INTERNET-DRAFT
draft-ietf-cat-kerberos-pk-init-02.txt
Updates: RFC 1510
expires April 19, 1997

Clifford Neuman
Brian Tung
ISI
John Wray
Digital Equipment Corporation
Jonathan Trostle
CyberSafe Corporation

Public Key Cryptography for Initial Authentication in Kerberos

0. Status Of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as ``work in progress.''

To learn the current status of any Internet-Draft, please check the ``1id-abstracts.txt'' listing contained in the Internet-Drafts Shadow Directories on ds.internic.net (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

The distribution of this memo is unlimited. It is filed as draft-ietf-cat-kerberos-pk-init-02.txt, and expires April 19, 1997. Please send comments to the authors.

1. Abstract

This document defines extensions to the Kerberos protocol specification (RFC 1510, "The Kerberos Network Authentication Service (V5)", September 1993) to provide a method for using public key cryptography during initial authentication. The method defined specifies the way in which preauthentication data fields and error data fields in Kerberos messages are to be used to transport public key data.

2. Motivation

Public key cryptography presents a means by which a principal may demonstrate possession of a key, without ever having divulged this

key to anyone else. In conventional cryptography, the encryption key and decryption key are either identical or can easily be derived from one another. In public key cryptography, however, neither the public key nor the private key can be derived from the other (although the private key RECORD may include the information required to generate BOTH keys). Hence, a message encrypted with a public key is private, since only the person possessing the private

key can decrypt it; similarly, someone possessing the private key can also encrypt a message, thus providing a digital signature.

Furthermore, conventional keys are often derived from passwords, so messages encrypted with these keys are susceptible to dictionary attacks, whereas public key pairs are generated from a pseudo-random number sequence. While it is true that messages encrypted using public key cryptography are actually encrypted with a conventional secret key, which is in turn encrypted using the public key pair, the secret key is also randomly generated and is hence not vulnerable to a dictionary attack.

The advantages provided by public key cryptography have produced a demand for its integration into the Kerberos authentication protocol. The public key integration into Kerberos described in this document has three goals.

First, by allowing users to register public keys with the KDC, the KDC can be recovered much more easily in the event it is compromised. With Kerberos as it currently stands, compromise of the KDC is disastrous. All keys become known by the attacker and all keys must be changed. Second, we allow users that have public key certificates signed by outside authorities to obtain Kerberos credentials for access to Kerberized services. Third, we obtain the above benefits while maintaining the performance advantages of Kerberos over protocols that use only public key authentication.

If users register public keys, compromise of the KDC does not divulge their private key. Compromise of security on the KDC is still a problem, since an attacker can impersonate any user by creating a ticket granting ticket for the user. When the compromise is detected, the KDC can be cleaned up and restored from backup media and loaded with a backup private/public key pair. Keys for application servers are conventional symmetric keys and must be changed.

Note: If a user stores his private key, in an encrypted form, on the KDC, then it may be desirable to change the key pair, since the private key is encrypted using a symmetric key derived from a password (as described below), and can therefore be vulnerable to dictionary attack if a good password policy is not used. Alternatively, if the encrypting symmetric key has 56 bits, then it may also be desirable to change the key pair after a short period

due to the short key length. The KDC does not have access to the user's unencrypted private key.

There are two important areas where public key cryptography will have immediate use: in the initial authentication of users registered with the KDC or using public key certificates from outside authorities, and to establish inter-realm keys for cross-realm authentication. This memo describes a method by which the first of these can be done. The second case will be the topic for a separate proposal.

Some of the ideas on which this proposal is based arose during discussions over several years between members of the SAAG, the IETF-CAT working group, and the PSRG, regarding integration of Kerberos and SPX. Some ideas are drawn from the DASS system, and

similar extensions have been discussed for use in DCE. These changes are by no means endorsed by these groups. This is an attempt to revive some of the goals of that group, and the proposal approaches those goals primarily from the Kerberos perspective.

3. Initial authentication of users with public keys

This section describes the extensions to Version 5 of the Kerberos protocol that will support the use of public key cryptography by users in the initial request for a ticket granting ticket.

Roughly speaking, the following changes to RFC 1510 are proposed: Users can initially authenticate using public key or conventional (symmetric key) cryptography. After a KDC compromise, the KDC replies with an error message that informs the client of the new KDC public backup key. Users must authenticate using public key cryptography in response to the error message. If applicable, the client generates the new user secret key at this point as well.

Public key initial authentication is performed using either the RSA encryption or Diffie Hellman public key algorithms. There is also an option to allow the user to store his/her private key encrypted in the user password in the KDC database; this option solves the problem of transporting the user private key to different workstations. The combination of this option and the provision for conventional symmetric key authentication allows organizations to smoothly migrate to public key cryptography.

This proposal will allow users either to use keys registered directly with the KDC, or to use keys already registered for use with X.509, PEM, or PGP, to obtain Kerberos credentials. These credentials can then be used, as before, with application servers supporting Kerberos. Use of public key cryptography will not be a requirement for Kerberos, but if one's organization runs a KDC

supporting public key, then users may choose to be registered with a public key pair, instead of or in addition to the current secret key.

The application request and response between Kerberos clients and application servers will continue to be based on conventional cryptography, or will be converted to use user-to-user authentication. There are performance issues and other reasons that servers may be better off using conventional cryptography. For this proposal, we feel that 80 percent of the benefits of integrating public key with Kerberos can be attained for 20 percent of the effort, by addressing only initial authentication. This proposal does not preclude separate extensions.

With these changes, users will be able to register public keys, only in realms that support public key, but they will still be able to perform initial authentication from a client that does not support public key. They will be able to use services registered in any realm. Furthermore, users registered with conventional keys will be able to use any client.

This proposal addresses three ways in which users may use public key cryptography for initial authentication with Kerberos, with

minimal change to the existing protocol. Users may register keys directly with the KDC, or they may present certificates by outside certification authorities (or certifications by other users) attesting to the association of the public key with the named user. In both cases, the end result is that the user obtains a conventional ticket granting ticket or conventional server ticket that may be used for subsequent authentication, with such subsequent authentication using only conventional cryptography.

Additionally, users may also register a digital signature verification key with the KDC. We provide this option for the licensing benefits, as well as a simpler variant of the initial authentication exchange. However, this option relies on the client to generate random keys.

We first consider the case where the user's key is registered with the KDC.

3.1 Definitions

Before we proceed, we will lay some groundwork definitions for encryption and signatures. We propose the following definitions of signature and encryption modes (and their corresponding values on the wire):

```
    #define ENCTYPE_SIGN_MD5_RSA      0x0011
```

```
#define ENCTYPE_ENCRYPT_RSA_PRIV  0x0021
#define ENCTYPE_ENCRYPT_RSA_PUB   0x0022
```

allowing further modes to be defined accordingly.

In the exposition below, we will use the notation E (T, K) to denote the encryption of data T, with key (or parameters) K.

If E is ENCTYPE_SIGN_MD5_RSA, then

```
E (T, K) = {T, RSAEncryptPrivate (MD5Hash (T), K)}
```

If E is ENCTYPE_ENCRYPT_RSA_PRIV, then

```
E (T, K) = RSAEncryptPrivate (T, K)
```

Correspondingly, if E is ENCTYPE_ENCRYPT_RSA_PUB, then

```
E (T, K) = RSAEncryptPublic (T, K)
```

3.2 Initial request for user registered with public key on KDC

In this scenario it is assumed that the user is registered with a public key on the KDC. The user's private key may be held by the user, or it may be stored on the KDC, encrypted so that it cannot be used by the KDC.

3.2.1 User's private key is stored locally

Implementation of the changes in this section is REQUIRED.

In this section, we present the basic Kerberos V5 pk-init protocol that all conforming implementations must support. The key features of the protocol are: (1) easy, automatic (for the clients) recovery after a KDC compromise, (2) the ability for a realm to support a mix of old and new Kerberos V5 clients with the new clients being a mix of both public key and symmetric key configured clients, and (3) support for Diffie-Hellman (DH) key exchange as well as RSA public key encryption. The benefit of having new clients being able to use either symmetric key or public key initial authentication is that it allows an organization to roll out the new clients as rapidly as possible without having to be concerned about the need to purchase additional hardware to support the CPU intensive public key cryptographic operations.

In order to give a brief overview of the four protocols in this section, we now give diagrams of the protocols. We denote encryption of message M with key K by {M}K and the signature of message M with key K by [M]K. All messages from the KDC to the

client are AS_REP messages unless denoted otherwise; similarly, all messages from the client to the KDC are AS_REQ messages unless denoted otherwise. Since only the padata fields are affected by this specification in the AS_REQ and AS_REP messages, we do not show the other fields. We first show the RSA encryption option in normal mode:

```
   certifier list, [cksum, time, nonce, kdcRealm,
   kdcName]User PrivateKey
C ----------------------------------------------------> KDC

   list of cert's, {[encReplyKey, nonce]KDC privkey}
   EncReplyTmpKey, {EncReplyTmpKey}Userpubkey
C <---------------------------------------------------- KDC
```

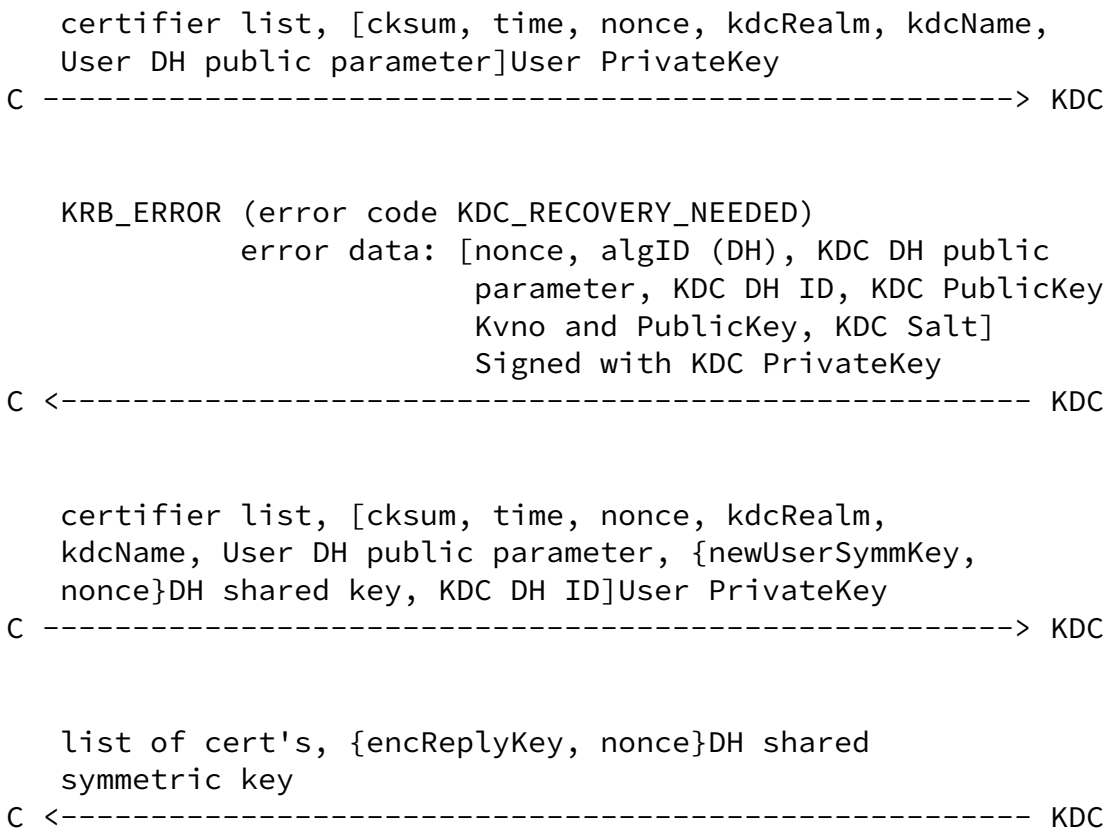
We now show RSA encryption in recovery mode:

```
   certifier list, [cksum, time, nonce, kdcRealm,
   kdcName]User PrivateKey
C ----------------------------------------------------> KDC


   KRB_ERROR (error code KDC_RECOVERY_NEEDED)
              error data: [nonce, algID (RSA),
              KDC PublicKey Kvno and PublicKey, KDC Salt]
              Signed with KDC PrivateKey
C <---------------------------------------------------- KDC


   certifier list, [cksum, time, nonce, kdcRealm, kdcName,
   {newUserSymmKey, nonce}KDC public key]User PrivateKey
C ----------------------------------------------------> KDC


   list of cert's, {[encReplyKey, nonce]KDC privkey}
   EncReplyTmpKey, {EncReplyTmpKey}Userpubkey
C <---------------------------------------------------- KDC
```

Next, we show Diffie Hellman in normal mode:

```
   certifier list, [cksum, time, nonce, kdcRealm, kdcName,
   User DH public parameter]User PrivateKey
C ----------------------------------------------------> KDC


   list of cert's, {encReplyKey, nonce}DH shared symmetric
   key, [KDC DH public parameter]KDC Private Key
C <---------------------------------------------------- KDC
```

Finally, we show Diffie Hellman in recovery mode:



If the user stores his private key locally, the initial request to the KDC for a ticket granting ticket proceeds according to RFC 1510, except that a preauthentication field containing a nonce signed by the user's private key is included. The preauthentication field may also include a list of the root certifiers trusted by the user.

```
PA-PK-AS-ROOT ::= SEQUENCE {
        rootCert[0]         SEQUENCE OF OCTET STRING,
        signedAuth[1]       SignedPKAuthenticator
}

SignedPKAuthenticator ::= SEQUENCE {
        authent[0]          PKAuthenticator,
        authentSig[1]       Signature
}

PKAuthenticator ::= SEQUENCE {
        cksum[0]            Checksum OPTIONAL,
        cusec[1]            INTEGER,
        ctime[2]            KerberosTime,
        nonce[3]            INTEGER,
        kdcRealm[4]         Realm,
```

```
        kdcName[5]          PrincipalName,
        clientPubValue[6]   SubjectPublicKeyInfo OPTIONAL,
                               -- DH algorithm
        recoveryData[7]     RecoveryData OPTIONAL
                               -- Recovery Alg.
}

RecoveryData ::= SEQUENCE {
        clientRecovData[0]  ClientRecovData,
        kdcPubValueId[1]    INTEGER OPTIONAL
                                -- DH algorithm, copied
                                -- from KDC response
}

ClientRecovData ::= CHOICE {
        newPrincKey[0]      EncryptedData, -- EncPaPkAsRoot
                                           -- encrypted with
                                           -- either KDC
                                           -- public key or
                                           -- DH shared key
        recovDoneFlag[1]    INTEGER        -- let KDC know that
                                           -- recovery is done
                                           -- when user uses a
                                           -- mix of clients or
                                           -- does not want to
                                           -- keep a symmetric
                                           -- key in the database
}

EncPaPkAsRoot ::= SEQUENCE {
        newSymmKey[0]       EncryptionKey  -- the principal's new
                                           -- symmetric key
        nonce[1]            INTEGER        -- the same nonce as
                                           -- the one in the
                                           -- PKAuthenticator
}

Signature ::= SEQUENCE {
        sigType[0]          INTEGER,
        kvno[1]             INTEGER OPTIONAL,
        sigHash[2]          OCTET STRING
}
```

Notationally, sigHash is then

```
    sigType (authent, userPrivateKey)
```

where userPrivateKey is the user's private key (corresponding to the public key held in the user's database record). Valid sigTypes are thus far limited to the above-listed ENCTYPE_SIGN_MD5_RSA; we expect that other types may be listed (and given on-the-wire values between

0x0011 and 0x001f).

The format of each certificate depends on the particular service used. (Alternatively, the KDC could send, with its reply, a sequence of certifications (see below), but since the KDC is likely to have more certifications than users have trusted root certifiers, we have chosen the first method.) In the event that the client believes it already possesses the current public key of the KDC, a zero-length root-cert field is sent.

The fields in the signed authenticator are the same as those in the Kerberos authenticator; in addition, we include a client-generated nonce, and the name of the KDC. The structure is itself signed using the user's private key corresponding to the public key registered with the KDC. We include the newSymmKey field so clients can generate a new symmetric key (for users, this key is based on a password and a salt value generated by the KDC) and confidentially send this key to the KDC during the recovery phase.

We now describe the recovery phase of the protocol. There is a bit associated with each principal in the database indicating whether recovery for that principal is necessary. After a KDC compromise, the KDC software is reloaded from backup media and a new backup KDC public/private pair is generated. The public half of this pair is then either made available to the KDC, or given to the appropriate certification authorities for certification. The private half is not made available to the KDC until after the next compromise clean-up. If clients are maintaining a copy of the KDC public key, they also have a copy of the backup public key.

After the reload of KDC software, the bits associated with recovery of each principal are all set. The KDC clears the bit for each principal that undergoes the recovery phase. In addition, there is a bit associated with each principal to indicate whether there is a valid symmetric key in the database for the principal. These bits are all cleared after the reload of the KDC software (the old symmetric keys are no longer valid). Finally, there is a bit associated with each principal indicating whether that principal still uses non-public key capable clients. If a user principal falls into this category, a public key capable client cannot transparently re-establish a symmetric key for that user, since the older clients would not be able to compute the new symmetric key that includes hashing the password with a KDC supplied salt value. The re-establishment of the symmetric key in this case is outside the scope of this protocol.

One method of re-establishing a symmetric key for public key capable clients is to generate a hash of the user password and a KDC supplied salt value. The KDC salt is changed after every compromise of the KDC. In the recovery protocol, if the principal

does not still use old clients, the KDC supplied salt is sent to the client principal in a KRB_ERROR message with error code KDC_RECOVERY_NEEDED. The error data field of the message contains the following structure which is encoded into an octet string.

```
PA-PK-KDC-ERR ::= CHOICE {
        recoveryDhErr      SignedDHError,   -- Used during recovery
                                            -- when algorithm is DH
                                            -- based
        recoveryPKEncErr   SignedPKEncError -- Used during recovery
                                            -- for PK encryption
                                            -- (RSA,...)
}

SignedDHError ::= SEQUENCE {
        dhErr              DHError,
        dhErrSig           Signature
}

SignedPKEncError ::= SEQUENCE {
        pkEncErr           PKEncryptError,
        pkEncErrSig        Signature
}

DHError ::= SEQUENCE {
        nonce              INTEGER,               -- From AS_REQ
        algorithmId        INTEGER,               -- DH algorithm
        kdcPubValue        SubjectPublicKeyInfo,  -- DH algorithm
        kdcPubValueId      INTEGER,               -- DH algorithm
        kdcPublicKeyKvno   INTEGER OPTIONAL,      -- New KDC public
                                                  -- key kvno
        kdcPublicKey       OCTET STRING OPTIONAL, -- New KDC pubkey
        kdcSalt            OCTET STRING OPTIONAL  -- If user uses
                                                  -- only new
                                                  -- clients
}

PKEncryptError ::= SEQUENCE {
        nonce              INTEGER,               -- From AS_REQ
        algorithmId        INTEGER,               -- Public Key
                                                  -- encryption alg
        kdcPublicKeyKvno   INTEGER OPTIONAL,      -- New KDC public
                                                  -- key kvno
        kdcPublicKey       OCTET STRING OPTIONAL, -- New KDC public
                                                  -- key
        kdcSalt            OCTET STRING OPTIONAL  -- If user uses
                                                  -- only new
                                                  -- clients
}
```

The KDC_RECOVERY_NEEDED error message is sent in response to a client AS_REQ message if the client principal needs to be recovered, unless the client AS_REQ contains the PKAuthenticator with a nonempty RecoveryData field (in this case the client has already received the KDC_RECOVERY_NEEDED error message. We will also see in section 3.2.2 that a different error response is sent by the KDC if the encrypted user private key is stored in the KDC database.) If the client principal uses only new clients, then the kdcSalt field is returned; otherwise, the kdcSalt field is absent.

If the client uses the Diffie Hellman algorithm during the recovery phase then the DHError field contains the public Diffie Hellman

parameter (kdcPubValue) for the KDC along with an identifier (kdcPubValueID). The client will then send this identifier to the KDC in an AS_REQ message; the identifier allows the KDC to look up the Diffie Hellman private value corresponding to the identifier. Depending on how often the KDC updates its private Diffie Hellman parameters, it will have to store anywhere between a handful and several dozen of these identifiers and their parameters. The KDC must send its Diffie Hellman public value to the client first so the client can encrypt its new symmetric key.

In the case where the user principal does not need to be recovered and the user still uses old clients as well as new clients, the KDC_ERR_NULL_KEY error is sent in response to symmetric AS_REQ messages when there is no valid symmetric key in the KDC database. This situation can occur if the user principal has been recovered but no new symmetric key has been established in the database.

In addition, the two error messages with error codes KDC_ERR_PREAUTH_FAILED and KDC_ERR_PREAUTH_REQUIRED are modified so the error data contains the kdcSalt encoded as an OCTET STRING. The reason for the modification is to allow principals that use new clients only to have their symmetric key transparently updated by the client software during the recovery phase. The kdcSalt is used to create the new symmetric key. As a performance optimization, the kdcSalt is stored in the /krb5/salt file along with the realm. Thus the /krb5/salt file consists of realm-salt pairs. If the file is missing, or the salt is not correct, the above error messages allow the client to find out the correct salt. New clients which are configured for symmetric key authentication attempt to preauthenticate with the salt from the /krb5/salt file as an input into their key, and if the file is not present, the new client does not use preauthentication. The error messages above return either the correct salt to use, or no salt at all which indicates that the principal is still using old clients (the client software should use the existing mapping from the user password to the symmetric key).

In order to assure interoperability between clients from different vendors and organizations, a standard algorithm is needed for creating the symmetric key from the principal password and kdcSalt. The algorithm for creating the symmetric key is as follows: take the SHA-1 hash of the kdcSalt concatenated with the principal password and use the 20 byte output as the input into the existing key generation process (string to key function). After a compromise, the KDC changes the kdcSalt; thus, the recovery algorithm allows users to obtain a new symmetric key without actually changing their password.

The response from the KDC would be identical to the response in RFC 1510, except that instead of being encrypted in the secret key shared by the client and the KDC, it is encrypted in a random key freshly generated by the KDC (of type ENCTYPE_ENC_CBC_CRC). A preauthentication field (specified below) accompanies the response, optionally containing a certificate with the public key for the KDC (since we do not assume that the client knows this public key), and a package containing the secret key in which the rest of the response is encrypted, along with the same nonce used in the rest of the response, in order to prevent replays. This package is itself

signed with the private key of the KDC, then encrypted with the symmetric key that is returned encrypted in the public key of the user (or for Diffie Hellman, encrypted in the shared secret Diffie Hellman symmetric key).

Pictorially, in the public key encryption case we have:

kdcCert, {[encReplyKey, nonce] Sig w/KDC privkey}EncReplyTmpKey, {EncReplyTmpKey}Userpubkey

Pictorially, in the Diffie Hellman case we have:

kdcCert, {encReplyKey, nonce}DH shared symmetric key, [DH public value]Sig w/KDC privkey

```
PA-PK-AS-REP ::= SEQUENCE {
        kdcCert[0]          SEQUENCE OF Certificate,
        encryptShell[1]     EncryptedData,
                                    -- EncPaPkAsRepPartShell
                                    -- encrypted by
                                    -- encReplyTmpKey or DH
                                    -- shared symmetric key
        pubKeyExchange[2]   PubKeyExchange OPTIONAL,
                                    -- a choice between
                                    -- a KDC signed DH
                                    -- value and a public
                                    -- key encrypted
                                    -- symmetric key.
                                    -- Not needed after
```

```
                                        -- recovery when
                                        -- DH is used.
}

PubKeyExchange ::= CHOICE {
        signedDHPubVal      SignedDHPublicValue,
        encryptKey          EncryptedData
                                        -- EncPaPkAsRepTmpKey
                                        -- encrypted by
                                        -- userPublicKey
}

SignedDHPublicValue ::= SEQUENCE {
        dhPublic[0]         SubjectPublicKeyInfo,
        dhPublicSig[1]      Signature
}

EncPaPkAsRepPartShell ::= SEQUENCE {
        encReplyPart[0]     EncPaPkAsRepPart,
        encReplyPartSig[1]  Signature OPTIONAL
                                -- encReplyPart
                                -- signed by kdcPrivateKey
                                -- except not present in
                                -- DH case
}


EncPaPkAsRepPart ::= SEQUENCE {
        encReplyKey[0]      EncryptionKey,
        nonce[1]            INTEGER,
}

EncPaPkAsRepTmpKey ::= SEQUENCE {
        encReplyTmpKey[0]   EncryptionKey
}
```

The kdc-cert specification is lifted, with slight modifications, from v3 of the X.509 certificate specification:

```
Certificate ::= SEQUENCE {
        version[0]          Version DEFAULT v1 (1),
        serialNumber[1]     CertificateSerialNumber,
        signature[2]        AlgorithmIdentifier,
        issuer[3]           PrincipalName,
        validity[4]         Validity,
        subjectRealm[5]     Realm,
        subject[6]          PrincipalName,
        subjectPublicKeyInfo[7] SubjectPublicKeyInfo,
        issuerUniqueID[8]   IMPLICIT UniqueIdentifier OPTIONAL,
        subjectUniqueID[9]  IMPLICIT UniqueIdentifier OPTIONAL,
        authentSig[10]      Signature
```

}

The kdc-cert must have as its root certification one of the certifiers sent to the KDC with the original request. If the KDC has no such certification, then it will instead reply with a KRB_ERROR of type KDC_ERROR_PREAUTH_FAILED. If a zero-length root-cert was sent by the client as part of the PA-PK-AS-ROOT, then a correspondingly zero-length kdc-cert may be absent, in which case the client uses its copy of the KDC's public key. In the case of recovery, the client uses its copy of the backup KDC public key.

Upon receipt of the response from the KDC, the client will verify the public key for the KDC from PA-PK-AS-REP preauthentication data field. The certificate must certify the key as belonging to a principal whose name can be derived from the realm name. If the certificate checks out, the client then decrypts the EncPaPkAsRepPart and verifies the signature of the KDC. It then uses the random key contained therein to decrypt the rest of the response, and continues as per RFC 1510. Because there is direct trust between the user and the KDC, the transited field of the ticket returned by the KDC should remain empty. (Cf. Section 3.3.)

Examples

We now give several examples illustrating the protocols in this section. Encryption of message M with key K is denoted {M}K and the signature of message M with key K is denoted [M]K.

Example 1: The requesting user principal needs to be recovered and uses only new clients. The recovery algorithm is Diffie Hellman (DH). Then the exchange sequence between the user principal and the KDC is:

```
Client  --------> AS_REQ (with or without preauth) --------> KDC

Client  <--- KRB_ERROR (error code KDC_RECOVERY_NEEDED) <--- KDC
             error data: [nonce, algID (DH), KDC DH public parameter,
                          KDC DH ID, KDC PublicKey Kvno and PublicKey,
                          KDC Salt]Signed with KDC PrivateKey
```

At this point, the client validates the KDC signature, checks to see if the nonce is the same as the one in the AS_REQ, and stores the new KDC public key and public key version number. The client then generates a Diffie Hellman private parameter and computes the corresponding Diffie Hellman public parameter; the client also computes the shared Diffie Hellman symmetric key using the KDC Diffie Hellman public parameter and its own Diffie Hellman private parameter. Next, the client prompts the user for his/her password (if it does not already have the password). The password is concatenated with the KDC Salt and then SHA1 hashed; the

result is fed into the string to key function to obtain the new user DES key.

The new user DES key will be encrypted (along with the AS_REQ nonce) using the Diffie Hellman symmetric key and sent to the KDC in the new AS_REQ message:

```
Client -> AS_REQ with preauth: rootCert, [PKAuthenticator with
          user DH public parameter, {newUser DES key, nonce}DH
          symmetric key, KDC DH ID]Signed with User PrivateKey
                                                    -> KDC
```

The KDC DH ID is copied by the client from the KDC_ERROR message received above. Upon receipt and validation of this message, the KDC first uses the KDC DH ID as an index to locate its private Diffie Hellman parameter; it uses this parameter in combination with the user public Diffie Hellman parameter to compute the symmetric Diffie Hellman key. The KDC checks if the encrypted nonce is the same as the one in the PKAuthenticator and the AS_REQ part. The KDC then enters the new user DES key into the database, resets the recovery needed bit, and sets the valid symmetric key in database bit. The KDC then creates the AS_REP message:

```
Client <-- AS_REP with preauth: kdcCert, {encReplyKey,
           nonce}DH symmetric key <--------------------- KDC
```

The AS_REP encrypted part is encrypted with the encReplyKey that is generated on the KDC. The nonces are copied from the client AS_REQ. The kdcCert is a sequence of certificates that have been certified by certifiers listed in the client rootCert field, unless a zero length rootCert field was sent. In the last case, the kdcCert will also have zero length.

3.2.2. Private key held by KDC

Implementation of the changes in this section is RECOMMENDED.

When the user's private key is not carried with the user, the user may encrypt the private key using conventional cryptography, and register the encrypted private key with the KDC. As described in the previous section, the SHA1 hash of the password concatenated with the kdcSalt is also stored in the KDC database if the user only uses new clients. We restrict users of this protocol to using new clients only. The reason for this restriction is that it is not secure to store both the user private key encrypted in the user's password and the user password on the KDC simultaneously.

There are several options for storing private keys. If the user stores their private key on a removable disk, it is less convenient since they need to always carry the disk around with them; in addition, the procedures for extracting the key may vary between different operating systems. Alternatively, the user can store a private key on the hard disks of systems that he/she uses; besides limiting the systems that the user can login from there is also a greater security risk to the private key. If smart card readers or slots are deployed in an organization, then the user can store his/her private key on a smart card. Finally, the user can store his/her private key encrypted in a password on the KDC. This last option is probably the most practical option currently; it is important that a good password policy be used.

When the user's private key is stored on the KDC, preauthentication is required. There are two cases depending on whether the requesting user principal needs to be recovered.

In order to obtain its private key, a user principal includes the padata type PA-PK-AS-REQ in the preauthentication data field of the AS_REQ message. The accompanying pa-data field is:

```
PA-PK-AS-REQ ::= SEQUENCE {
        algorithmId[0]       INTEGER,      -- Public Key Alg.
        encClientPubVal[1]   EncryptedData -- EncPaPkAsReqDH
                                           -- (encrypted with key
                                           -- K1)
}

EncPaPkAsReqDH ::= SEQUENCE {
        clientPubValue[0]    SubjectPublicKeyInfo
}
```

Pictorially, PA-PK-AS-REQ is algorithmID, {clientPubValue}K1.

The user principal sends its Diffie-Hellman public value encrypted in the key K1. The key K1 is derived by performing string to key on the SHA1 hash of the user password concatenated with the kdcSalt which is stored in the /krb5/salt file. If the file is absent, the concatenation step is skipped in the above algorithm. The Diffie Hellman parameters g and p are implied by the algorithmID field. By choosing g and p correctly, dictionary attacks against the key K1 can be made more difficult [Jaspan].

If the requesting user principal needs recovery, the encrypted user private key is stored in the KDC database, and the AS_REQ RecoveryData field is not present in the PKAuthenticator, then the KDC replies with a KRB_ERROR message, with msg-type set to

KDC_ERR_PREAUTH_REQUIRED, and e-data set to:

```
PA-PK-AS-INFO ::= SEQUENCE {
        signedDHErr         SignedDHError,          -- signed by KDC
        encUserKey          OCTET STRING OPTIONAL   -- encrypted by
                                                    -- user password
                                                    -- key; (recovery
                                                    -- response)

}
```

The user principal should then continue with the section 3.2.1.1 protocol using the Diffie Hellman algorithm.

We now assume that the requesting user principal does not need recovery.

Upon receipt of the authentication request with the PA-PK-AS-REQ, the KDC generates the AS response as defined in RFC 1510, but additionally includes a preauthentication field of type PA-PK-USER-KEY.

```
PA-PK-USER-KEY ::= SEQUENCE {
        kdcCert             SEQUENCE OF Certificate,
        encUserKeyPart      EncryptedData, -- EncPaPkUserKeyPart
        kdcPrivKey          KDCPrivKey,
        kdcPrivKeySig       Signature
}
```

The kdc-cert field is identical to that in the PA-PK-AS-REP preauthentication data field returned with the KDC response, and must be validated as belonging to the KDC in the same manner.

```
KDCPrivKey ::= SEQUENCE {
        nonce               INTEGER,                -- From AS_REQ
        algorithmId         INTEGER,                -- DH algorithm
        kdcPubValue         SubjectPublicKeyInfo,   -- DH algorithm
        kdcSalt             OCTET STRING            -- Since user
                                                    -- uses only new
                                                    -- clients
}
```

The KDCPrivKey field is signed using the KDC private key.
The encrypted part of the AS_REP message is encrypted using the Diffie Hellman derived symmetric key, as is the EncPaPkUserKeyPart.

```
EncPaPkUserKeyPart ::= SEQUENCE {
        encUserKey          OCTET STRING,
        nonce               INTEGER                 -- From AS_REQ
}
```

Notationally, if encryption algorithm A is used, then enc-key-part is

```
    A ({encUserKey, nonce}, Diffie-Hellman-symmetric-key).
```

If the client has used an incorrect kdcSalt to compute the key K1, then the client needs to resubmit the above AS_REQ message using the correct kdcSalt field from the KDCPrivKey field.

This message contains the encrypted private key that has been registered with the KDC by the user, as encrypted by the user, super-encrypted with the Diffie Hellman derived symmetric key. Because there is direct trust between the user and the KDC, the transited field of the ticket returned by the KDC should remain empty. (Cf. Section 3.3.)

Examples

We now give several examples illustrating the protocols in this section.

Example 1: The requesting user principal needs to be recovered and stores his/her encrypted private key on the KDC. Then the exchange sequence between the user principal and the KDC is:

```
Client  --------> AS_REQ (with or without preauth) -----> KDC

Client  <--- KRB_ERROR (error code KDC_ERR_PREAUTH_REQUIRED)
             error data: [nonce, algID (DH), KDC DH public
                         parameter, KDC DH ID, KDC PublicKey
                         Kvno and PublicKey, KDC Salt]Signed
                         with KDC PrivateKey, {user private
                         key}user password <------------- KDC
```

The protocol now continues with the second AS_REQ as in Example 1 of section 3.2.1.1.

Example 2: The requesting user principal does not need to be recovered and stores his/her encrypted private key on the KDC. Then the exchange sequence between the user principal and the KDC when the user principal wants to obtain his/her private key is:

```
Client  -> AS_REQ with preauth: algID,
           {DH public parameter}K1 -> KDC
```

The key K1 is generated by using the string to key function on the SHA1 hash of the password concatenated with the kdcSalt from the /krb5/salt file. If the file is absent, then the concatenation step is skipped, and the client will learn

the correct kdcSalt in the following AS_REP message from the KDC. The algID should indicate some type of Diffie Hellman algorithm.

The KDC replies with the AS_REP message with a preauthentication data field:

```
Client <-- AS_REP with preauth: kdcCert, {encUserKey, <-- KDC
           nonce}DH symmetric key, [nonce, algID, DH
           public parameter, kdcSalt]KDC privateKey
```

The client validates the KDC's signature and checks that the nonce matches the nonce in its AS_REQ message. If the kdcSalt does not match what the client used, it starts the protocol over. The client then uses the KDC Diffie Hellman public parameter along with its own Diffie Hellman private parameter to compute the Diffie Hellman symmetric key. This key is used to decrypt the encUserKey field; the client checks if the nonce matches its AS_REQ nonce. At this point, the initial authentication protocol is complete.

Example 3: The requesting user principal does not need to be recovered and stores his/her encrypted private key on the KDC. In this example, the user principal uses the conventional symmetric key Kerberos V5 initial authentication protocol exchange.

We note that the conventional protocol exposes the user password to dictionary attacks; therefore, the user password must be changed more often. An example of when this protocol would be used is when new clients have been installed but an organization has not phased in public key authentication for all clients due to performance concerns.

```
Client ----> AS_REQ with preauthentication: {time}K1 --> KDC

Client <--------------------  AS_REP <------------------ KDC
```

The key K1 is derived as in the preceding two examples.

3.3. Clients with a public key certified by an outside authority

Implementation of the changes in this section is OPTIONAL.

In the case where the client is not registered with the current KDC, the client is responsible for obtaining the private key on its own. The client will request initial tickets from the KDC using the TGS exchange, but instead of performing

preauthentication using a Kerberos ticket granting ticket, or with the PA-PK-AS-REQ that is used when the public key is known to the KDC, the client performs preauthentication using the preauthentication data field of type PA-PK-AS-EXT-CERT:

```
PA-PK-AS-EXT-CERT ::= SEQUENCE {
        userCert[0]         SEQUENCE OF OCTET STRING,
        signedAuth[1]       SignedPKAuthenticator
}
```

where the user-cert specification depends on the type of certificate that the user possesses. In cases where the service has separate key pairs for digital signature and for encryption, we recommend that the signature keys be used for the purposes of sending the preauthentication (and deciphering the response).

The authenticator is the one used from the exchange in section 3.2.1, except that it is signed using the private key corresponding to the public key in the user-cert.

The KDC will verify the preauthentication authenticator, and check the certification path against its own policy of legitimate certifiers. This may be based on a certification hierarchy, or simply a list of recognized certifiers in a system like PGP.

If all checks out, the KDC will issue Kerberos credentials, as in 3.2, but with the names of all the certifiers in the certification path added to the transited field of the ticket, with a principal name taken from the certificate (this might be a long path for X.509, or a string like "John Q. Public <jqpublic@company.com>" if the certificate was a PGP certificate. The realm will identify the kind of certificate and the final certifier as follows:

```
    cert_type/final_certifier
```

as in PGP/<endorser@company.com>.

3.4. Digital Signature

Implementation of the changes in this section is OPTIONAL.

We offer this option with the warning that it requires the client process to generate a random DES key; this generation may not be able to guarantee the same level of randomness as the KDC.

If a user registered a digital signature key pair with the KDC, a separate exchange may be used. The client sends a KRB_AS_REQ as described in section 3.2.2. If the user's database record indicates that a digital signature key is to be used, then the

KDC sends back a KRB_ERROR as in section 3.2.2.

It is assumed here that the signature key is stored on local disk. The client generates a random key of enctype ENCTYPE_DES_CBC_CRC, signs it using the signature key (otherwise the signature is performed as described in section 3.2.1), then encrypts the whole with the public key of the KDC. This is returned with a separate KRB_AS_REQ in a preauthentication of type

```
PA-PK-AS-SIGNED ::= SEQUENCE {
        signedKey[0]        EncryptedData -- PaPkAsSignedData
}

PaPkAsSignedData ::= SEQUENCE {
        signedKeyPart[0]    SignedKeyPart,
        signedKeyAuth[1]    PKAuthenticator,
        sig[2]              Signature
}

SignedKeyPart ::= SEQUENCE {
        encSignedKey[0]     EncryptionKey,
        nonce[1]            INTEGER
}
```

where the nonce is the one from the request. Upon receipt of the request, the KDC decrypts, then verifies the random key. It then replies as per RFC 1510, except that instead of being encrypted with the password-derived DES key, the reply is encrypted using the randomKey sent by the client. Since the client already knows this key, there is no need to accompany the reply with an extra preauthentication field. Because there is direct trust between the user and the KDC, the transited field of the ticket returned by the KDC should remain empty. (Cf. Section 3.3.)

In the event that the KDC database indicates that the user principal must be recovered, and the PKAuthenticator does not contain the RecoveryData field, the KDC will reply with the KDC_RECOVERY_NEEDED error. The user principal then sends another AS_REQ message that includes the RecoveryData field in the PKAuthenticator. The AS_REP message is the same as in the basic Kerberos V5 protocol.

4. Preauthentication Data Types

We propose that the following preauthentication types be allocated for the preauthentication data packages described in this draft:

```
    #define KRB5_PADATA_ROOT_CERT     17  /* PA-PK-AS-ROOT */
    #define KRB5_PADATA_PUBLIC_REP    18  /* PA-PK-AS-REP */
```

```
#define KRB5_PADATA_PUBLIC_REQ    19  /* PA-PK-AS-REQ */
#define KRB5_PADATA_PRIVATE_REP   20  /* PA-PK-USER-KEY */
#define KRB5_PADATA_PUBLIC_EXT    21  /* PA-PK-AS-EXT-CERT */
#define KRB5_PADATA_PUBLIC_SIGN   22  /* PA-PK-AS-SIGNED */
```

5. Encryption Information

For the public key cryptography used in direct registration, we used (in our implementation) the RSAREF library supplied with the PGP 2.6.2 release. Encryption and decryption functions were implemented directly on top of the primitives made available therein, rather than the fully sealing operations in the API.

6. Compatibility with One-Time Passcodes

We solicit discussion on how the use of public key cryptography for initial authentication will interact with the proposed use of one time passwords discussed in Internet Draft <draft-ietf-cat-kerberos-passwords-00.txt>.

7. Strength of Encryption and Signature Mechanisms

In light of recent findings on the strengths of MD5 and various DES modes, we solicit discussion on which modes to incorporate into the protocol changes.

8. Expiration

This Internet-Draft expires on April 19, 1997.

9. Authors' Addresses

B. Clifford Neuman
USC/Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, CA 90292-6695

Phone: 310-822-1511
EMail: bcn@isi.edu

Brian Tung
USC/Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, CA 90292-6695

Phone: 310-822-1511

EMail: brian@isi.edu

John Wray
Digital Equipment Corporation
550 King Street, LKG2-2/Z7
Littleton, MA 01460

Phone: 508-486-5210
EMail: wray@tuxedo.enet.dec.com

Jonathan Trostle
CyberSafe Corporation
1605 NW Sammamish Rd., Suite 310
Issaquah, WA 98027-5378

Phone: 206-391-6000
EMail: jonathan.trostle@cybersafe.com