INTERNET-DRAFT
draft-ietf-cat-kerberos-pk-init-06.txt
Updates: RFC 1510
expires September 15, 1998

Brian Tung
Clifford Neuman
ISI
John Wray
Digital Equipment Corporation
Ari Medvinsky
Matthew Hur
CyberSafe Corporation
Jonathan Trostle
Novell

Public Key Cryptography for Initial Authentication in Kerberos

0. Status Of This Memo

1. Abstract

This document defines extensions (PKINIT) to the Kerberos protocol specification (RFC 1510 [1]) to provide a method for using public key cryptography during initial authentication. The methods defined specify the ways in which preauthentication data fields and error data fields in Kerberos messages are to be used to transport public key data.

2. Introduction

The popularity of public key cryptography has produced a desire for its support in Kerberos [2]. The advantages provided by public key cryptography include simplified key management (from the Kerberos perspective) and the ability to leverage existing and developing public key certification infrastructures.

Public key cryptography can be integrated into Kerberos in a number of ways. One is to associate a key pair with each realm, which can then be used to facilitate cross-realm authentication; this is the topic of another draft proposal. Another way is to allow users with public key certificates to use them in initial authentication. This is the concern of the current document.

One of the guiding principles in the design of PKINIT is that changes should be as minimal as possible. As a result, the basic mechanism of PKINIT is as follows: The user sends a request to the KDC as before, except that if that user is to use public key cryptography in the initial authentication step, his certificate accompanies the initial request, in the preauthentication fields.

Upon receipt of this request, the KDC verifies the certificate and issues a ticket granting ticket (TGT) as before, except that the encPart from the AS-REP message carrying the TGT is now encrypted in a randomly-generated key, instead of the user's long-term key (which is derived from a password). This random key is in turn encrypted using the public key from the certificate that came with the request and signed using the KDC's private key, and accompanies the reply, in the preauthentication fields.

PKINIT also allows for users with only digital signature keys to authenticate using those keys, and for users to store and retrieve private keys on the KDC.

The PKINIT specification may also be used for direct peer to peer authentication without contacting a central KDC. This application of PKINIT is described in PKTAPP [4] and is based on concepts introduced in [5, 6]. For direct client-to-server authentication, the client uses PKINIT to authenticate to the end server (instead of a central KDC), which then issues a ticket for itself. This approach has an advantage over SSL [7] in that the server does not need to save state (cache session keys). Furthermore, an additional benefit is that Kerberos tickets can facilitate delegation (see [8]).

3. Proposed Extensions

This section describes extensions to RFC 1510 for supporting the use of public key cryptography in the initial request for a ticket

granting ticket (TGT).

In summary, the following changes to RFC 1510 are proposed:

* Users may authenticate using either a public key pair or a conventional (symmetric) key. If public key cryptography is used, public key data is transported in preauthentication data fields to help establish identity.
* Users may store private keys on the KDC for retrieval during Kerberos initial authentication.

This proposal addresses two ways that users may use public key cryptography for initial authentication. Users may present public key certificates, or they may generate their own session key, signed by their digital signature key. In either case, the end result is that the user obtains an ordinary TGT that may be used for subsequent authentication, with such authentication using only conventional cryptography.

Section 3.1 provides definitions to help specify message formats. Section 3.2 and 3.3 describe the extensions for the two initial authentication methods. Section 3.4 describes a way for the user to store and retrieve his private key on the KDC, as an adjunct to the initial authentication.

3.1. Definitions

The extensions involve new encryption methods; we propose the addition of the following types:

dsa-sign	8
rsa-priv	9
rsa-pub	10
rsa-pub-md5	11
rsa-pub-sha1	12

The proposal of these encryption types notwithstanding, we do not mandate the use of any particular public key encryption method.

The extensions involve new preauthentication fields; we propose the addition of the following types:

PA-PK-AS-REQ	14
PA-PK-AS-REP	15
PA-PK-AS-SIGN	16
PA-PK-KEY-REQ	17
PA-PK-KEY-REP	18

The extensions also involve new error types; we propose the addition of the following types:

```
    KDC_ERR_CLIENT_NOT_TRUSTED              62
    KDC_ERR_KDC_NOT_TRUSTED                 63
    KDC_ERR_INVALID_SIG                     64
    KDC_ERR_KEY_TOO_WEAK                    65
    KDC_ERR_CERTIFICATE_MISMATCH            66
```

In addition, PKINIT does not define, but does permit, the following algorithm identifiers for use with the Signature data structure:

```
    md4WithRSAEncryption (as defined in PKCS 1)
    md5WithRSAEncryption (as defined in PKCS 1)
    sha-1WithRSAEncryption ::= { iso(1) member-body(2) us(840)
                                 rsadsi(113549) pkcs(1) pkcs-1(1) 5 }
    dsaWithSHA1 ::= { OIW oIWSecSig(3) oIWSecAlgorithm(2)
                      dsaWithSHA1(27) }
```

where

```
    OIW ::= iso(1) identifiedOrganization(3) oIW(14)
```

In many cases, PKINIT requires the encoding of an X.500 name as a Realm. In these cases, the realm will be represented using a different style, specified in RFC 1510 with the following example:

```
    NAMETYPE:rest/of.name=without-restrictions
```

For a realm derived from an X.500 name, NAMETYPE will have the value X500-RFC1779. The full realm name will appear as follows:

```
    X500-RFC1779:RFC1779Encode(DistinguishedName)
```

where DistinguishedName is an X.500 name, and RFC1779Encode is a readable ASCII encoding of an X.500 name, as defined by RFC 1779. To ensure that this encoding is unique, we add the following rules to those specified by RFC 1779:

* The optional spaces specified in RFC 1779 are not allowed.
* The character that separates relative distinguished names must be ',' (i.e., it must never be ';').
* Attribute values must not be enclosed in double quotes.
* Attribute values must not be specified as hexadecimal numbers.
* When an attribute name is specified in the form of an OID, it must start with the 'OID.' prefix, and not the 'oid.' prefix.
* The order in which the attributes appear in the RFC 1779 encoding must be the reverse of the order in the ASN.1 encoding of the X.500 name that appears in the public key certificate, because RFC 1779 requires that the least significant relative distinguished name appear first. The

order of the relative distinguished names, as well as the order of the attributes within each relative distinguished name, will be reversed.

Similarly, PKINIT may require the encoding of an X.500 name as a PrincipalName. In these cases, the name-type of the principal name shall be set to NT-X500-PRINCIPAL. This new name type is defined as:

```
#define CSFC5c_NT_X500_PRINCIPAL    6
```

The name-string shall be set as follows:

```
RFC1779Encode(DistinguishedName)
```

as described above.

3.1.1. Encryption and Key Formats

In the exposition below, we use the terms public key and private key generically. It should be understood that the term "public key" may be used to refer to either a public encryption key or a signature verification key, and that the term "private key" may be used to refer to either a private decryption key or a signature generation key. The fact that these are logically distinct does not preclude the assignment of bitwise identical keys.

All additional symmetric keys specified in this draft shall use the same encryption type as the session key in the response from the KDC. These include the temporary keys used to encrypt the signed random key encrypting the response, as well as the key derived from Diffie-Hellman agreement. In the case of Diffie-Hellman, the key shall be produced from the agreed bit string as follows:

* Truncate the bit string to the appropriate length.
* Rectify parity in each byte (if necessary) to obtain the key.

For instance, in the case of a DES key, we take the first eight bytes of the bit stream, and then adjust the least significant bit of each byte to ensure that each byte has odd parity.

RFC 1510, Section 6.1, defines EncryptedData as follows:

```
EncryptedData ::= SEQUENCE {
    etype               [0] INTEGER,
    kvno                [1] INTEGER OPTIONAL,
    cipher              [2] OCTET STRING
                            -- is CipherText
}
```

RFC 1510 also defines how CipherText is to be composed. It is not an ASN.1 data structure, but rather an octet string which is the encryption of a plaintext string. This plaintext string is in turn a concatenation of the following octet strings: a confounder, a checksum, the message, and padding. Details of how these components are arranged can be found in RFC 1510.

The PKINIT protocol introduces several new types of encryption. Data that is encrypted with a public key will allow only the check optional field, as it is defined above. This type of the checksum will be specified in the etype field, together with the encryption type.

In order to identify the checksum type, etype will have the following values:

```
            rsa-pub-MD5
            rsa-pub-SHA1
```

In the case that etype is set to rsa-pub, the optional 'check' field will not be provided.

Data that is encrypted with a private key will not use any optional fields. PKINIT uses private key encryption only for signatures, which are encrypted checksums. Therefore, the optional check field is not needed.

3.2. Standard Public Key Authentication

Implementation of the changes in this section is REQUIRED for compliance with PKINIT.

It is assumed that all public keys are signed by some certification authority (CA). The initial authentication request is sent as per RFC 1510, except that a preauthentication field containing data signed by the user's private key accompanies the request:

```
PA-PK-AS-REQ ::= SEQUENCE {
                                -- PA TYPE 14
    signedAuthPack          [0] SignedAuthPack
    userCert                [1] SEQUENCE OF Certificate OPTIONAL,
                                -- the user's certificate chain
    trustedCertifiers       [2] SEQUENCE OF PrincipalName OPTIONAL,
                                -- CAs that the client trusts
    serialNumber            [3] CertificateSerialNumber OPTIONAL
                                -- specifying a particular
                                -- certificate if the client
                                -- already has it;
                                -- must be accompanied by
                                -- a single trustedCertifier
```

```
}

CertificateSerialNumber ::= INTEGER
                            -- as specified by PKCS 6

SignedAuthPack ::= SEQUENCE {
    authPack                [0] AuthPack,
    authPackSig             [1] Signature,
                                -- of authPack
                                -- using user's private key
}

AuthPack ::= SEQUENCE {
    pkAuthenticator         [0] PKAuthenticator,
    clientPublicValue       [1] SubjectPublicKeyInfo OPTIONAL
                                -- if client is using Diffie-Hellman
}

PKAuthenticator ::= SEQUENCE {
    kdcName                 [0] PrincipalName,
    kdcRealm                [1] Realm,
    cusec                   [2] INTEGER,
                                -- for replay prevention
    ctime                   [3] KerberosTime,
                                -- for replay prevention
    nonce                   [4] INTEGER
}

Signature ::= SEQUENCE {
    signatureAlgorithm      [0] SignatureAlgorithmIdentifier,
    pkcsSignature           [1] BIT STRING
                                -- octet-aligned big-endian bit
                                -- string (encrypted with signer's
                                -- private key)
}

SignatureAlgorithmIdentifier ::= AlgorithmIdentifier

AlgorithmIdentifier ::= SEQUENCE {
    algorithm                   ALGORITHM.&id,
                                -- for DH, equals
                                -- dhKeyAgreement
                                -- ({iso(1) member-body(2) US(840)
                                -- rsadsi(113549) pkcs(1) pkcs-3(3)
                                -- 1})
    parameters                  ALGORITHM.&type
                                -- for DH, is DHParameter
}   -- as specified by the X.509 recommendation [9]

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm                   AlgorithmIdentifier,
```

```
    subjectPublicKey            BIT STRING
                                -- for DH, equals
                                -- public exponent (INTEGER encoded
                                -- as payload of BIT STRING)
}   -- as specified by the X.509 recommendation [9]

DHParameter ::= SEQUENCE {
    prime                       INTEGER,
                                -- p
    base                        INTEGER,
                                -- g
    privateValueLength          INTEGER OPTIONAL
}   -- as specified by the X.509 recommendation [9]

Certificate ::= SEQUENCE {
    certType                [0] INTEGER,
                                -- type of certificate
                                -- 1 = X.509v3 (DER encoding)
                                -- 2 = PGP (per PGP specification)
    certData                [1] OCTET STRING
                                -- actual certificate
                                -- type determined by certType
}
```

If the client passes a certificate serial number in the request, the KDC is requested to use the referred-to certificate. If none exists, then the KDC returns an error of type KDC_ERR_CERTIFICATE_MISMATCH. It also returns this error if, on the other hand, the client does not pass any trustedCertifiers, believing that it has the KDC's certificate, but the KDC has more than one certificate.

The PKAuthenticator carries information to foil replay attacks, to bind the request and response, and to optionally pass the client's Diffie-Hellman public value (i.e. for using DSA in combination with Diffie-Hellman). The PKAuthenticator is signed with the private key corresponding to the public key in the certificate found in userCert (or cached by the KDC).

In the PKAuthenticator, the client may specify the KDC name in one of two ways:

* The Kerberos principal name krbtgt/<realm_name>@<realm_name>, where <realm_name> is replaced by the applicable realm name.
* The name in the KDC's certificate (e.g., an X.500 name, or a PGP name).

Note that the first case requires that the certificate name and the Kerberos principal name be bound together (e.g., via an X.509v3 extension).

The userCert field is a sequence of certificates, the first of which must be the user's public key certificate. Any subsequent certificates will be certificates of the certifiers of the user's certificate. These cerificates may be used by the KDC to verify the user's public key. This field may be left empty if the KDC already has the user's certificate.

The trustedCertifiers field contains a list of certification authorities trusted by the client, in the case that the client does not possess the KDC's public key certificate. If the KDC has no certificate signed by any of the trustedCertifiers, then it returns an error of type KDC_ERR_CERTIFICATE_MISMATCH.

Upon receipt of the AS_REQ with PA-PK-AS-REQ pre-authentication type, the KDC attempts to verify the user's certificate chain (userCert), if one is provided in the request. This is done by verifying the certification path against the KDC's policy of legitimate certifiers. This may be based on a certification hierarchy, or it may be simply a list of recognized certifiers in a system like PGP.

If verification of the user's certificate fails, the KDC sends back an error message of type KDC_ERR_CLIENT_NOT_TRUSTED. The e-data field contains additional information pertaining to this error, and is formatted as follows:

```
    METHOD-DATA ::= SEQUENCE {
        method-type         [0] INTEGER,
                                -- 1 = cannot verify public key
                                -- 2 = invalid certificate
                                -- 3 = revoked certificate
                                -- 4 = invalid KDC name
                                -- 5 = client name mismatch
        method-data         [1] OCTET STRING OPTIONAL
    } -- syntax as for KRB_AP_ERR_METHOD (RFC 1510)
```

The values for the method-type and method-data fields are described in Section 3.2.1.

If trustedCertifiers is provided in the PA-PK-AS-REQ, the KDC verifies that it has a certificate issued by one of the certifiers trusted by the client. If it does not have a suitable certificate, the KDC returns an error message of type KDC_ERR_KDC_NOT_TRUSTED to the client.

If a trust relationship exists, the KDC then verifies the client's signature on AuthPack. If that fails, the KDC returns an error message of type KDC_ERR_INVALID_SIG. Otherwise, the KDC uses the timestamp in the PKAuthenticator to assure that the request is not a replay. The KDC also verifies that its name is specified in the PKAuthenticator.

If the clientPublicValue field is filled in, indicating that the client wishes to use Diffie-Hellman key agreement, then the KDC checks to see that the parameters satisfy its policy. If they do not (e.g., the prime size is insufficient for the expected encryption type), then the KDC sends back an error message of type KDC_ERR_KEY_TOO_WEAK. Otherwise, it generates its own public and private values for the response.

The KDC also checks that the timestamp in the PKAuthenticator is within the allowable window. If the local (server) time and the client time in the authenticator differ by more than the allowable clock skew, then the KDC returns an error message of type KRB_AP_ERR_SKEW.

Assuming no errors, the KDC replies as per RFC 1510, except as follows: The user's name in the ticket is as represented in the certificate, unless a Kerberos principal name is bound to the name in the certificate (e.g., via an X.509v3 extension). The user's realm in the ticket shall be the name of the Certification Authority that issued the user's public key certificate.

The KDC encrypts the reply not with the user's long-term key, but with a random key generated only for this particular response. This random key is sealed in the preauthentication field:

```
PA-PK-AS-REP ::= SEQUENCE {
                                -- PA TYPE 15
    encSignedReplyKeyPack   [0] EncryptedData,
                                -- of type SignedReplyKeyPack
                                -- using the temporary key
                                -- in encTmpKey
    encTmpKeyPack           [1] EncryptedData,
                                -- of type TmpKeyPack
                                -- using either the client public
                                -- key or the Diffie-Hellman key
                                -- specified by SignedDHPublicValue
    signedKDCPublicValue    [2] SignedKDCPublicValue OPTIONAL
                                -- if one was passed in the request
    kdcCert                 [3] SEQUENCE OF Certificate OPTIONAL,
                                -- the KDC's certificate chain
}

SignedReplyKeyPack ::= SEQUENCE {
    replyKeyPack            [0] ReplyKeyPack,
    replyKeyPackSig         [1] Signature,
                                -- of replyEncKeyPack
                                -- using KDC's private key
}

ReplyKeyPack ::= SEQUENCE {
```

```
    replyKey                [0] EncryptionKey,
                                -- used to encrypt main reply
                                -- of same ENCTYPE as session key
    nonce                   [1] INTEGER
                                -- binds response to the request
                                -- must be same as the nonce
                                -- passed in the PKAuthenticator
}

TmpKeyPack ::= SEQUENCE {
    tmpKey                  [0] EncryptionKey,
                                -- used to encrypt the
                                -- SignedReplyKeyPack
                                -- of same ENCTYPE as session key
}

SignedKDCPublicValue ::= SEQUENCE {
    kdcPublicValue          [0] SubjectPublicKeyInfo,
                                -- as described above
    kdcPublicValueSig       [1] Signature
                                -- of kdcPublicValue
                                -- using KDC's private key
}
```

The kdcCert field is a sequence of certificates, the first of which must be the KDC's public key certificate. Any subsequent certificates will be certificates of the certifiers of the KDC's certificate. The last of these must have as its certifier one of the certifiers sent to the KDC in the PA-PK-AS-REQ. These cerificates may be used by the client to verify the KDC's public key. This field is empty if the client did not send to the KDC a list of trusted certifiers (the trustedCertifiers field was empty).

Since each certifier in the certification path of a user's certificate is essentially a separate realm, the name of each certifier shall be added to the transited field of the ticket. The format of these realm names is defined in Section 3.1 of this document. If applicable, the transit-policy-checked flag should be set in the issued ticket.

The KDC's certificate must bind the public key to a name derivable from the name of the realm for that KDC. For an X.509 certificate, this is done as follows. The name of the KDC will be represented as an OtherName, encoded as a GeneralString:

```
    GeneralName ::= CHOICE {
        otherName       [0] KDCPrincipalName,
        ...
    }

    KDCPrincipalNameTypes OTHER-NAME ::= {
```

```
        { PrincipalNameSrvInst IDENTIFIED BY principalNameSrvInst }
    }

    KDCPrincipalName ::= SEQUENCE {
        nameType        [0] OTHER-NAME.&id ( { KDCPrincipalNameTypes } ),
        name            [1] OTHER-NAME.&type ( { KDCPrincipalNameTypes }
                            { @nameType } )
    }

    PrincipalNameSrvInst ::= GeneralString
```

where (from the Kerberos specification) we have

```
    krb5 OBJECT IDENTIFIER ::= { iso (1)
                                 org (3)
                                 dod (6)
                                 internet (1)
                                 security (5)
                                 kerberosv5 (2) }

    principalName OBJECT IDENTIFIER ::= { krb5 2 }

    principalNameSrvInst OBJECT IDENTIFIER ::= { principalName 2 }
```

The client then extracts the random key used to encrypt the main reply. This random key (in encPaReply) is encrypted with either the client's public key or with a key derived from the DH values exchanged between the client and the KDC.

3.2.1. Additional Information for Errors

This section describes the interpretation of the method-type and method-data fields of the KDC_ERR_CLIENT_NOT_TRUSTED error.

If method-type=1, the client's public key certificate chain does not contain a certificate that is signed by a certification authority trusted by the KDC. The format of the method-data field will be an ASN.1 encoding of a list of trusted certifiers, as defined above:

```
    TrustedCertifiers ::= SEQUENCE OF PrincipalName
```

If method-type=2, the signature on one of the certificates in the chain cannot be verified. The format of the method-data field will be an ASN.1 encoding of the integer index of the certificate in question:

```
    CertificateIndex ::= INTEGER
                         -- 0 = 1st certificate,
                         -- 1 = 2nd certificate, etc
```

If method-type=3, one of the certificates in the chain has been revoked. The format of the method-data field will be an ASN.1 encoding of the integer index of the certificate in question:

```
    CertificateIndex ::= INTEGER
                         -- 0 = 1st certificate,
                         -- 1 = 2nd certificate, etc
```

If method-type=4, the KDC name or realm in the PKAuthenticator does not match the principal name of the KDC. There is no method-data field in this case.

If method-type=5, the client name or realm in the certificate does not match the principal name of the client. There is no method-data field in this case.

3.3. Digital Signature

Implementation of the changes in this section are OPTIONAL for compliance with PKINIT.

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

If the user registered, or presents a certificate for, a digital signature key with the KDC instead of an encryption key, then a separate exchange must be used. The client sends a request for a TGT as usual, except that it (rather than the KDC) generates the random key that will be used to encrypt the KDC response. This key is sent to the KDC along with the request in a preauthentication field, encrypted with the KDC's public key:

```
PA-PK-AS-SIGN ::= SEQUENCE {
                             -- PA TYPE 16
    encSignedRandomKeyPack  [0] EncryptedData,
                                -- of type SignedRandomKeyPack
                                -- using the key in encTmpKeyPack
    encTmpKeyPack           [1] EncryptedData,
                                -- of type TmpKeyPack
                                -- using the KDC's public key
    userCert                [2] SEQUENCE OF Certificate OPTIONAL
                                -- the user's certificate chain
}

SignedRandomKeyPack ::= SEQUENCE {
    randomkeyPack           [0] RandomKeyPack,
    randomkeyPackSig        [1] Signature
                                -- of keyPack
                                -- using user's private key
```

```
}

RandomKeyPack ::= SEQUENCE {
    randomKey               [0] EncryptionKey,
                                -- will be used to encrypt reply
    randomKeyAuth           [1] PKAuthenticator
                                -- nonce copied from AS-REQ
}
```

If the KDC does not accept client-generated random keys as a matter of policy, then it sends back an error message of type KDC_ERR_KEY_TOO_WEAK. Otherwise, it extracts the random key as follows.

Upon receipt of the PA-PK-AS-SIGN, the KDC decrypts then verifies the randomKey. It then replies as per RFC 1510, except that the reply is encrypted not with a password-derived user key, but with the randomKey sent in the request. Since the client already knows this key, there is no need to accompany the reply with an extra preauthentication field. The transited field of the ticket should specify the certification path as described in Section 3.2.

3.4. Retrieving the User's Private Key from the KDC

Implementation of the changes described in this section are OPTIONAL for compliance with PKINIT.

When the user's private key is not stored local to the user, he may choose to store the private key (normally encrypted using a password-derived key) on the KDC. In this case, the client makes a request as described above, except that instead of preauthenticating with his private key, he uses a symmetric key shared with the KDC.

For simplicity's sake, this shared key is derived from the password-derived key used to encrypt the private key, in such a way that the KDC can authenticate the user with the shared key without being able to extract the private key.

We provide this option to present the user with an alternative to storing the private key on local disk at each machine where he expects to authenticate himself using PKINIT. It should be noted that it replaces the added risk of long-term storage of the private key on possibly many workstations with the added risk of storing the private key on the KDC in a form vulnerable to brute-force attack.

Denote by K1 the symmetric key used to encrypt the private key. Then construct symmetric key K2 as follows:

* Perform a hash on K1.
* Truncate the digest to Length(K1) bytes.

    * Rectify parity in each byte (if necessary) to obtain K2.

The KDC stores K2, the public key, and the encrypted private key. This key pair is designated as the "primary" key pair for that user. This primary key pair is the one used to perform initial authentication using the PA-PK-AS-REP preauthentication field. If he desires, he may also store additional key pairs on the KDC; these may be requested in addition to the primary. When the client requests initial authentication using public key cryptography, it must then include in its request, instead of a PA-PK-AS-REQ, the following preauthentication sequence:

```
PA-PK-KEY-REQ ::= SEQUENCE {
                                -- PA TYPE 17
    signedPKAuth            [0] SignedPKAuth,
    trustedCertifiers       [1] SEQUENCE OF PrincipalName OPTIONAL,
                                -- CAs that the client trusts
    keyIDList               [2] SEQUENCE OF Checksum OPTIONAL
                                -- payload is hash of public key
                                -- corresponding to desired
                                -- private key
                                -- if absent, KDC will return all
                                -- stored private keys
}

Checksum ::= SEQUENCE {
    cksumtype               [0] INTEGER,
    checksum                [1] OCTET STRING
}   -- as specified by RFC 1510

SignedPKAuth ::= SEQUENCE {
    pkAuth                  [0] PKAuthenticator,
    pkAuthSig               [1] Signature
                                -- of pkAuth
                                -- using the symmetric key K2
}
```

If a keyIDList is present, the first identifier should indicate the primary private key. No public key certificate is required, since the KDC stores the public key along with the private key. If there is no keyIDList, all the user's private keys are returned.

Upon receipt, the KDC verifies the signature using K2. If the verification fails, the KDC sends back an error of type KDC_ERR_INVALID_SIG. If the signature verifies, but the requested keys are not found on the KDC, then the KDC sends back an error of type KDC_ERR_PREAUTH_FAILED. If all checks out, the KDC responds as described in Section 3.2, except that in addition, the KDC appends the following preauthentication sequence:

```
PA-PK-KEY-REP ::= SEQUENCE {
```

```
                                -- PA TYPE 18
    encKeyRep                   [0] EncryptedData
                                    -- of type EncKeyReply
                                    -- using the symmetric key K2
}

EncKeyReply ::= SEQUENCE {
    keyPackList                 [0] SEQUENCE OF KeyPack,
                                    -- the first KeyPair is
                                    -- the primary key pair
    nonce                       [1] INTEGER
                                    -- binds reply to request
                                    -- must be identical to the nonce
                                    -- sent in the SignedAuthPack
}

KeyPack ::= SEQUENCE {
    keyID                       [0] Checksum,
    encPrivKey                  [1] OCTET STRING
}
```

Upon receipt of the reply, the client extracts the encrypted private keys (and may store them, at the client's option). The primary private key, which must be the first private key in the keyPack SEQUENCE, is used to decrypt the random key in the PA-PK-AS-REP; this key in turn is used to decrypt the main reply as described in Section 3.2.

4. Logistics and Policy

This section describes a way to define the policy on the use of PKINIT for each principal and request.

The KDC is not required to contain a database record for users that use either the Standard Public Key Authentication or Public Key Authentication with a Digital Signature. However, if these users are registered with the KDC, it is recommended that the database record for these users be modified to include three additional flags in the attributes field.

The first flag, use_standard_pk_init, indicates that the user should authenticate using standard PKINIT as described in Section 3.2. The second flag, use_digital_signature, indicates that the user should authenticate using digital signature PKINIT as described in Section 3.3. The third flag, store_private_key, indicates that the user has stored his private key on the KDC and should retrieve it using the exchange described in Section 3.4.

If one of the preauthentication fields defined above is included in the request, then the KDC shall respond as described in Sections 3.2

through 3.4, ignoring the aforementioned database flags. If more than one of the preauthentication fields is present, the KDC shall respond with an error of type KDC_ERR_PREAUTH_FAILED.

In the event that none of the preauthentication fields defined above are included in the request, the KDC checks to see if any of the above flags are set. If the first flag is set, then it sends back an error of type KDC_ERR_PREAUTH_REQUIRED indicating that a preauthentication field of type PA-PK-AS-REQ must be included in the request.

Otherwise, if the first flag is clear, but the second flag is set, then the KDC sends back an error of type KDC_ERR_PREAUTH_REQUIRED indicating that a preauthentication field of type PA-PK-AS-SIGN must be included in the request.

Lastly, if the first two flags are clear, but the third flag is set, then the KDC sends back an error of type KDC_ERR_PREAUTH_REQUIRED indicating that a preauthentication field of type PA-PK-KEY-REQ must be included in the request.

5. Security Considerations

PKINIT raises a few security considerations, which we will address in this section.

First of all, PKINIT introduces a new trust model, where KDCs do not (necessarily) certify the identity of those for whom they issue tickets. PKINIT does allow KDCs to act as their own CAs, in order to simplify key management, but one of the additional benefits is to align Kerberos authentication with a global public key infrastructure. Anyone using PKINIT in this way must be aware of how the certification infrastructure they are linking to works.

Secondly, PKINIT also introduces the possibility of interactions between different cryptosystems, which may be of widely varying strengths. Many systems, for instance, allow the use of 512-bit public keys. Using such keys to wrap data encrypted under strong conventional cryptosystems, such as triple-DES, is inappropriate; it adds a weak link to a strong one at extra cost. Implementors and administrators should take care to avoid such wasteful and deceptive interactions.

5. Transport Issues

Certificate chains can potentially grow quite large and span several UDP packets; this in turn increases the probability that a Kerberos message involving PKINIT extensions will be broken in transit. In light of the possibility that the Kerberos specification will

require KDCs to accept requests using TCP as a transport mechanism, we make the same recommendation with respect to the PKINIT extensions as well.

6. Bibliography

[1] J. Kohl, C. Neuman. The Kerberos Network Authentication Service (V5). Request for Comments 1510.

[2] B.C. Neuman, Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks, IEEE Communications, 32(9):33-38. September 1994.

[3] A. Medvinsky, M. Hur. Addition of Kerberos Cipher Suites to Transport Layer Security (TLS). draft-ietf-tls-kerb-cipher-suites-00.txt

[4] A. Medvinsky, M. Hur, B. Clifford Neuman. Public Key Utilizing Tickets for Application Servers (PKTAPP). draft-ietf-cat-pktapp-00.txt

[5] M. Sirbu, J. Chuang. Distributed Authentication in Kerberos Using Public Key Cryptography. Symposium On Network and Distributed System Security, 1997.

[6] B. Cox, J.D. Tygar, M. Sirbu. NetBill Security and Transaction Protocol. In Proceedings of the USENIX Workshop on Electronic Commerce, July 1995.

[7] Alan O. Freier, Philip Karlton and Paul C. Kocher. The SSL Protocol, Version 3.0 - IETF Draft.

[8] B.C. Neuman, Proxy-Based Authorization and Accounting for Distributed Systems. In Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993.

[9] ITU-T (formerly CCITT) Information technology - Open Systems Interconnection - The Directory: Authentication Framework Recommendation X.509 ISO/IEC 9594-8

7. Acknowledgements

Sasha Medvinsky contributed several ideas to the protocol changes and specifications in this document. His additions have been most appreciated.

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 have also been drawn from the DASS system. These changes are by no means endorsed by these groups. This is an attempt to revive some of the goals of those groups, and this proposal approaches those goals primarily from the Kerberos perspective. Lastly, comments from groups working on similar ideas in DCE have been invaluable.

8. Expiration Date

This draft expires September 15, 1998.

9. Authors

Brian Tung
Clifford Neuman
USC Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey CA 90292-6695
Phone: +1 310 822 1511
E-mail: {brian, bcn}@isi.edu

John Wray
Digital Equipment Corporation
550 King Street, LKG2-2/Z7
Littleton, MA 01460
Phone: +1 508 486 5210
E-mail: wray@tuxedo.enet.dec.com

Ari Medvinsky
Matthew Hur
CyberSafe Corporation
1605 NW Sammamish Road Suite 310
Issaquah WA 98027-5378
Phone: +1 206 391 6000
E-mail: {ari.medvinsky, matt.hur}@cybersafe.com

Jonathan Trostle
Novell Corporation
Provo UT
E-mail: jtrostle@novell.com