Credential Management and Secure Single Login for SPKM

Detlef Hühnlein
secunet Security Networks GmbH
Mergenthaler Allee 79-81
D-65760 Eschborn
email huehnlein@secunet.de

Abstract

The GSS-API [20, 21] offers security services independent of underlying mechanisms. A possible GSS-mechanism is the Simple Public Key Mechanism (SPKM) specified in [1]. In this paper we will focus on the credential management for SPKM. If more than one connection is needed, the standard credential management requires either to cache the secret keys in insecure storage or to make the user entering a password to access the long term secret keys for every new GSS-connection. For environments in which neither one is acceptable we propose a Secure Single Login (SSLogin) variant which works with temporary asymmetric keys and combines security and user comfort.

1 Introduction

The GSS-API [20, 21] "offers security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing source-level portabiltiy of applications to different environments". Possible GSS mechanisms are e.g. the well known Kerberos V5 [18, 22] based on symmetric cryptography, the Simple Public Key Mechanism [1] or SECUDE [7, 9]. The latter may be viewed as an SPKM variant tailormade for the SAP R/3 ™ environment.

To set up a secure (i.e. *authentic*, *integer*, *confidential* and *non-repudiable*) communication the initiator (client) and the target (server) have to establish a *GSS-context*, like discussed in section 2 more detailed. Note, that non-repudiation requires the application of digital signatures, i.e. public key algorithms. During context establishment the communication partners verify the peer's identity and authorization and agree on a common session key, which can be used for confidentiality and integrity purposes during the actual communication. To proof the identity one has to acquire *credentials*. In Kerberos these credentials are so called tickets with *limited lifetime*. In public-key based mechanisms like SPKM, which is discussed in section 3 more detailed, and SECUDE the credentials are the secret keys and certificates for the public keys. The secret keys are stored in a *personal security environment* (PSE), which is 'opened', i.e. made accessible, by entering a password. In practice the PSE usually is a smartcard or a PKCS#5 (see [31]) encrypted file. This credentials have to be available whenever a new GSS-connection is requested, i.e. a GSS-context is to be established. This means, that either the PSE has to be open for a long time or the user has to enter the password everytime a new connection is set up. If this no problem, this is the most obvious way to provide access to the secret keys and, while not specified in [1], X.509 v3 / PKIX certification for the related public keys, which is briefly discussed in section 4.

However if keeping the PSE open for a long time bears security problems or multiple entering of the password is not possible for usability reasons the proposed SSLogin variant, as discussed in section 5, is preferable. To implement this SSLogin we need to specify a slightly different credential management (see section 6), which allows the end user to certify its own *temporary keys*. In section 7 we will compare the discussed variants in terms of security, usabilty and performance.

2 GSS-API Context Establishment

In this section we will briefly recall the relevant topics of the GSS-API. Thus we focus on the calls and tokens for context establishment and credential management, rather than per-message-, and support calls. For a comprehensive treatment we refer to [20, 21, 34].

Like mentioned in the introduction the communication partners have to aquire credentials, before they establish a GSS-context. This is done by calling

GSS_Acquire_cred

In:

(

desired_name	INTERNAL NAME
lifetime_req	INTEGER
desired_mechs	SET OF OBJECT IDENTIFIER
...	

)

Out:

(

output_cred_handle	CREDENTIAL HANDLE
...	

)

In our case, i.e. the client wants to establish an SPKM-GSS-context, *desired_name* is the clients (user-) name, the *lifetime_req* is in seconds , e.g. 86400 for one day and *desired_mechs* is $\{1\ 3\ 6\ 1\ 5\ 5\ 1\ variant\}$, where $variant$ equals 1 (for SPKM with random numbers) or 2 (for SPKM with secure timestamps), like specified in [1].

Besides GSS_Acquire_cred there are functions to destroy the credentials after use (GSS_Release_cred), construct credentials incrementally (GSS_Add_cred) and gather information about credentials (GSS_Inquire_cred, GSS_Inquire_cred_by_mech).

As soon as the credential is available the client may call

GSS_Init_sec_context

In:

(

claimant_cred_handle	CREDENTIAL HANDLE
input_context_handle	CONTEXT HANDLE
targ_name	INTERNAL NAME
mech_type	OBJECT IDENTIFIER
mutual_req_flag	BOOLEAN
lifetime_req	INTEGER
input_token	OCTET STRING
...	

)

Out:

(

major_status	INTEGER
output_context_handle	CONTEXT HANDLE
output_token	OCTET STRING
mutual_state	BOOLEAN
lifetime_rec	INTEGER
...	

)

Here *claimant_cred_handle* is the handle to the credential obtained by GSS_Acquire_cred. Since we are just at the beginning of context establishment, there is no context and thus *input_context_handle* is NULL. The *targ_name* is the name of the server to connect to and the *mech_type* is like above. If mutual authentication is required, which is recommended, the *mutual_req_flag* is TRUE. Finally the desired context lifetime, i.e. *lifetime_req* and the *input_token* is handed over. Since there is no preceeding token *input_token* is NULL.

The output of this call consists of the *major_status*, a handle to the context to be established, the *output_token* to be passed to the server, the *mutual_state* flag and the *lifetime_rec*, which indicates the 'time to live' in seconds. Since we requested mutual authentication the *mutual_state*-flag is set and the *major_status* is GSS_S_CONTINUE_NEEDED. The client knows, that the *output_token* has to be passed to the server.

Note, that the presented interface description is not at all complete. We confine ourselves to a rather small subset which is needed to explain how a typical mutual authentication works.

After the server also got his credential by calling GSS_Acquire_cred he can use this handle, NULL as *input_context_handle* and the received token to call

GSS_Accept_sec_context

In:

(

acceptor_cred_handle	CREDENTIAL HANDLE
input_context_handle	CONTEXT HANDLE
input_token	OCTET STRING
...	

)

Out:

(

major_status	INTEGER
src_name	INTERNAL NAME
output_context_handle	CONTEXT HANDLE
mutual_state	BOOLEAN
lifetime_rec	INTEGER
output_token	OCTET STRING
...	

)

Since the *mutual_state*-flag is set, the *output_token* has to be passed back to the client. For the server however the contex establishment is finished, because *major_status* is GSS_S_COMPLETE. The client again calls GSS_Init_sec_context with the received token. Now the *major_status* is GSS_S_COMPLETE. The context establishment is finished, both peers access the parameters and keys of the negotiated security context via their *context_handle*.

If something went wrong during this context establishment, if the 'time to live' is up or the context will not be needed anymore it may be deleted with GSS_Delete_sec_context.

3 Simple Public Key Mechanism

In this section we will briefly recall the most basic facts about SPKM. Like in the previous section we will focus on the context establishment rather than the per-message calls. The credential management will be discussed in section 4. For a comprehensive treatment we refer to [1].

SPKM permits the negotiation of the algorithms to be used for integrity (I-algs) and confidentiality (C-algs) purposes, oneway functions for subkey derivation (O-algs) and (K-algs) for key establishment. Note, that the initial negotiation in SPKM might be enhanced by concepts presented in [3]. The I-alg md5WithRSAEncryption and the K-alg RSAEncryption defined in [29] are specified *mandatory*, while other algorithms are *recommended* or *optional*. In SPKM there are two different variants specified. SPKM-1 uses random numbers for replay detection during authentication and SPKM-2 requests the presence of secure timestamps. Since these secure timestamps might not be available in some environments and mutual authentication is recommended, we briefly recall the well known Three-Way-Authentication specified in [13] or [11] section 10.4.

The initiator (client) wants to authenticate to the target (server) and vice versa. The client C and the server S are in possesion of their own credentials, i.e. the secret keys for encryption Se_C/Se_S and signatures Ss_C/Ss_S and the corresponding certificates $Cert_{Pe_C}, Cert_{Ps_C}$ and $Cert_{Pe_S}, Cert_{Ps_S}$ containing the respective public keys. The credential management is discussed in the following sections more detailed. For authentication we need an encryption- $Cip = ENC(M, Pe)$ and decryption algorithm $M = DEC(Cip, Se)$, e.g. RSAEncryption and a signature- $Sig = SIG(h(M), Ss)$ and verification algorithm $h(M) = VER(Sig, Ps)$, like md5WithRSAEncryption. Furthermore R_C and R_S are random numbers generated by the client and server respectively. The symbol $|$ denotes concatenation, e.g. $R_C|S$ is the clients random number concatenated with the servers identity. The authentication procedure typically works like this:

(1)	C	generates R_C, $M = R_C\|S\|C$ and signs $F = SIG(h(M), Ss_C)$
(2)	$C \rightarrow S:$	$M, F, Cert_{Pe_C}, Cert_{Ps_C}$
(3)	S	verifies $Cert_{Pe_C}, Cert_{Ps_C}$, checks if the identities S and C are ok and verifies $h(M) = VER(F, Ps_C)$
(4)	S	generates R_S, a random session-key K_{CS}, $N = C\|R_S\|R_C\|K_{CS}$ and computes $G = ENC(N, Pe_C)$ and $H = SIG(h(N), Ss_S)$
(5)	$S \rightarrow C:$	$G, H, Cert_{Pe_S}, Cert_{Ps_S}$
(6)	C	verifies $Cert_{Pe_S}, Cert_{Ps_S}$,decrypts G, checks C and R_C, stores the session key K_{CS}, verifies the signature H and computes $I = ENC(R_S, Pe_S)$
(7)	$C \rightarrow S:$	I
(8)	S	finally decrypts $R'_S = DEC(I, Se_S)$ and checks, whether $R_S = R'_S$

Note, that if the client knows the certificate for the server's encryption key $Cert_{Pe_S}$ prior to (1), he can instead of sending the plain message M in (2) send $E = ENC(M, Pe_S)$ for even more security. Of course the server has to decrypt E in (3), i.e. $M' = R_C|S = DEC(E, Se_S)$. Furthermore [1] covers the possibility that the initiator remains anonymous. In this case, the client does not send his identity C in step (1) and the tokens are MACed instead of signed. As before, the content of the exchanged tokens presented here is not complete. For a comprehensive treatment we refer to [1].

4 Standard Credential Management - X.509

In this section we will focus on the credential management for SPKM. In [1] there is not very much said about this problem.

> "The key management employed in SPKM is intended to be as compatible as possible with both X.509 [11] and PEM [15], since these represent large communities of interest and show relative maturity in standards."

In this section we will treat this topic with more scrutiny. Before we will discuss the public key management more detailed, we will briefly recall how the secret keys are handled.

Since the *secret signature key* represents the *owners identity* for a long time, possibly many years, it has to be protected very carefully. This secret key is stored right after key generation in a secure environment (PSE), usually on a tamperproof smartcard. Ideally the key generation itself is performed inside the card, which is not possible with every card, because key generation, especially for RSA keys, is a very time consuming operation. If no chipcard is available to house the secret key it may be symmetrically encrypted

with a key derived from the entered password, like specified in PKCS#5 [31]. In any case the secret keys and other sensitive information like the trusted public root key and in some cases the latest Certificate Revocation List, or its hash-value should only be accessible after entering the password. The time available for this access, before the PSE will be closed again, should be as short as possible, that no attacker will have the opportunity to obtain the secret key easier than inverting the oneway function. If, for usability reasons, it is not possible to make the user entering the password everytime the secret key is needed to establish a new GSS-context we propose to use the SSLogin mechanism discussed in section 5.

Before the keys can be used the public key has to be certified, by a trusted Certification Authority (CA). The certification request may be proceeded using PKCS#10 [32]. A certification request consists of the user's distinguished name, the public key, and optionally a set of attributes, collectively signed with the corresponding secret key. This certification request is sent to a CA, which verifies the signature and transforms the request into an X.509 public key certificate which is handed back to the user and may be stored in a public X.500 Directory. This Directory contains certificates and certificate revocation lists (CRL) and may for example be accessed via the Lightweight Directory Access Protocol (LDAP) [35].

The certificate format X.509 v1 [11] and the related public key infrastructures [15] showed out to be deficient and too restrictive for broad application. Especially the rigid hierarchical model with Internet Policy Registration Authority (IPRA), Policy Certification Authorities (PCA) and CAs and the name subordination rule proposed in RFC1422, as well as the missing possibility to extend the certificates by application specific information turned out to be unsuitable for a lot of environments. Therefore ISO/IEC defined the X.509 v3 format [12], which overcomes this problems by providing *certificate extensions* to the X.509 v1 format. Note, that the v2 format equals the v1 format extended by some specifications to make the Directory access easier. The extensions relating to *certificate policies* obviate the need for PCAs and the *constraint extensions* make the name subordination rule unnecessary. An extension field consists of an extension identifier, a criticality flag and a DER-encoding of a data value of an ASN.1 type associated with the identified extension. Some extensions specified in [12] have to be *non-critical*. In all other cases, e.g. the five discussed below, the criticality is *at the option of the certificate issuer*. In X.509v3 no certificate extensions are forced to be critical. "If an extension is flagged critical and a certificate-using system does not recognize the extension field type or does not implement the semantics of the extension, then that system shall consider the certificate invalid." [12]

Since X.509 v3 is designed to achieve maximum flexibility, it was necessary to complement [12] with a more concrete profile tailormade for internet application [8]. We will see, that this PKIX profile, which is still in draft status, *almost* fits our needs. Because a comprehensive treatment would be beyond the scope of this work, we will only present the extensions in [8] which are affected by our proposal.

- *Subject Alternative Name*
  This extension provides the possibility to associate additional identities with the *subject* referenced in the subject distinguished name. If this extension is present and the subject distinguished name is empty this extension should be critical. Possible name types are
  - rfc822Name (email adress)
  - dNSName
  - X400Address
  - directoryName
  - ediPartyName
  - uniformResourceIdentifier
    or any
  - otherName
- *Issuer Alternative Name*
  This extension allows to associate additional identities with the *issuer* of a certificate. The designated name types and the criticality flags are like above.
- *Basic constraints*
  The basic constraints extension allows to distinguish between an end user- and a CA-certificate. There is a boolean flag 'cA' which is set to FALSE by default. If this cA-flag is FALSE, the certificate belongs to an end user. Since this extension is recommended to be *critical* an end user cannot act as a CA without notice. If an end user, i.e. cA=FALSE, signs a certificate it will not be valid. Optional the length of the verification path may be limited to 'pathLenConstraint'.
- *Key usage*
  This extension allows to restrict the usage of the key contained in the certificate and is recommended to be critical. The information is represented in a BitString, where the bits are as follows:

  (0) digitalSignature
  (1) nonRepudiation

(2) keyEncipherment
(3) dataEncipherment
(4) keyAgreement
(5) keyCertSign
(6) cRLSign
(7) encipherOnly
(8) decipherOnly

If only one user key pair is available, (0) through (4) are set, while the CA's certificate has (5) and (6) set. If the user has more than one key pair, (0) and (1) is set for the signature key and the bits (2) through (4) are set for the encryption key. If (4) is set, it is possible to restrict the usage of the key to enciphering (7) or deciphering (8). In any case a user key would *never* have (5) or (6) set. Thus it is impossible for a user to act as a CA without notice.

- *Extended key usage*
  Here it is possible to restrict the key usage even further by explicitly specifying *Key Purposes*. This extensions may be used in addition to or in place of the basic purposes indicated in the *Key usage* extension field. This extension field was not present in the X.509 recommendations [12] and the PKIX profile [8] until recently. Experiences in deploying the base standard showed, that there are situations, in which it is necessary to specify the key usage somewhat more concrete. In the next section we will see, that the conflicts between our proposed credential management (in an earlier version of this paper) and the base X.509 standard may now be elegantly resolved by introducing a tailormade *Extended key usage* field.

The *name extensions* are thought to improve the name mapping for different environments, like Email, X.400 mail or EDI for example.

The *basic constraints* extension is very important, since it enables the distinction between end user- and CA-certificates. It obviates the need for the name subordination rule in X.509 v1. Therefore it has to be critical, i.e. the certificate verification will fail, if this extension is not recognized or if the certificate is certified (signed) by an end user. While the conservative evaluation of this extension is very important and totally sensible in general, we need to evaluate this extension *a little more relaxed* to implement our proposed SSLogin, like discussed in the next sections.

5 Secure Single Login

In SPKM the secret keys have to be available for every context establishment, like pointed out in section 3 and 4. Since a client usually requests multiple GSS-connections to different servers at different times, the PSE has to be kept open for a long time or the user has to enter the PIN for every new connection. To overcome this problem we propose to use the SSLogin variant, which combines security and user comfort. It works with *temporary asymmetric keys* instead of the more valuable long term keys. The philosophy of SSLogin is equal to Kerberos' [18], where we use temporary keys (tickets) with limited lifetime to authenticate. If an attacker manages to access the memory or the harddrive he will only get the temporary secret keys instead of the valuable long term keys. If this disclosure is recognized the temporary certificate may be revoked. Note, that the Distributed Authentication Security Service (DASS) [14] also proposes the use of "self-certificates". However the main purpose of self-certification there is to implement a "smooth key rollover", if the keys have to be changed.

We propose the following operations for Secure Single Login:

1. Entering the PIN and *opening the PSE* to access the long term secret keys
2. *Generation of temporary key(s)*
3. *Self-Certification* of the temporary public key(s) with the long term signature key
4. *Closing the PSE*

Note, that in some environments a further benefit of SSLogin might be that it only requires a certified *signature* key. However, if two key pairs are available, the security can be further enhanced:
The first GSS-context could be established as usual with the long term keys and the temporary secret key(s) generated in 2. could be PKCS#5 encrypted with the symmetric session key negotiated in the first context establishment. In this case we have to set up a handle to this special context and take care, if this context is to be deleted.

Furthermore it is possible to parallelize this two steps. I.e. the temporary key(s) may be generated, while one is waiting for the peer's response.

If we use temporary RSA-keys we only have to generate one key pair which can be used for signatures and encryption. However it is well known, that the generation of an RSA key pair is a rather time consuming task. Two large (strong) primes p and q have to be found. The public modulus $n = pq$ and the public exponent e has to be computed by inverting the random secret key d modulo $\varphi(n) = (p-1)(q-1)$. The bottleneck in this procedure is to find the two primes. Since it is not possible to use multiple pairs (e_1, d_1), ... (e_i, d_i) with the same modulus n, because of the common modulus attack [33, 5],

we have to perform the tedious search for good primes everytime we generate a temporary key pair. Therefore we recommend to use *discrete logarithm based systems*, like e.g. ElGamal-encryption [6] and DSA-signatures [25] for SSLogin-SPKM. In this case we only have to search for a suitable prime once. The actual key generation for a temporary key, which is typically performed every day, consists of a simple exponentiation of the generating element g with the random secret key. That is why we propose to extend the list of *recommended* algorithms in [1] by the two following algorithms defined in [27]

1. elGamal ALGORITHM
   PARAMETER NULL ::= {1 3 14 7 2 1 }
2. dsaWithSHA1 ALGORITHM
   PARAMETER DSAParameters ::= { 1 3 14 7 3 27 }

Note, that a valid alternative might be the application of the DSA-variant proposed by Nyberg and Rüppel [26], which features message-recovery. The most time consuming part in these algorithms is the modular exponentiation. Therefore the time required for context establishment may be significantly reduced by applying some *precomputation variant*, like [4, 19] for exponentiation of the generating element g. Note, that these precomputation variants are *not* applicable to RSA-type cryptosystems.

Since there is no known weakness in using the same prime p for ElGamal encryption and DSA signatures, we may save some time and memory by doing so. In this case only the parameters g_e for ElGamal encryption and g_s for DSA signatures are different, because g_e usually is of order $p-1$, while the order of g_s is q, a 160 bit prime factor of $p-1$. Also, one might think of using the analogous algorithms based on *elliptic curves*, like proposed in [24, 17] and currently standardized in [10].

Note, that the *self* - certification in step 3. violates the PKIX-profile [8], because an end user is not allowed to sign certificates. In the proposed SSLogin-SPKM however it is essential, that this self-certification is allowed. It would take too long for a user to contact a trusted certification authority to get the temporary public key(s) certified. Since the conservative evaluation of the basic constraints and the key-usage extension is not necessary in this context, we can change the proposed SPKM-profile accordingly, like discussed in the next section.

6 Credential Management for SPKM

In section 4 we briefly discussed the standard credential management, i.e. the secret key handling, the X.509 v3 certificates [12] and the more concrete PKIX profile [8] for internet application. In this section we will give concrete recommendations for the credential management to be applied in SPKM. We propose to use the PKIX profile with the following incremental changes to allow the self-certification of the temporary public key(s).

We define two new key purposes for the *Extended key usage* field:

- **id-kp-SignTempCert**
  OBJECT IDENTIFIER ::= {**id-kp** 1}
  If this key purpose is present it is allowed to sign a certificate for a temporary key, even if the cA-flag in the Basic constraints extension is set to FALSE.
  The Key usage bit (0) 'digitalSignature' has to be set and (1) 'nonRepudiation' may be set.
- **id-kp-Temporary**
  OBJECT IDENTIFIER ::= {**id-kp** 2}
  This key purpose indicates, that it is a temporary key and that the next certificate in the verification chain may be a user certificate in which **id-kp-SignTempCert** has to be present.

Since the proposed SSLogin mechanism is tailormade for SPKM we recommend to subordinate these two object identifiers to SPKM. Thus **id-kp** OBJECT IDENTIFIER::= {**id-spkm** 3} and **id-spkm** OBJECT IDENTIFIER::={1 3 6 1 5 5 1}.

To implement the proposed SSLogin we will allow a slightly more relaxed certificate verification for this temporary certificates, like discussed in the sequel.

We will assume, that a user already got X.509 v3 / PKIX certified public keys. We will only need the signature key, where 'id-kp-SignTempCert' is present. The relevant fields of a certificate containing the users public signature key Ps_U informally look like this:

User-Certificate-$CertPs_U$:

- issuer=CA
- validity=
  - U-notBefore
  - U-notAfter
- subject=User
- subjectAltName=UserAlt
- issuerAltName=CAAlt
- Key usage

  - critical=TRUE
  - digitalSignature=TRUE
  - nonRepudiation=TRUE
  - all others=FALSE
- Extended key usage
  - critical=FALSE
  - id-kp-SignTempCert
- Basic constraints
  - critical=TRUE
  - cA=FALSE

This certificate is issued by a trusted CA. I.e. it is signed with the CA's private key. In the certificate for the corresponding CA-public key, the Key Usage indicates 'keyCertSign' and 'cRLSign'. The cA-Flag in the Basic constraints extension is TRUE.

Now the User will be allowed to generate and self-certify temporary public keys, like proposed in section 5 2. and 3. respectively. Note, that the Extended key usage extension above is flagged non-critical, because the long term signature key may be used for other purposes as well. Thus the presence of 'id-kp-SignTempCert' only has informative character. The certificate for the temporary public signature key will (informally) look like this:

Temporary-User-Signature-Certificate-$CertPs_{TU}$:

- issuer=User
- validity=
  - T-notBefore
  - T-notAfter
- subject=User
- subjectAltName=UserAlt
- issuerAltName=UserAlt
- Key usage
  - critical=TRUE
  - digitalSignature=TRUE
  - all others=FALSE
- Extended key usage
  - critical=TRUE
  - id-kp-Temporary
- Basic constraints
  - critical=TRUE
  - cA=FALSE

The certificate for the temporary public encryption key looks similar, except for the Key usage extension:

Temporary-User-Encryption-Certificate-$CertPe_{TU}$:

- ...
- Key usage
  - critical=TRUE
  - keyEncipherment=TRUE
  - keyAgreement=TRUE
  - all others=FALSE

Clearly, a certificate verification procedure conform to the PKIX profile will reject this temporary certificate, because it is certified (signed) with an end user signature key. The (long term) user certificate $Cert_{Ps_U}$ has not set the 'keyCertSign' bit and the 'cA' flag in the Basic constraints extension is FALSE. Therefore we propose a somewhat relaxed certificate verification. A certificate will be valid, if

- *all* certificates in the verification chain are PKIX conform
  or if
- *all* of the following requirements are fulfilled:

1. The first (temporary) certificate has the following properties:
   1.1 issuer=subject,
   1.2 issuerAltName=subjectAltName,
   1.3 validity.T-notBefore $>$ validity.U-notBefore,
   1.4 validity.T-notAfter $<$ validity.U-notAfter,
   1.5 the Key Usage extension is critical,
   1.6 nonRepudiation=FALSE,
   1.7 keyCertSign=FALSE,
   1.8 cRLSign=FALSE,
   1.9 the Extended Key Usage extension is critical,
   1.10 'id-kp-Temporary' is present
   1.11 the Basic constraints extension is critical,
   1.12 cA=FALSE,
2. *all* other certificates in the certification chain are PKIX conform,

3. and the second (User's long term) Certificate has the following properties:

   3.1 the Key Usage extension is critical

   3.2 digitalSignature=TRUE

   3.3 'id-kp-SignTempKey' is present

It is obvious, that this requirements restrict the presented *temporary certificate concept* to the first certificate in the verification chain. Since the Basic constraints and Key Usage extension is set appropriate, it will not be possible to act as CA, by malicously presenting the temporary certificate to an ordinary (PKIX conform) verification procedure. Furthermore the validity period of the temporary certificate is smaller than the validity period of the long term user certificate and it is not possible to issue such temporary certificates for other subjects. Typically the validity-period of the temporary certificate will not exceed one day. The last requirement makes sure, that only the long term *signature* key can be used to produce temporary certificates and that this self-certification is allowed at all. Note, that other extensions, like Policy constraints are not affected by these changes, because we self-certify only user keys.

7 Security - Usability - Efficiency

In this section we will briefly recall the different variants for the SPKM credential management, compare them in terms of Security, Usability, Time-Efficiency and Space-Efficiency and give more concrete recommendations for implementation. To compare the time efficiency, we group the operations to be performed in *Once*, *Every GSS-session* and *Every GSS-context establishment* and estimate the workload in terms of modular (1024 bit) multiplications. To compare the space efficiency we estimate the number of bytes, which have to be stored permanently in *Secure Storage*, i.e. inside the PSE and *Insecure Storage*, i.e. on the harddisk. We may assume, that a user already got certified long term (1024 bit) RSA key pairs for signatures and encryption and that the SPKM - credential management like presented in the previous section is applied to all variants. We will only focus on additional time and space requirements. I.e. we neither consider the time needed to generate the long term keys, nor the space required to store them in the PSE. Since we neglect some operations, e.g. computing hash values, generation of random numbers etc. and have to 'convert' the time for some operations to our 'unit' (1024 bit modular multiplication), this estimates are very rough in nature. We may assume, that the secret RSA keys are stored in the CRT *RSAPrivateKey*-format [28, 29], i.e. $p, q, d_p = d \bmod p-1, d_q = d \bmod q-1$ and $invq = q^{-1} \bmod p$ to speed up the decryption and signature operation by application of the Chinese Remainder Theorem. In this case the decryption of the ciphertext C works like this:

$$\begin{aligned} M_q &= C^{d_q} \bmod q \\ M_p &= C^{d_p} \bmod p \\ M_p &= M_p - M_q \bmod p \\ M &= (M_p \cdot invq \bmod p)\, q + M_q \end{aligned}$$

We use an m bit window method (see e.g. [16]) for exponentiation, where l is the bitlength of the exponent. On average we will need about

$$2^{m-1} + \frac{2^m - 1}{2^m}\lceil l/m \rceil + l \qquad (1)$$

modular multiplications. Thus we may optimally choose $m = 5$ for bitlength up to 512 and $m = 6$ for bitlenght 768 and 1024. Therefore we need about $2 \cdot 628 + 2 = 1258$ modular (512 bit) multiplications for one 1024 bit RSA signature using the CRT. If we assume, that the complexity of the implemented multiplication algorithm is quadratic this equals about 315 modular (1024 bit) multiplications. If we use the Fermat number $F_4 = 2^{16} + 1$ as public exponent e, we can perform the encryption and signature verification with only 17 modular (1024 bit) multiplications. If we use a random e we would need about 1224 modular (1024 bit) multiplications. We assume, that the length of every (long term) certification path is one.

1. *Single Login*
   This variant is the most obvious way for handling SPKM credentials. The user opens the PSE by entering the PIN. The PSE is kept open and the secret keys are accessible, until the user logs out.

   1.1 Security
   Since the (long term) secret keys are exposed for a long time, we may classify the security **low**.

   1.2 Usability
   Since the user only has to enter the PIN once, the usability is **good**.

   1.3 Time-Efficiency

   1.3.1 *Once*
   /

   1.3.2 *Every GSS-session*
   /

   1.3.3 *Every GSS-context establishment*
   We use the Three-Way authentication presented in section 3. Client and server have to perform one signature, one verification,

one encryption and one decryption. If the certification path has length one, we have in total two RSA-decryptions and four RSA-encryptions. Thus we need about $2 \cdot 315 + 4 \cdot 17 =$ **698** multiplications (using $e = F_4$) or about $2 \cdot 315 + 4 \cdot 1224 =$ **5526** modular (1024 bit) multiplications in general.

1.4 Space-Efficiency

1.4.1 *Secure Storage*
/

1.4.2 *Insecure Storage*
/

2. *Multiple Login*
This variant is very similar to the *Single Login* variant. The difference is, that we close the PSE after every context establishment. Therefore the user has to enter the PIN for *every new context* to be established.

2.1 Security
Since the PSE with the (long term) secret keys is closed almost always, the security is **high**.

2.2 Usability
Since the user has to enter the PIN for *every* context to be established, the usability is **bad**.

2.3 Time-Efficiency

2.3.1 *Once*
/

2.3.2 *Every GSS-session*
/

2.3.3 *Every GSS-context establishment*
Like in 1.3.3 we need about **698** modular (1024 bit) multiplications, if we use $e = F_4$ and about **5526** modular (1024 bit) multiplications in general.

2.4 Space-Efficiency

2.4.1 *Secure Storage*
/

2.4.2 *Insecure Storage*
/

3. *Secure Single Login - RSA*
This variant uses the SSLogin mechanism discussed in section 5, where the temporary key pair is an RSA key-pair used for signatures *and* encryption.

3.1 Security
Since the (long term) secret keys are only exposed until the temporary certificate is signed, the security is **high**.

3.2 Usability
The user only has to enter the PIN once. Therefore the usability may be classified **good**.

3.3 Time-Efficiency

3.3.1 *Once*
/

3.3.2 *Every GSS-session*
We have to generate a temporary RSA-key pair at the time of Login. Since it has a limited lifetime, we consider 768 bit keys to be more than sufficient. A crude estimate for the key generation (in terms of 1024 bit modular multiplications) is about 108000. Note, that this estimate is based on practical measurements with SECUDE [7], rather than theoretical considerations. Since SECUDE takes a very conservative approach in prime generation, this value might be considerable smaller in other implementations. E.g. SECUDE uses 15 bases for the Miller-Rabin pseudo primality test and trial division with the first 1000 primes. Neglecting the trial division this corresponds to a probability $< 2^{-121}$, that p and q are composite 384 bit numbers instead of primes (see [23], page 147). Finally we need another 315 multiplications to sign the certificate containing the temporary public key. Thus we need about **108315** modular (1024 bit) multiplications in total.

3.3.3 *Every GSS-context establishment*
We have to perform two (768 bit) RSA-encryptions and two (768 bit) RSA-decryptions. We need about $2 \cdot 475 + 2 = 952$ modular (384 bit) multiplications for one decryption using the CRT and 17 respectively 926 modular (768 bit) multiplications for one encryption. The length of the verification path now is two (for only one certificate). Therefore we need $2 \cdot 17 = 34$ (using $e = F_4$) or $2 \cdot 1224 = 2448$ modular (1024 bit) multiplications to verifiy the peers certificate. Thus, if we assume quadratic complexity for multiplication we need totally about $2 \cdot 952 \cdot (384/1024)^2 + 2 \cdot 17 \cdot (768/1024)^2 + 34 \approx$ **321** modular (1024 bit) multiplications (using $e = F_4$) or $2 \cdot 952 \cdot (384/1024)^2 + 2 \cdot 926 \cdot (768/1024)^2 + 2448 \approx$ **3758** modular (1024 bit) multiplications in general.

3.4 Space-Efficiency

3.4.1 *Secure Storage*
/

3.4.2 *Insecure Storage*
To store the secret (768 bit) key in CRT-

format, we need about $5 \cdot 384/8 = 240$ byte. Furthermore we have to store the temporary certificate, which may be conservatively estimated with 500 bytes. Thus in total we need to store about **740** more bytes e.g. on the harddrive.

4. *Secure Single Login - DL based (without Precomputation)*
This variant also uses the SSLogin mechanism discussed in section 5, where the temporary key pairs are ElGamal for encryption and DSA for signatures. *All* exponentiations are performed using the (ordinary) window method [16] with a suitable window size depending on the bitlength of the exponent.

4.1 Security

Since the (long term) secret keys are only exposed until the temporary certificates are signed, the security is **high**.

4.2 Usability

Since the user only has to enter the PIN once, the usability may be classified **good**.

4.3 Time-Efficiency

4.3.1 *Once*

We have to generate a 768 bit prime p, and the parameters q, g_s and g_e. This is possible in the time needed to perform about **116000** modular (1024 bit) multiplications. Like in 3.3.2 this estimate is based on measurements with SECUDE and might be somewhat smaller in other implementations.

4.3.2 *Every GSS-session*

We have to generate a temporary ElGamal and DSA key pair at the time of Login. This may be performed by a simple exponentiation for each keypair. According to (1) this can be done with 926 modular (768 bit) multiplications for the ElGamal key pair (window width $m = 6$) and 207 modular (768 bit) multiplications with the 5-bit window method for the DSA key pair. Note, that for DSA the private key x_s and the public key $y_s = g_s^{x_s}$ are 160 bit numbers. Here we have to sign two certificates, i.e. we need about $2 \cdot 315 = 630$ modular (1024 bit) multiplications. Assuming quadratic complexity for multiplication this equals about $(768/1024)^2 \cdot (926 + 207) + 630 \approx$ **1267** modular (1024 bit) multiplications in total.

4.3.3 *Every GSS-context establishment*

An ElGamal encryption is essentially two exponentiations and a decryption is essentially another exponentiation. For each exponentiation we need about 926 (768 bit) multiplications. A DSA signature may be performed with about $207 + 20 = 227$ (768 bit) multiplications. Note, that we conservatively estimated the time to compute $s = (k^{-1}(h(m) + xr)) \bmod q$ with 20 (768 bit) multiplications. A signature verification needs about $2 \cdot 207 + 20 = 434$ modular (768 bit) multiplications. For the verification of the two (peer-) certificates we need $4 \cdot 17 = 68$ (1024 bit) multiplications using $e = F_4$ and about $4 \cdot 1224 = 4896$ modular (1024 bit) multiplications. Thus in total we need about $(768/1024)^2 \cdot (3 \cdot 926 + 227 + 434) + 68 \approx$ **2408** (1024 bit) multiplications using $e = F_4$ or $(768/1024)^2 \cdot (3 \cdot 926 + 227 + 434) + 4896 \approx$ **7236** modular (1024 bit) multiplications in general.

4.4 Space-Efficiency

4.4.1 *Secure Storage*

We store the global DL parameters on the harddrive and compute a hash-value of this parameters. Before we use them, we check whether this parameters stored in insecure memory are still OK. Therefore we only need to store the hash-value (e.g. **20** bytes with SHA-1 or RIPEMD-160) in the secure environment.

4.4.2 *Insecure Storage*

Here we have to store p, q, g_e, g_s permanently. This will take about $(768 + 160 + 768 + 768)/8 = 308$ bytes. Furthermore we have to store the secret exponents x_e, x_s needing about $(768 + 160)/8 = 116$ bytes and the two (own) temporary certificates with about $500 + 424 = 924$ bytes. Note, that a DSA certificate is smaller, than an ElGamal certificate. All in all we have to store about $308 + 116 + 924 =$ **1348** bytes on the harddrive.

5. *Secure Single Login - DL based (with Precomputation)*
This variant also uses the SSLogin mechanism discussed in section 5, where the temporary key pairs are ElGamal for encryption and DSA for signatures. The difference to variant 4 is, that we perform the exponentiations of g_e and g_s using the BGMW precomputation variant proposed in [4]. The average number of multiplications using this exponentiation technique is

$$\frac{2^m - 1}{2^m} \lceil l/m \rceil + 2^m - 3, \qquad (2)$$

where m is the window width and l is the bitlength of the exponent. We have to precomputate and store

$$g^{im}, 0 \leq i \leq \lceil l/m \rceil - 1, \quad (3)$$

where $g = g_e$ for ElGamal or $g = g_s$ for DSA respectively. For all other exponentiations the ordinary window method [16] is used.

5.1 Security

Since the (long term) secret keys are only exposed until the temporary certificates are signed, the security is **high**.

5.2 Usability

Since the user only has to enter the PIN once, the usability may be classified **good**.

5.3 Time-Efficiency

5.3.1 *Once*

The time to generate the global DL parameters is of course equal to 4.3.1 . We need about 116000 modular (1024 bit) multiplications. Furthermore we have to precomputate and store the powers of g_e and g_s (see (3)). According to (2) we optimally choose $m_e = 5$ for the 768 bit ElGamal exponentiations and $m_s = 4$ for the 160 bit DSA exponentiations. Therefore (according to (3)) we need 156 squarings for the DSA-precomputation and 763 modular (768 bit) squarings for the ElGamal-precomputation. Assuming quadratic complexity for multiplication we need about $(768/1024)^2 \cdot (156 + 763) + 116000 \approx$ **116517** modular (1024 bit) multiplications in total.

5.3.2 *Every GSS-session*

We have to generate a temporary ElGamal and DSA key pair at the time of Login. This may be performed by a simple exponentiation for each keypair. According to (2) we only need about 22 modular (768 bit) multiplications for the computation of $g_s^{x_s}$ and about 58 modular (768 bit) multiplications to compute $g_e^{x_e}$. Like in 4.3.2 we have to sign two certificates, i.e. we need about $2 \cdot 315 = 630$ modular (1024 bit) multiplications. Assuming quadratic complexity for multiplication this equals about $(768/1024)^2 \cdot (22 + 58) + 630 \approx$ **675** modular (1024 bit) multiplications.

5.3.3 *Every GSS-context establishment*

An ElGamal encryption equals essentially two exponentiations. The first exponentiation $(g^k \bmod p)$ needs about 58 multiplications using the precomputation variant. The second encryption-exponentiation and the decryption-exponentiation may be performed with about 926 (768 bit) multiplications each using the ordinary 6-bit window method. A DSA signature may be performed with about $22 + 20 = 42$ (768 bit) multiplications. A signature verification needs about $22 + 207 + 20 = 249$ modular (768 bit) multiplications. For the verification of the two (peer-) certificates we need $4 \cdot 17 = 68$ (1024 bit) multiplications using $e = F_4$ and about $4 \cdot 1224 = 4896$ modular (1024 bit) multiplications in general. Thus in total we need about $(768/1024)^2 \cdot (58 + 2 \cdot 926 + 227 + 434) + 68 \approx$ **1540** (1024 bit) multiplications (using $e = F_4$) or $(768/1024)^2 \cdot (58 + 2 \cdot 926 + 227 + 434) + 4896 \approx$ **6368** modular (1024 bit) multiplications in general.

5.4 Space-Efficiency

5.4.1 *Secure Storage*

We store the global DL parameters on the harddrive and compute a hash over this parameters. Before we use them, we check whether the parameters stored in insecure memory are still OK. Therefore we only need to store the hash-value (e.g. **20** bytes with SHA-1 or RIPEMD-160) in the secure environment, e.g. on a smartcard.

5.4.2 *Insecure Storage*

Here we have to store $p, q, g_e^{5i}, g_s^{4j}, 0 \leq i < 40, 0 \leq j < 154$ permanently. This will take about $(768 + 160 + 768 \cdot 154 + 768 \cdot 40)/8 = 18740$ bytes. Furthermore we have to store the secret exponents x_e, x_s needing about $(768 + 160)/8 = 116$ bytes and the two (own) temporary certificates with about $500 + 424 = 924$ bytes. Note, that a DSA certificate is smaller, than an ElGamal certificate. All in all we have to store about $18740 + 116 + 924 =$ **19780** bytes on the harddrive.

The results of our discussion are summarized in Table 1. From this table we see, that we can combine *security* and *usability* at only slightly higher expenses at Login - time and context establishment. The application of DL based algorithms turns out to be very well suited to implement the SSLogin variant, because the time for the actual key generation is negligible small and may be well performed at Login

Variant	Security	Usability	Time-Efficiency			Space-Efficiency	
			Once Mult.	Session Mult.	Context Mult. $e = F_4$ / gen.	Secure Byte	Insecure Byte
1. *Single Login*	low	good	/	/	698 / 5526	/	/
2. *Multiple Login*	high	bad	/	/	698 / 5526	/	/
3. *SSLogin - RSA*	high	good	/	108315	321 / 3758	/	740
4. *SSLogin - DL (naive)*	high	good	116000	1267	2408 / 7236	20	1348
5. *SSLogin - DL (prec.)*	high	good	116517	675	1540 / 6368	20	19780

Table 1. Variants for the SPKM credential management

- time. Furthermore it is possible to speed up the context establishment by applying exponentiation variants with precomputation, which is not possible in RSA-type cryptosystems. In this case only slightly more memory is needed to store the precomputed values. Since the additional storage of about 20000 byte should be possible in every implementation, we recommend the application of this exponentiation technique.

8 Conclusion

In this paper we briefly recalled the necessary basics of the GSS-API, SPKM and X.509 v3 / PKIX public key certification. We proposed concrete changes to the PKIX profile to enable the application of a *Secure Single Login* mechanism, which combines security and user comfort. Furthermore we proposed to extend the list of recommended SPKM K-algs by signature and encryption algorithms based on discrete logarithms, like DSA and ElGamal. We compared the different variants for SPKM credential management in terms of security, usability and efficiency and gave recommendations for the implementation of the different variants. The proposed changes for the certificate verification in section 6 will appear in a more technical form in a forthcomming internet-draft.

9 Acknowledgement

I would like to thank the entire SECUDE - team, especially Stephan André, Petra Glöckner and Hans Schupp for all the fruitful discussions and the anonymous referees for providing several helpful remarks.

References

[1] C. Adams: *The Simple Public-Key GSS-API Mechanism (SPKM)*, RFC 2025, Okt. 1996

[2] ANSI: *Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry: Transport of Symmetric Algorithm Keys Using RSA*, X9.44-1993

[3] E. Baize, D. Pinkas: *The Simple and Protected GSS-API Negotiation Mechanism*, Internet-Draft, 16th May 1997

[4] E. Brickell, D. Gordon, K. McCurley, D. Wilson: *Fast Exponentiation with Precomputation*, Proceedings of EUROCRYPT '93 - LNCS 658, Springer, Berlin, 1994, pp. 200-207

[5] J.M. DeLaurentis: *A Further Weakness in the Common Modulus Protocol for the RSA Cryptosystem*, Cryptologia, v.8, n.3, Jul 1984, pp. 253-259

[6] T. ElGamal: *A public key cryptosystem and a signature schem based on discrete logarithms*, IEEE Transactions on Information Theory 31, 1985, pp. 469-472

[7] GMD/TKT-SIT: *SECUrity Development Environment for Open Networks - Online Documentation*, http://www.darmstadt.gmd.de/secude/Doc/index.htm

[8] R. Housley, W. Ford, S. Farrel, D. Solo: *Internet Public Key Infrastructure, Part I: X.509 Certificate and CRL Profile*, Internet Draft: draft-ietf-pkix-ipki-part1-06.txt, 15th October 1997

[9] D. Hühnlein: *Generic Security - The GSS-API and three of its mechanisms*, (in german), in FIFF-Communication 97/3 'Security Infrastructures', ISSN 0938-3476, pp. 38-41, 1997

[10] IEEE : *IEEE P1363 Working Draft*, via ftp://stdsbbs.ieee.org/pub/p1363/predrafts

[11] ISO/IEC 9594-8: *Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, CCITT/ITU Recommendation X.509, 1993.

[12] ISO/IEC JTC 1/SC 21/WG 4 and ITU-T Q15/7: *Final Text of Draft Amendment 1 to ISO/IEC 9594-8 on Certificate Extensions*, June 1997

[13] ISO/IEC 9798-3: *Information technology - Security Techniques - Entity authentication mechanisms - Part 3: Entity authentication using a public key algorithm*, ISO/IEC, 1993.

[14] C. Kaufman: *Distributed Authentication Security Service (DASS)*, RFC 1507, Sept. 1993

[15] S. Kent: *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management*, RFC 1422, 1993

[16] D.E. Knuth: *The Art of Computer Programming Vol. 2: Seminumerical algorithms*, Addison-Wesley, Reading MA, 1981

[17] N. Koblitz: *Elliptic Curve Cryptosystems*, Mathematics of Computation, Vol. 48, N. 177, Jan. 1987, pp. 203-209

[18] J. Kohl, C. Neumann: *The Kerberos Network Authentication Service (V5)*, RFC 1510, Sep. 1993

[19] C.H. Lim, P.J. Lee: *More Flexible Exponentiation with Precomputation*, Proceedings of CRYPTO '94, Springer, Berlin, SS. 95-107

[20] J. Linn: *Generic Security Service Application Program Interface*, RFC 1508, Sep. 1993

[21] J. Linn: *Generic Security Service Application Program Interface Version 2*, RFC 2078, Jan. 1997

[22] J. Linn: *The Kerberos Version 5 GSS-API Mechanism*, RFC 1964, Juni 1996

[23] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone: *Handbook of Applied Cryptography*, CRC Press, ISBN 0-8493-8523-7, 1996

[24] V. Miller: *Use of Elliptic Curves in Cryptography*, Proceedings of Crypto '85, LNCS 218, Springer, Berlin, 1986

[25] National Institute of Standards and Technology (NIST): *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication 186 (FIPS-186), 19th May, 1994

[26] K. Nyberg, R. Rüppel: *A new signature scheme based on the DSA giving message recovery*, 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, Nov. 3-5, 1993.

[27] OSI/OIW: *Part 12 - OS Security (Stable)*, June 1995 http://nemo.ncsl.nist.gov/oiw/agreements/stable/OSI/12s_9506.txt

[28] J. J. Quisquater and C. Couvreur: *Fast Decipherment Algorithm for RSA Public-Key Cryptosystem*, Electronics Letters, vol. 18, no. 21, Oct 1982, pp. 905-907

[29] RSA Lab.: *PKCS#1 - RSA Encryption Standard, Version 1.5*, 1993

[30] RSA Lab.: *PKCS#3 - Diffie-Hellman Key-Agreement Standard, Version 1.4*, 1993

[31] RSA Lab.: *PKCS#5 - Password based encryption, Version 1.5*, 1993

[32] RSA Lab.: *PKCS#10 - Certification Request Syntax Standard Version 1.0*, 1993

[33] G.J. Simmons: *A 'Weak' Privacy Protocol Using the RSA Cryptosystem*, Cryptologia, v.7, n.2, Apr. 1983, pp. 180-182

[34] J. Wray: *Generic Security Service API: C-bindings*, RFC 1509, Sep. 1993

[35] W. Yeong, T. Howes, S. Killie: *CURRENT LDAP Version 2*, RFC 1777, Mar. 1995