended to Registry.Broken (16). Now the transaction is committed (17), if no error occurred, or aborted (18).

Tab.6: Known TLV objects in datatype UPDATE_COMPONENT_DATA

Tag	Length	Value	Description
'01'	variable	ICC-ID	Unique ID of ICC
'02'	4	MessageID	Sequence Counter
'03'	variable	To Delete	Constructed – which component should be deleted – contains a sequence of AlgoIDs
'04'	variable	To Install	Constructed – which component should be installed – contains a sequence of Algorithm
'05'	variable	Algorithm	Constructed – contains a pair of AlgoID and Binary
'06'	variable	AlgoID	ID of the concerned Algorithm
'07'	variable	Binary	Code that should be installed
'08'	variable	CA-Key: KeyID	
'09'	variable	CA-Key: Key	The Key binary coded
'0A'	variable	CH-Key: KeyID	
'0B'	variable	Signature	Constructed, contains Signature Info and Signature Binary
'0C'	variable	Signature Info	
'0D'	variable	Signature Binary	
'0E'	variable	CV certificate	If security anchor is not associated to this signature

After the new component is installed, eventually new keys must be generated and new certificates have to be created. This depends on the exchanged component. After installation of a new hash function for example, there is no need to generate new keys in the opposite of the exchange of a new signature algorithm. The smart card generates new keys for the CH and it must be guaranteed that the new public key is transferred authentic to the CA, that the CA can generate the corresponding certificate.

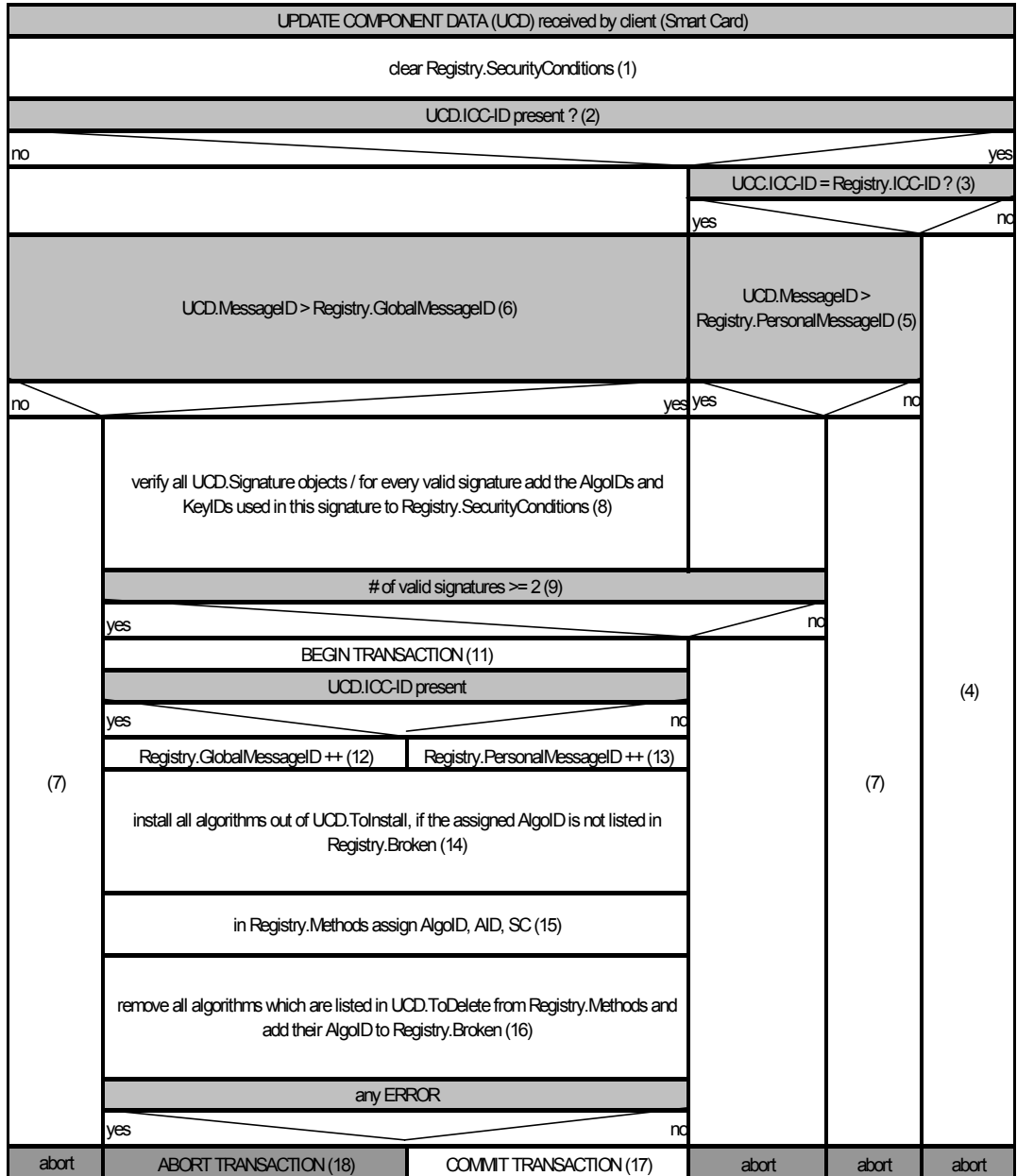


Fig.1: Process in the card after receiving UCD

Modification of / Integration in existing Standards

Since only the Open Platform (OP) standard [OP21] describes secure download mechanisms for smart cards, we decided to modify this standard for our special needs. We will show the modifications that are necessary to introduce algorithm identifiers and additional cryptographic algorithms for data authentication pattern (DAP) generation and how these should be handled in the smart card. OP only supports Triple DES and RSA with 1024 bit key length therefore we have to introduce algorithm identifiers and additional cryptoalgorithms for data authentication pattern generation. Our proposal also implies modifications of the Card Manager, Issuer Security Domain and Security Domain implementations to realize the modified functionality. The Card Manager has to implement Delegated Management, the extended Registry and (a bit more complicated) the Java Card Runtime Environment has to be extended, that it is possible to exchange cryptographic components within the Java Card API, transparent for the applets that reside on the card. The (Issuer) Security Domain needs the ability to verify the new DAP and to process the UCD object. How this implementation is done is out of the scope of this article and will be considered in further publications.

In OP the necessary mechanisms are already available and only have to be changed in some details. The internal structure of the Load File has to be changed (ref. table 9). We simply added the UPDATE COMPONENT DATA to the Load File. The UCD includes all the necessary information for the security domain to verify multiple signatures of the Load File Data Block. Since with multiple signatures we also need multiple hash functions, we have to modify the INSTALL for LOAD Command Data Field (ref. table 8). To this field we added a second Load File Data Block Hash. Both Load File Data Block Hashes are hash values over a concatenation of UCD and the Load File Data Block. The Load Token got a new structure (ref. table 7) and therefore we had to extend the Length of Load Token value in the INSTALL for LOAD Command Data Field. The answer to the INSTALL for LOAD command is a LOAD RECEIPT generated by the ISSUER Security Domain. This LOAD RECEIPT also has to include two signatures. The must be generated using valid algorithms that are listed in Registry.Methods. The LOAD TOKEN and LOAD RECEIPT calculation is illustrated in table 13 and 14.

After the INSTALL for LOAD command the modified Load File could be loaded into the card with known mechanisms from OP.

To install the new components we have to execute an INSTALL for INSTALL command. The data field of this command is also modified (ref. table 10) and includes a new Install Token (ref. table 7, 10, 12).

Within the INSTALL command the Security Domain processes the UCD object and the Issuer Security Domain generates the modified Install Receipt (ref. table 7, 11)

Tab.7: Registry – Additional Tags

Tag	Length	Value	Description
' 10'	variable	Load Token	Constructed - contains 2 Signature objects
' 11'	variable	Install Token	Constructed - contains 2 Signature objects
' 12'	variable	Load Receipt	Constructed - contains 2 Signature objects
' 13'	variable	Install Receipt	Constructed - contains 2 Signature objects
' 14'	variable	Hash	Constructed - contains 1 AlgoID object and 1 Binary object

Tab.8: Modified INSTALL for LOAD Command Data Field

Presence	Length	Name
Mandatory	1	Length of Load File AID
Mandatory	5-16	Load File AID
Mandatory	1	Length of Security Domain AID
Conditional	5-16	Security Domain AID
Mandatory	1	Length of Load File Data Block Hash 1
Mandatory	1-n	Load File Data Block Hash 1
Mandatory	1	Length of Load File Data Block Hash 2
Mandatory	1-n	Load File Data Block Hash 2
Mandatory	1	Length of Load Parameter Field
Conditional	0-n	Load Parameter Field
Mandatory	2	Length of Load Token
Mandatory	1-n	Load Token

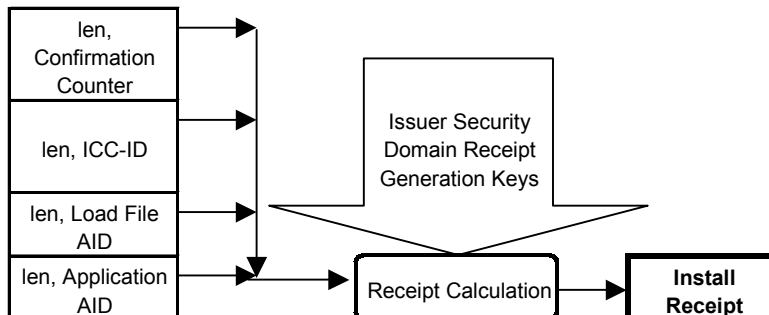
Tab.9: Modified Load File Structure

Tag	Length	Value
'E2'	Variable	DAP Block
'4F'	5-16	Security Domain AID
'C3'	Variable	Load File Data Block Signature
...
...
'E2'	Variable	DAP Block
'4F'	5-16	Security Domain AID
'C3'	Variable	Load File Data Block Signature
XY	Variable	Update Component Data
'C4'	Variable	Load File Data Block

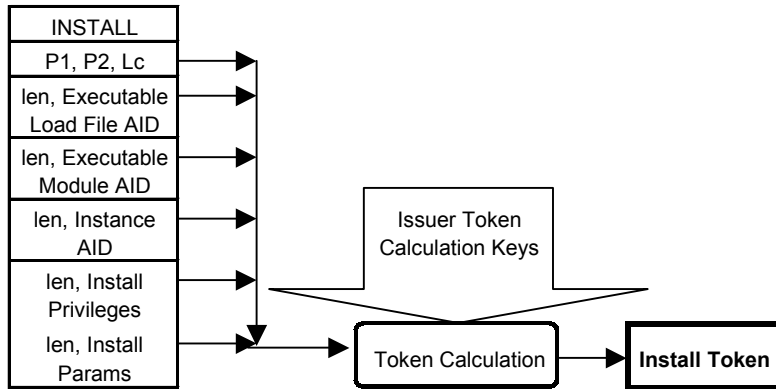
Tab.10: Modified INSTALL for Install Command Data Field

Presence	Length	Name
Mandatory	1	Length of Executable Load File AID
Mandatory	5-16	Load File AID
Mandatory	1	Length of Executable Module AID
Conditional	5-16	Executable Module AID
Mandatory	1	Length of Application AID
Mandatory	1-n	Application AID
Mandatory	1	Length of Application Privileges
Mandatory	1	Application Privileges
Mandatory	1	Length of Install Parameter Field
Conditional	0-n	Install Parameter Field
Mandatory	2	Length of Install Token
Mandatory	1-n	Install Token

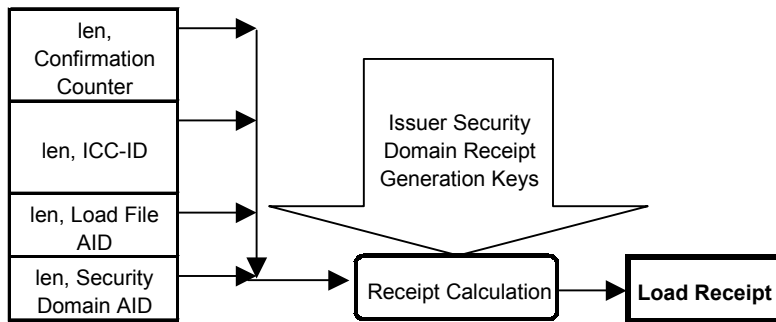
Tab.11: Modified Install Receipt Calculation



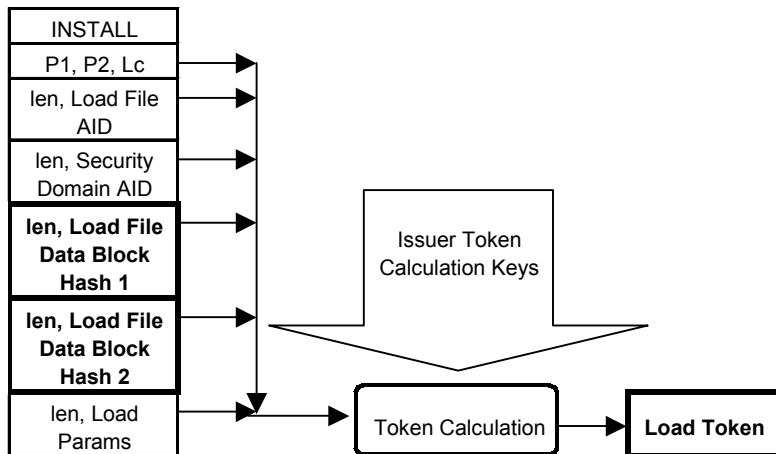
Tab.12: Modified Install Token Calculation



Tab.13: Modified Load Receipt Calculation



Tab.14: Modified Load Token Calculation



Proof of Concept

Unfortunately we do not have the resources to build up our own operating system. Therefore we would like to realize our ideas by modifying the operating systems of existing java cards (for example GemXpresso). Currently we are mapping our ideas onto GemXpresso 211 Java Card, regarding the existing OP 2.01 Standard [OP201]. Our proposal will use Security Domains to represent the application provider on the card. The exchangeable cryptographic components will be implemented in a JCA like provider concept. This concept will be realized with applets communicating via the shareable interface. A known disadvantage will be, that the security anchor is weak because currently OP only supports Triple DES and RSA with 1024 bit key length. Additionally to RSA we implemented Elliptic Curve DSA on this card and are working on more signature algorithms.

References

- [AAM+99] C. Adams, R. Ankney, A. Malpani, M. Myers, S. Galperin: Internet X.509 Public Key Infrastructure - Online Certificate Status Protocol - OCSP. Request for Comments 2560, 1999.
- [BSI] Bundesamt für Sicherheit in der Informationstechnik. <http://www.bsi.de>
- [BuRT00] J. Buchmann, M. Ruppert, M. Tak: FlexiPKI - Realisierung einer flexiblen Public-Key Infrastruktur. In: P. Horster: Systemsicherheit, Vieweg, 2000.
- [HaMa01] M. Hartmann, S. Maseberg: Fail-Safe-Konzept für FlexiPKI. In: P. Horster: Kommunikationssicherheit – Im Zeichen des Internet, Vieweg, 2001.
- [HFPS00] R. Housley, W. Ford, W. Polk, D. Solo: Internet X.509 Public Key Infrastructure - Certificate and CRL Profile, Internet Draft, 2000.
- [OP21] Open Platform Card Specification Final Draft 2.1, April 16th 2001.
- [OP201] Open Platform Card Specification Version 2.01, April 7th 2000.