Network Working Group
Internet Draft
Intended Status: Informational
Expires: February 27, 2017

M. Jenkins
National Security Agency
M. Peck
The MITRE Corporation
K. Burgin

August 26, 2016

AES Encryption with HMAC-SHA2 for Kerberos 5
draft-ietf-kitten-aes-cts-hmac-sha2-11

Abstract

This document specifies two encryption types and two corresponding checksum types for Kerberos 5. The new types use AES in CTS mode (CBC mode with ciphertext stealing) for confidentiality and HMAC with a SHA-2 hash for integrity.

Status of this Memo

Copyright and License Notice

Table of Contents

1. Introduction

This document defines two encryption types and two corresponding checksum types for Kerberos 5 using AES with 128-bit or 256-bit keys.

To avoid ciphertext expansion, we use a variation of the CBC-CS3 mode defined in [SP800-38A+], also referred to as ciphertext stealing or CTS mode. The new types conform to the framework specified in [RFC3961], but do not use the simplified profile, as the simplified profile is not compliant with modern cryptographic best practices such as calculating MACs over ciphertext rather than plaintext.

The encryption and checksum types defined in this document are intended to support environments that desire to use SHA-256 or SHA-384 (defined in [FIPS180]) as the hash algorithm. Differences between the encryption and checksum types defined in this document and the pre-existing Kerberos AES encryption and checksum types specified in [RFC3962] are:

* The pseudorandom function used by PBKDF2 is HMAC-SHA-256 or HMAC-SHA-384 (HMAC is defined in [RFC2104]).

* A key derivation function from [SP800-108] using the SHA-256 or SHA-384 hash algorithm is used to produce keys for encryption, integrity protection, and checksum operations.

* The HMAC is calculated over the cipherstate concatenated with the AES output, instead of being calculated over the confounder and plaintext. This allows the message receiver to verify the integrity of the message before decrypting the message.

* The HMAC algorithm uses the SHA-256 or SHA-384 hash algorithm for integrity protection and checksum operations.

2. Protocol Key Representation

The AES key space is dense, so we can use random or pseudorandom octet strings directly as keys. The byte representation for the key is described in [FIPS197], where the first bit of the bit string is the high bit of the first byte of the byte string (octet string).

3. Key Derivation Function

We use a key derivation function from Section 5.1 of [SP800-108] which uses the HMAC algorithm as the PRF.

```
function KDF-HMAC-SHA2(key, label, [context,] k):
   k-truncate(K1)
```

where the value of K1 is computed as below.

key: The source of entropy from which subsequent keys are derived (this is known as Ki in [SP800-108]).

label: An octet string describing the intended usage of the derived key.

context: This parameter is optional. An octet string containing the information related to the derived keying material. This specification does not dictate a specific format for the context field. The context field is only used by the pseudo-random function defined in section 5, where it is set to the pseudo-random function's octet-string input parameter. The content of the octet-string input parameter is defined by the application that uses it.

k: Length in bits of the key to be outputted, expressed in big-endian binary representation in 4 bytes (this is called L in [SP800-108]). Specifically, k=128 is represented as 0x00000080, 192 as 0x000000C0, 256 as 0x00000100, and 384 as 0x00000180.

When the encryption type is aes128-cts-hmac-sha256-128, k must be no greater than 256 bits. When the encryption type is aes256-cts-hmac-sha384-192, k must be no greater than 384 bits.

The k-truncate function is defined in [RFC3961], Section 5.1. It returns the 'k' leftmost bits of the bitstring input.

In all computations in this document, | indicates concatenation.

When the encryption type is aes128-cts-hmac-sha256-128, then K1 is computed as follows:

```
If the context parameter is not present:
K1 = HMAC-SHA-256(key, 0x00000001 | label | 0x00 | k)

If the context parameter is present:
K1 = HMAC-SHA-256(key, 0x00000001 | label | 0x00 | context | k)
```

When the encryption type is aes256-cts-hmac-sha384-192, then K1 is computed as follows:

```
If the context parameter is not present:
K1 = HMAC-SHA-384(key, 0x00000001 | label | 0x00 | k)

If the context parameter is present:
K1 = HMAC-SHA-384(key, 0x00000001 | label | 0x00 | context | k)
```

In the definitions of K1 above, ’0x00000001’ is the i parameter (the iteration counter) from Section 5.1 of [SP800-108].

4. Key Generation from Pass Phrases

As defined below, the string-to-key function uses PBKDF2 [RFC2898] and KDF-HMAC-SHA2 to derive the base-key from a passphrase and salt. The string-to-key parameter string is four octets indicating an unsigned number in big-endian order, consistent with [RFC3962], except that the default is decimal 32768 if the parameter is not specified.

To ensure that different long-term base-keys are used with different enctypes, we prepend the enctype name to the salt, separated by a null byte. The enctype-name is "aes128-cts-hmac-sha256-128" or "aes256-cts-hmac-sha384-192" (without the quotes).

The user’s long-term base-key is derived as follows:

```
iter_count = string-to-key parameter, default is decimal 32768
saltp = enctype-name | 0x00 | salt
tkey = random-to-key(PBKDF2(passphrase, saltp,
                            iter_count, keylength))
base-key = random-to-key(KDF-HMAC-SHA2(tkey, "kerberos",
                                        keylength))

where "kerberos" is the octet-string 0x6B65726265726F73.
```

where PBKDF2 is the function of that name from RFC 2898, the pseudorandom function used by PBKDF2 is HMAC-SHA-256 when the enctype is "aes128-cts-hmac-sha256-128" and HMAC-SHA-384 when the enctype is "aes256-cts-hmac-sha384-192", the value for keylength is the AES key length (128 or 256 bits), and the algorithm KDF-HMAC-SHA2 is defined in Section 3.

5. Kerberos Algorithm Protocol Parameters

The RFC 3961 cipher state that maintains cryptographic state across different encryption operations using the same key is used as the formal initialization vector (IV) input into CBC-CS3. The plaintext is prepended with a 16-octet random value generated by the message originator, known as a confounder.

The ciphertext is a concatenation of the output of AES in CBC-CS3 mode and the HMAC of the cipher state concatenated with the AES output. The HMAC is computed using either SHA-256 or SHA-384 depending on the encryption type. The output of HMAC-SHA-256 is truncated to 128 bits and the output of HMAC-SHA-384 is truncated to

192 bits. Sample test vectors are given in Appendix A.

Decryption is performed by removing the HMAC, verifying the HMAC against the cipher state concatenated with the ciphertext, and then decrypting the ciphertext if the HMAC is correct. Finally, the first 16 octets of the decryption output (the confounder) is discarded, and the remainder is returned as the plaintext decryption output.

The following parameters apply to the encryption types aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192.

protocol key format: as defined in Section 2.

specific key structure: three derived keys: { Kc, Ke, Ki }.

Kc: the checksum key, inputted into HMAC to provide the checksum mechanism defined in Section 6.

Ke: the encryption key, inputted into AES encryption and decryption as defined in "encryption function" and "decryption function" below.

Ki: the integrity key, inputted into HMAC to provide authenticated encryption as defined in "encryption function" and "decryption function" below.

required checksum mechanism: as defined in Section 6.

key-generation seed length: key size (128 or 256 bits).

string-to-key function: as defined in Section 4.

default string-to-key parameters: iteration count of decimal 32768.

random-to-key function: identity function.

key-derivation function: KDF-HMAC-SHA2 as defined in Section 3. The key usage number is expressed as four octets in big-endian order.

```
If the enctype is aes128-cts-hmac-sha256-128:
Kc = KDF-HMAC-SHA2(base-key, usage | 0x99, 128)
Ke = KDF-HMAC-SHA2(base-key, usage | 0xAA, 128)
Ki = KDF-HMAC-SHA2(base-key, usage | 0x55, 128)

If the enctype is aes256-cts-hmac-sha384-192:
Kc = KDF-HMAC-SHA2(base-key, usage | 0x99, 192)
Ke = KDF-HMAC-SHA2(base-key, usage | 0xAA, 256)
Ki = KDF-HMAC-SHA2(base-key, usage | 0x55, 192)
```

cipher state: a 128-bit CBC initialization vector derived from a previous (if any) ciphertext using the same encryption key, as specified below.

initial cipher state: all bits zero.

encryption function: as follows, where E() is AES encryption in CBC-CS3 mode, and h is the size of truncated HMAC (128 bits or 192 bits as described above).

```
N = random value of length 128 bits (the AES block size)
IV = cipher state
C = E(Ke, N | plaintext, IV)
H = HMAC(Ki, IV | C)
ciphertext = C | H[1..h]

Steps to compute the 128-bit cipher state:
 L = length of C in bits
 portion C into 128-bit blocks, placing any remainder
 of less than 128 bits into a final block
 if L == 128: cipher state = C
 else if L mod 128 > 0: cipher state = last full (128-bit)
                                       block of C (the
                                       next-to-last block)
 else if L mod 128 == 0: cipher state = next-to-last block
                                        of C
 (note that L will never be less than 128 because of the
 presence of N in the encryption input)
```

decryption function: as follows, where D() is AES decryption in CBC-CS3 mode, and h is the size of truncated HMAC.

```
(C, H) = ciphertext (Note: H is the last h bits of the ciphertext)
IV = cipher state
if H != HMAC(Ki, IV | C)[1..h]
    stop, report error
(N, P) = D(Ke, C, IV)
Note: N is set to the first block of the decryption output,
P is set to the rest of the output.
cipher state = same as described above in encryption function
```

pseudo-random function:

```
If the enctype is aes128-cts-hmac-sha256-128:
PRF = KDF-HMAC-SHA2(input-key, "prf", octet-string, 256)

If the enctype is aes256-cts-hmac-sha384-192:
PRF = KDF-HMAC-SHA2(input-key, "prf", octet-string, 384)
```

where "prf" is the octet-string 0x707266

6. Checksum Parameters

The following parameters apply to the checksum types hmac-sha256-128-aes128 and hmac-sha384-192-aes256, which are the associated checksums for aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192, respectively.

associated cryptosystem: aes128-cts-hmac-sha256-128 or aes256-cts-hmac-sha384-192 as appropriate.

get_mic: HMAC(Kc, message)[1..h].
where h is 128 bits for checksum type hmac-sha256-128-aes128
and 192 bits for checksum type hmac-sha384-192-aes256

verify_mic: get_mic and compare.

7. IANA Considerations

IANA is requested to assign:

Encryption type numbers for aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192 in the Kerberos Encryption Type Numbers registry.

Etype	Encryption type	Reference
TBD1	aes128-cts-hmac-sha256-128	[this document]
TBD2	aes256-cts-hmac-sha384-192	[this document]

Checksum type numbers for hmac-sha256-128-aes128 and hmac-sha384-192-aes256 in the Kerberos Checksum Type Numbers registry.

Sumtype	Checksum type	Size	Reference
TBD3	hmac-sha256-128-aes128	16	[this document]
TBD4	hmac-sha384-192-aes256	24	[this document]

8. Security Considerations

This specification requires implementations to generate random values. The use of inadequate pseudo-random number generators (PRNGs) can result in little or no security. The generation of quality random numbers is difficult. [RFC4086] offers random number

generation guidance.

This document specifies a mechanism for generating keys from passphrases or passwords. The use of PBKDF2, a salt, and a large iteration count adds some resistance to off-line dictionary attacks by passive eavesdroppers. Salting prevents rainbow table attacks, while large iteration counts slow password guess attempts. Nonetheless, computing power continues to rapidly improve, including the potential for use of graphics processing units (GPUs) in password guess attempts. It is important to choose strong passphrases. Use of Kerberos extensions that protect against off-line dictionary attacks should also be considered, as should the use of public key cryptography for initial Kerberos authentication [RFC4556] to eliminate the use of passwords or passphrases within the Kerberos protocol.

The NIST guidance in section 5.3 of [SP800-38A], requiring that CBC initialization vectors be unpredictable, is satisfied by the use of a random confounder as the first block of plaintext. The confounder fills the cryptographic role typically played by an initialization vector. This approach was chosen to align with other Kerberos cryptosystem approaches.

8.1. Random Values in Salt Strings

NIST guidance in Section 5.1 of [SP800-132] requires at least 128 bits of the salt to be randomly generated. The string-to-key function as defined in [RFC3961] requires the salt to be valid UTF-8 strings [RFC3629]. Not every 128-bit random string will be valid UTF-8, so a UTF-8 compatible encoding would be needed to encapsulate the random bits. However, using a salt containing a random portion may have the following issues with some implementations:

* Cross-realm krbtgt keys are typically managed by entering the same password at two KDCs to get the same keys. If each KDC uses a random salt, they won’t have the same keys.

* Random salts may interfere with password history checking.

8.2. Algorithm Rationale

This document has been written to be consistent with common implementations of AES and SHA-2. The encryption and hash algorithm sizes have been chosen to create a consistent level of protection, with consideration to implementation efficiencies. So, for instance, SHA-384, which would normally be matched to AES-192, is instead matched to AES-256 to leverage the fact that there are efficient hardware implementations of AES-256. Note that, as indicated by the

enc-type name "aes256-cts-hmac-sha384-192", the truncation of the HMAC-SHA-384 output to 192-bits results in an overall 192-bit level of security.

9. Acknowledgements

Kelley Burgin was employed at the National Security Agency during much of the work on this document.

10. References

10.1. Normative References

[RFC2104] Krawczyk, H. et al., "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 3629, November 2003.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[RFC3962] Raeburn, K., "Advanced Encryption Standard (AES) Encryption for Kerberos 5", RFC 3962, February 2005.

[FIPS180] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-4, August 2015.

[FIPS197] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

[SP800-38A+] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode", NIST Special Publication 800-38A Addendum, October 2010.

[SP800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, October 2009.

10.2. Informative References

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker,

"Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[SP800-38A] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, December 2001.

[SP800-132] National Institute of Standards and Technology, "Recommendation for Password-Based Key Derivation, Part 1: Storage Applications", NIST Special Publication 800-132, June 2010.

Appendix A. Test Vectors

```
Sample results for string-to-key conversion:
---------------------------------------------

Iteration count = 32768
Pass phrase = "password"
Saltp for creating 128-bit base-key:
   61 65 73 31 32 38 2D 63 74 73 2D 68 6D 61 63 2D
   73 68 61 32 35 36 2D 31 32 38 00 10 DF 9D D7 83
   E5 BC 8A CE A1 73 0E 74 35 5F 61 41 54 48 45 4E
   41 2E 4D 49 54 2E 45 44 55 72 61 65 62 75 72 6E

(The saltp is "aes128-cts-hmac-sha256-128" | 0x00 |
 random 16 byte valid UTF-8 sequence | "ATHENA.MIT.EDUraeburn")
128-bit base-key:
   08 9B CA 48 B1 05 EA 6E A7 7C A5 D2 F3 9D C5 E7

Saltp for creating 256-bit base-key:
   61 65 73 32 35 36 2D 63 74 73 2D 68 6D 61 63 2D
   73 68 61 33 38 34 2D 31 39 32 00 10 DF 9D D7 83
   E5 BC 8A CE A1 73 0E 74 35 5F 61 41 54 48 45 4E
   41 2E 4D 49 54 2E 45 44 55 72 61 65 62 75 72 6E
(The saltp is "aes256-cts-hmac-sha384-192" | 0x00 |
 random 16 byte valid UTF-8 sequence | "ATHENA.MIT.EDUraeburn")
256-bit base-key:
   45 BD 80 6D BF 6A 83 3A 9C FF C1 C9 45 89 A2 22
   36 7A 79 BC 21 C4 13 71 89 06 E9 F5 78 A7 84 67

Sample results for key derivation:
----------------------------------

enctype aes128-cts-hmac-sha256-128:
128-bit base-key:
   37 05 D9 60 80 C1 77 28 A0 E8 00 EA B6 E0 D2 3C
```

```
Kc value for key usage 2 (label = 0x0000000299):
   B3 1A 01 8A 48 F5 47 76 F4 03 E9 A3 96 32 5D C3
Ke value for key usage 2 (label = 0x00000002AA):
   9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
Ki value for key usage 2 (label = 0x0000000255):
   9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C

enctype aes256-cts-hmac-sha384-192:
256-bit base-key:
   6D 40 4D 37 FA F7 9F 9D F0 D3 35 68 D3 20 66 98
   00 EB 48 36 47 2E A8 A0 26 D1 6B 71 82 46 0C 52
Kc value for key usage 2 (label = 0x0000000299):
   EF 57 18 BE 86 CC 84 96 3D 8B BB 50 31 E9 F5 C4
   BA 41 F2 8F AF 69 E7 3D
Ke value for key usage 2 (label = 0x00000002AA):
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
Ki value for key usage 2 (label = 0x0000000255):
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F

Sample encryptions (all using the default cipher state):
--------------------------------------------------------
These sample encryptions use the above sample key
derivation results, including use of the same
base-key and key usage values.

The following test vectors are for
enctype aes128-cts-hmac-sha256-128:

Plaintext: (empty)
Confounder:
   7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
128-bit AES key (Ke):
   9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
128-bit HMAC key (Ki):
   9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C
AES Output:
   EF 85 FB 89 0B B8 47 2F 4D AB 20 39 4D CA 78 1D
Truncated HMAC Output:
   AD 87 7E DA 39 D5 0C 87 0C 0D 5A 0A 8E 48 C7 18
Ciphertext (AES Output | HMAC Output):
   EF 85 FB 89 0B B8 47 2F 4D AB 20 39 4D CA 78 1D
   AD 87 7E DA 39 D5 0C 87 0C 0D 5A 0A 8E 48 C7 18

Plaintext: (length less than block size)
   00 01 02 03 04 05
Confounder:
```

```
      7B CA 28 5E 2F D4 13 0F B5 5B 1A 5C 83 BC 5B 24
   128-bit AES key (Ke):
      9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
   128-bit HMAC key (Ki):
      9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C
   AES Output:
      84 D7 F3 07 54 ED 98 7B AB 0B F3 50 6B EB 09 CF
      B5 54 02 CE F7 E6
   Truncated HMAC Output:
      87 7C E9 9E 24 7E 52 D1 6E D4 42 1D FD F8 97 6C
   Ciphertext:
      84 D7 F3 07 54 ED 98 7B AB 0B F3 50 6B EB 09 CF
      B5 54 02 CE F7 E6 87 7C E9 9E 24 7E 52 D1 6E D4
      42 1D FD F8 97 6C

   Plaintext: (length equals block size)
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   Confounder:
      56 AB 21 71 3F F6 2C 0A 14 57 20 0F 6F A9 94 8F
   128-bit AES key (Ke):
      9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
   128-bit HMAC key (Ki):
      9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C
   AES Output:
      35 17 D6 40 F5 0D DC 8A D3 62 87 22 B3 56 9D 2A
      E0 74 93 FA 82 63 25 40 80 EA 65 C1 00 8E 8F C2
   Truncated HMAC Output:
      95 FB 48 52 E7 D8 3E 1E 7C 48 C3 7E EB E6 B0 D3
   Ciphertext:
      35 17 D6 40 F5 0D DC 8A D3 62 87 22 B3 56 9D 2A
      E0 74 93 FA 82 63 25 40 80 EA 65 C1 00 8E 8F C2
      95 FB 48 52 E7 D8 3E 1E 7C 48 C3 7E EB E6 B0 D3

   Plaintext: (length greater than block size)
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
      10 11 12 13 14
   Confounder:
      A7 A4 E2 9A 47 28 CE 10 66 4F B6 4E 49 AD 3F AC
   128-bit AES key (Ke):
      9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
   128-bit HMAC key (Ki):
      9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C
   AES Output:
      72 0F 73 B1 8D 98 59 CD 6C CB 43 46 11 5C D3 36
      C7 0F 58 ED C0 C4 43 7C 55 73 54 4C 31 C8 13 BC
      E1 E6 D0 72 C1
   Truncated HMAC Output:
      86 B3 9A 41 3C 2F 92 CA 9B 83 34 A2 87 FF CB FC
```

```
Ciphertext:
   72 0F 73 B1 8D 98 59 CD 6C CB 43 46 11 5C D3 36
   C7 0F 58 ED C0 C4 43 7C 55 73 54 4C 31 C8 13 BC
   E1 E6 D0 72 C1 86 B3 9A 41 3C 2F 92 CA 9B 83 34
   A2 87 FF CB FC
```

The following test vectors are for enctype aes256-cts-hmac-sha384-192:

```
Plaintext: (empty)
Confounder:
   F7 64 E9 FA 15 C2 76 47 8B 2C 7D 0C 4E 5F 58 E4
256-bit AES key (Ke):
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
192-bit HMAC key (Ki):
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F
AES Output:
   41 F5 3F A5 BF E7 02 6D 91 FA F9 BE 95 91 95 A0
Truncated HMAC Output:
   58 70 72 73 A9 6A 40 F0 A0 19 60 62 1A C6 12 74
   8B 9B BF BE 7E B4 CE 3C
Ciphertext:
   41 F5 3F A5 BF E7 02 6D 91 FA F9 BE 95 91 95 A0
   58 70 72 73 A9 6A 40 F0 A0 19 60 62 1A C6 12 74
   8B 9B BF BE 7E B4 CE 3C

Plaintext: (length less than block size)
   00 01 02 03 04 05
Confounder:
   B8 0D 32 51 C1 F6 47 14 94 25 6F FE 71 2D 0B 9A
256-bit AES key (Ke):
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
192-bit HMAC key (Ki):
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F
AES Output:
   4E D7 B3 7C 2B CA C8 F7 4F 23 C1 CF 07 E6 2B C7
   B7 5F B3 F6 37 B9
Truncated HMAC Output:
   F5 59 C7 F6 64 F6 9E AB 7B 60 92 23 75 26 EA 0D
   1F 61 CB 20 D6 9D 10 F2
Ciphertext:
   4E D7 B3 7C 2B CA C8 F7 4F 23 C1 CF 07 E6 2B C7
   B7 5F B3 F6 37 B9 F5 59 C7 F6 64 F6 9E AB 7B 60
   92 23 75 26 EA 0D 1F 61 CB 20 D6 9D 10 F2
```

```
Plaintext: (length equals block size)
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Confounder:
   53 BF 8A 0D 10 52 65 D4 E2 76 42 86 24 CE 5E 63
256-bit AES key (Ke):
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
192-bit HMAC key (Ki):
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F
AES Output:
   BC 47 FF EC 79 98 EB 91 E8 11 5C F8 D1 9D AC 4B
   BB E2 E1 63 E8 7D D3 7F 49 BE CA 92 02 77 64 F6
Truncated HMAC Output:
   8C F5 1F 14 D7 98 C2 27 3F 35 DF 57 4D 1F 93 2E
   40 C4 FF 25 5B 36 A2 66
Ciphertext:
   BC 47 FF EC 79 98 EB 91 E8 11 5C F8 D1 9D AC 4B
   BB E2 E1 63 E8 7D D3 7F 49 BE CA 92 02 77 64 F6
   8C F5 1F 14 D7 98 C2 27 3F 35 DF 57 4D 1F 93 2E
   40 C4 FF 25 5B 36 A2 66

Plaintext: (length greater than block size)
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14
Confounder:
   76 3E 65 36 7E 86 4F 02 F5 51 53 C7 E3 B5 8A F1
256-bit AES key (Ke):
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
192-bit HMAC key (Ki):
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F
AES Output:
   40 01 3E 2D F5 8E 87 51 95 7D 28 78 BC D2 D6 FE
   10 1C CF D5 56 CB 1E AE 79 DB 3C 3E E8 64 29 F2
   B2 A6 02 AC 86
Truncated HMAC Output:
   FE F6 EC B6 47 D6 29 5F AE 07 7A 1F EB 51 75 08
   D2 C1 6B 41 92 E0 1F 62
Ciphertext:
   40 01 3E 2D F5 8E 87 51 95 7D 28 78 BC D2 D6 FE
   10 1C CF D5 56 CB 1E AE 79 DB 3C 3E E8 64 29 F2
   B2 A6 02 AC 86 FE F6 EC B6 47 D6 29 5F AE 07 7A
   1F EB 51 75 08 D2 C1 6B 41 92 E0 1F 62
```

Sample checksums:

These sample checksums use the above sample key derivation results, including use of the same base-key and key usage values.

```
Checksum type: hmac-sha256-128-aes128
128-bit HMAC key (Kc):
   B3 1A 01 8A 48 F5 47 76 F4 03 E9 A3 96 32 5D C3
Plaintext:
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14
Checksum:
   D7 83 67 18 66 43 D6 7B 41 1C BA 91 39 FC 1D EE

Checksum type: hmac-sha384-192-aes256
192-bit HMAC key (Kc):
   EF 57 18 BE 86 CC 84 96 3D 8B BB 50 31 E9 F5 C4
   BA 41 F2 8F AF 69 E7 3D
Plaintext:
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14
Checksum:
   45 EE 79 15 67 EE FC A3 7F 4A C1 E0 22 2D E8 0D
   43 C3 BF A0 66 99 67 2A
```

```
Sample pseudorandom function (PRF) invocations:
----------------------------------------

PRF input octet-string: "test" (0x74657374)

enctype aes128-cts-hmac-sha256-128:
input-key value / HMAC-SHA-256 key:
   37 05 D9 60 80 C1 77 28 A0 E8 00 EA B6 E0 D2 3C
HMAC-SHA-256 input message:
   00 00 00 01 70 72 66 00 74 65 73 74 00 00 01 00
PRF output:
   9D 18 86 16 F6 38 52 FE 86 91 5B B8 40 B4 A8 86
   FF 3E 6B B0 F8 19 B4 9B 89 33 93 D3 93 85 42 95

enctype aes256-cts-hmac-sha384-192:
input-key value / HMAC-SHA-384 key:
   6D 40 4D 37 FA F7 9F 9D F0 D3 35 68 D3 20 66 98
   00 EB 48 36 47 2E A8 A0 26 D1 6B 71 82 46 0C 52
HMAC-SHA-384 input message:
   00 00 00 01 70 72 66 00 74 65 73 74 00 00 01 80
PRF output:
   98 01 F6 9A 36 8C 2B F6 75 E5 95 21 E1 77 D9 A0
   7F 67 EF E1 CF DE 8D 3C 8D 6F 6A 02 56 E3 B1 7D
   B3 C1 B6 2A D1 B8 55 33 60 D1 73 67 EB 15 14 D2
```

Authors' Addresses

Michael J. Jenkins
National Security Agency

EMail: mjjenki@tycho.ncsc.mil

Michael A. Peck
The MITRE Corporation

EMail: mpeck@mitre.org

Kelley W. Burgin

Email: kelley.burgin@gmail.com

Internet Engineering Task Force
Internet-Draft
Updates: 4120 (if approved)
Intended status: Standards Track
Expires: May 4, 2016

S. Sorce
Red Hat
T. Yu
MIT

November 1, 2015

Kerberos Authorization Data Container Authenticated by Multiple MACs
draft-ietf-kitten-cammac-04

Abstract

This document specifies a Kerberos Authorization Data container that supersedes AD-KDC-ISSUED. It allows for multiple Message Authentication Codes (MACs) or signatures to authenticate the contained Authorization Data elements. The multiple MACs are needed to mitigate shortcomings in the existing AD-KDC-ISSUED container. This document updates RFC 4120.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document specifies a new Authorization Data container for Kerberos, called the CAMMAC (Container Authenticated by Multiple MACs). The ASN.1 type implementing the CAMMAC concept is the AD-CAMMAC, which supersedes the AD-KDC-ISSUED Authorization Data type specified in [RFC4120]. This new container allows both the receiving application service and the Key Distribution Center (KDC) itself to verify the authenticity of the contained authorization data. The AD-CAMMAC container can also include additional verifiers that "trusted services" can use to verify the contained authorization data.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Motivations

The Kerberos protocol allows clients to submit arbitrary authorization data for a KDC to insert into a Kerberos ticket. These client-requested authorization data allow the client to express authorization restrictions that the application service will interpret. With few exceptions, the KDC can safely copy these client-requested authorization data to the issued ticket without necessarily inspecting, interpreting, or filtering their contents.

The AD-KDC-ISSUED authorization data container specified in RFC 4120 [RFC4120] is a means for KDCs to include positive or permissive (rather than restrictive) authorization data in service tickets in a way that the service named in a ticket can verify that the KDC has issued the contained authorization data. This capability takes advantage of a shared symmetric key between the KDC and the service to assure the service that the KDC did not merely copy client-requested authorization data to the ticket without inspecting them.

The AD-KDC-ISSUED container works well for situations where the flow of authorization data is from the KDC to the service. However, protocol extensions such as Constrained Delegation (S4U2Proxy [MS-SFU]) require that a service present to the KDC a service ticket that the KDC previously issued, as evidence that the service is authorized to impersonate the client principal named in that ticket. In the S4U2Proxy extension, the KDC uses the evidence ticket as the basis for issuing a derivative ticket that the service can then use to impersonate the client. The authorization data contained within the evidence ticket constitute a flow of authorization data from the application service to the KDC. The properties of the AD-KDC-ISSUED container are insufficient for this use case because the service knows the symmetric key for the checksum in the AD-KDC-ISSUED container. Therefore, the KDC has no way to detect whether the service has tampered with the contents of the AD-KDC-ISSUED container within the evidence ticket.

The new AD-CAMMAC authorization data container specified in this document improves upon AD-KDC-ISSUED by including additional verifier elements. The svc-verifier (service verifier) element of the AD-CAMMAC has the same functional and security properties as the ad-checksum element of AD-KDC-ISSUED; the svc-verifier allows the service to verify the integrity of the AD-CAMMAC contents as it already could with the AD-KDC-ISSUED container. The kdc-verifier and other-verifiers elements are new to AD-CAMMAC and provide its enhanced capabilities.

The kdc-verifier element of the AD-CAMMAC container allows a KDC to verify the integrity of authorization data that it previously inserted into a ticket, by using a key that only the KDC knows. The KDC thus avoids recomputing all of the authorization data for the issued ticket; this recomputation might not always be possible when that data includes ephemeral information such as the strength or type of authentication method used to obtain the original ticket.

The verifiers in the other-verifiers element of the AD-CAMMAC container are not required, but can be useful when a lesser-privileged service receives a ticket from a client and needs to extract the AD-CAMMAC to demonstrate to a higher-privileged "trusted

service" on the same host that it is legitimately acting on behalf of that client. The trusted service can use a verifier in the other-verifiers element to validate the contents of the AD-CAMMAC without further communication with the KDC.

4. Encoding

The Kerberos protocol is defined in [RFC4120] using Abstract Syntax Notation One (ASN.1) [X.680] and using the ASN.1 Distinguished Encoding Rules (DER) [X.690]. For consistency, this specification also uses ASN.1 for specifying the layout of AD-CAMMAC. The ad-data of the AD-CAMMAC authorization data element is the ASN.1 DER encoding of the AD-CAMMAC ASN.1 type specified below.

```
KerberosV5CAMMAC {
        iso(1) identified-organization(3) dod(6) internet(1)
        security(5) kerberosV5(2) modules(4) cammac(7)
} DEFINITIONS EXPLICIT TAGS ::= BEGIN

IMPORTS
      AuthorizationData, PrincipalName, Checksum, UInt32, Int32
        FROM KerberosV5Spec2 { iso(1) identified-organization(3)
          dod(6) internet(1) security(5) kerberosV5(2)
          modules(4) krb5spec2(2) };
          -- as defined in RFC 4120.

AD-CAMMAC                   ::= SEQUENCE {
      elements              [0] AuthorizationData,
      kdc-verifier          [1] Verifier-MAC OPTIONAL,
      svc-verifier          [2] Verifier-MAC OPTIONAL,
      other-verifiers       [3] SEQUENCE (SIZE (1..MAX))
                                OF Verifier OPTIONAL
}

Verifier              ::= CHOICE {
      mac             Verifier-MAC,
      ...
}

Verifier-MAC          ::= SEQUENCE {
      identifier      [0] PrincipalName OPTIONAL,
      kvno            [1] UInt32 OPTIONAL,
      enctype         [2] Int32 OPTIONAL,
      mac             [3] Checksum
}

END
```

elements:
A sequence of authorization data elements issued by the KDC. These elements are the authorization data that the verifier fields authenticate.

Verifier:
A CHOICE type that currently contains only one alternative: Verifier-MAC. Future extensions might add support for public-key signatures.

Verifier-MAC:
Contains an RFC 3961 [RFC3961] Checksum (MAC) computed over the ASN.1 DER encoding of the AuthorizationData value in the elements field of the AD-CAMMAC. The identifier, kvno, and enctype fields help the recipient locate the key required for verifying the MAC. For the kdc-verifier and the svc-verifier, the identifier, kvno and enctype fields are often obvious from context and MAY be omitted. For the kdc-verifier, the MAC is computed differently than for the svc-verifier and the other-verifiers, as described later. The key usage number for computing the MAC (Checksum) is 64.

kdc-verifier:
A Verifier-MAC where the key is a long-term key of the local Ticket-Granting Service (TGS). The checksum type is the required checksum type for the enctype of the TGS key. In contrast to the other Verifier-MAC elements, the KDC computes the MAC in the kdc-verifier over the ASN.1 DER encoding of the EncTicketPart of the surrounding ticket, but where the AuthorizationData value in the EncTicketPart contains the AuthorizationData value contained in the AD-CAMMAC instead of the AuthorizationData value that would otherwise be present in the ticket. This altered Verifier-MAC computation binds the kdc-verifier to the other contents of the ticket, assuring the KDC that a malicious service has not substituted a mismatched AD-CAMMAC received from another ticket.

svc-verifier:
A Verifier-MAC where the key is the same long-term service key that the KDC uses to encrypt the surrounding ticket. The checksum type is the required checksum type for the enctype of the service key used to encrypt the ticket. This field MUST be present if the service principal of the ticket is not the local TGS, including when the ticket is a cross-realm Ticket-Granting Ticket (TGT).

other-verifiers:
A sequence of additional verifiers. In each additional Verifier-MAC, the key is a long-term key of the principal name specified in the identifier field. The PrincipalName MUST be present and be a

valid principal in the realm. KDCs MAY add one or more "trusted service" verifiers. Unless otherwise administratively configured, the KDC SHOULD determine the "trusted service" principal name by replacing the service identifier component of the sname of the surrounding ticket with "host". The checksum is computed using a long-term key of the identified principal, and the checksum type is the required checksum type for the enctype of that long-term key. The kvno and enctype SHOULD be specified to disambiguate which of the long-term keys of the trusted service is used.

5. Usage

Application servers and KDCs MAY ignore the AD-CAMMAC container and the authorization data elements it contains. For compatibility with older Kerberos implementations, a KDC issuing an AD-CAMMAC SHOULD enclose it in an AD-IF-RELEVANT container [RFC4120] unless the KDC knows that the application server is likely to recognize it.

6. Assigned numbers

RFC 4120 is updated in the following ways:

- o The ad-type number 96 is assigned for AD-CAMMAC, updating the table in Section 7.5.4 of [RFC4120].

- o The table in Section 5.2.6 of [RFC4120] is updated to map the ad-type 96 to "DER encoding of AD-CAMMAC".

- o The key usage number 64 is assigned for the Verifier-MAC checksum, updating the table in Section 7.5.1 of [RFC4120].

7. IANA Considerations

[RFC Editor: please remove this section prior to publication.]

There are no IANA considerations in this document. Any numbers assigned in this document are not in IANA-controlled number spaces.

8. Security Considerations

The CAMMAC provides data origin authentication for authorization data contained in it, attesting that it originated from the KDC. This section describes the precautions required to maintain the integrity of that data origin authentication through various information flows involving a Kerberos ticket containing a CAMMAC.

When handling a TGS-REQ containing a CAMMAC, a KDC makes a policy decision on how to produce the CAMMAC contents of the newly issued

ticket based on properties of the ticket(s) accompanying the TGS-REQ. This policy decision can involve filtering, transforming, or verbatim copying of the original CAMMAC contents. The following paragraphs provide some guidance on formulating such policies.

A KDC verifies a CAMMAC as originating from a local realm KDC when at least one of following the criteria is true:

1. The KDC successfully verifies the kdc-verifier; or

2. The KDC successfully verifies the svc-verifier, and the svc-verifier uses a key known only to the local realm KDCs; or

3. No verifiers are present, the ticket-encrypting key is known only to local realm KDCs, and all local realm KDCs properly filter out client-submitted CAMMACs. (This can require particular caution in a realm that has KDCs with mixed CAMMAC support, as might happen when incrementally upgrading KDCs in a realm to support CAMMAC.)

A CAMMAC that originates from a local realm KDC might contain information that originates from elsewhere. Originating from a local realm KDC means that a local realm KDC attests that the CAMMAC contents conform to the local realm’s policy, regardless of the ultimate origin of the information in the CAMMAC (which could be a remote realm in the case of a CAMMAC contained in a cross-realm TGT).

Local policy determines when a KDC can apply a kdc-verifier to a CAMMAC (or otherwise creates a CAMMAC that satisfies the local origin criteria listed above). Semantically, a CAMMAC that a KDC verifies as originating from a local realm KDC attests that the CAMMAC contents conformed to local policy at the time of creation of the CAMMAC. Such a local policy can include allowing verbatim copying of CAMMAC contents from cross-realm TGTs from designated remote realms and applying a kdc-verifier to the new CAMMAC.

Usually, when a KDC verifies a CAMMAC as originating from a local realm KDC, the KDC can assume that the CAMMAC contents continue to conform to the local realm’s policies. It is generally safe for a KDC to make verbatim copies of the contents of such a CAMMAC into a new CAMMAC when handling a TGS-REQ. Particularly strict implementations might conduct additional policy checks on the contents of a CAMMAC originating from a local realm KDC if the local realm’s policy materially changes during the life of the CAMMAC.

A KDC MAY omit the kdc-verifier from the CAMMAC when it is not needed, according to how realm policies will subsequently treat the containing ticket. An implementation might choose to do this

omission to reduce the size of tickets it issues. Some examples of when such an omission is safe are:

1. For a local realm TGT, if all local realm KDCs correctly filter out client-submitted CAMMACs, the local realm origin criteria listed above allow omission of the kdc-verifier.

2. An application service might not use the S4U2Proxy extension, or the realm policy might disallow the use of S4U2Proxy by that service. In such situations where there is no flow of authorization data from the service to the KDC, the application service could modify the CAMMAC contents, but such modifications would have no effect on other services. Because of the lack of security impact to other application services, the KDC MAY omit the kdc-verifier from a CAMMAC contained in a ticket for that service.

Extracting a CAMMAC from a ticket for use as a credential removes it from the context of the ticket. In the general case, this could turn it into a bearer token, with all of the associated security implications. Also, the CAMMAC does not itself necessarily contain sufficient information to identify the client principal. Therefore, application protocols that rely on extracted CAMMACs might need to duplicate a substantial portion of the ticket contents and include that duplicated information in the authorization data contained within the CAMMAC. The extent of this duplication would depend on the security properties required by the application protocol.

The method for computing the kdc-verifier binds it only to the authorization data contained within the CAMMAC; it does not bind the CAMMAC to any authorization data within the containing ticket but outside the CAMMAC. At least one (non-standard) authorization data type, AD-SIGNEDPATH, attempts to bind to other authorization data in a ticket, and it is very difficult for two such authorization data types to coexist.

The kdc-verifier in CAMMAC does not bind the service principal name to the CAMMAC contents, because the service principal name is not part of the EncTicketPart. An entity that has access to the keys of two different service principals can decrypt a ticket for one service and encrypt it in the key of the other service, altering the svc-verifier to match. Both the kdc-verifier and the svc-verifier would still validate, but the KDC never issued this fabricated ticket. The impact of this manipulation is minor if the CAMMAC contents only communicate attributes related to the client. If an application requires an authenticated binding between the service principal name and the CAMMAC or ticket contents, the KDC MUST include in the CAMMAC some authorization data element that names the service principal.

9. Acknowledgements

Shawn Emery, Sam Hartman, Greg Hudson, Ben Kaduk, Barry Leiba, Meral Shirazipour, Zhanna Tsitkov, Qin Wu, and Kai Zheng provided helpful technical and editorial feedback on earlier versions of this document. Thomas Hardjono helped with the initial editing to split this document from a predecessor document that had a wider scope.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[X.680] ITU-T, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation -- ITU-T Recommendation X.680 (ISO/IEC International Standard 8824-1:2008)", 2008.

[X.690] ITU-T, "Information technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) -- ITU-T Recommendation X.690 (ISO/IEC International Standard 8825-1:2008)", 2008.

10.2. Informative References

[MS-SFU] Microsoft, "[MS-SFU]: Kerberos Protocol Extensions: Service for User and Constrained Delegation Protocol", January 2013, <http://msdn.microsoft.com/en-us/library/cc246071.aspx>.

Authors' Addresses

Simo Sorce
Red Hat

Email: ssorce@redhat.com

Tom Yu
MIT

Email: tlyu@mit.edu

KITTEN
Internet-Draft
Updates: 2743, 2744 (if approved)
Intended status: Standards Track
Expires: August 9, 2019

R. Harwood
Red Hat
N. Williams
Cryptonector

February 5, 2019

Channel Binding Signalling for the Generic Security Services Application Programming Interface

draft-ietf-kitten-channel-bound-flag-04

Abstract

Channel binding is a technique that allows applications to use a secure channel at a lower layer without having to use authentication at that lower layer. The concept of channel binding comes from the Generic Security Services Application Programming Interface (GSS-API). It turns out that the semantics commonly implemented are different than those specified in the base GSS-API RFC (RFC2743), and that that specification has a serious bug. This document addresses both the inconsistency as-implemented and the specification bug.

This Internet-Draft proposes the addition of a "channel bound" return flag for the GSS_Init_sec_context() and GSS_Accept_sec_context() functions. Two behaviors are specified: a default, safe behavior reflecting existing implementation deployments, and a behavior that is only safe when the application specifically tells the GSS-API that it (the application) supports the new behavior. Additional API elements related to this are also added, including a new security context establishment API.

Status of This Memo

Table of Contents

1. Introduction

The GSS-API [RFC2743] supports "channel binding" [RFC5056], a technique for detection of man-in-the-middle (MITM) attacks in secure channels at lower network layers. This facility is meant to be all-or-nothing: either both the initiator and acceptor use it and it succeeds, or both must not use it. This has created a negotiation problem when retrofitting the use of channel binding into existing application protocols.

However, GSS-APIv2u1 [RFC2743] does not specify channel binding behavior when only one party provides provides none. In practice, some mechanisms (such as Kerberos [RFC4121]) ignore channel bindings when the acceptor provides none, but not when the initiator provides none. Note that it would be useless to allow security context establishment to succeed when the initiator does not provide channel bindings but the acceptor does, at least as long as there's no outward indication of whether channel binding was used! Since the GSS-APIv2u1 does not provide any such indication, this document corrects that flaw.

Allowing the connection to succeed when an initiator provides bindings but an acceptor does not has helped deployment of channel binding in existing applications: first fix all the initiators, then fix all the acceptors. But even this technique is insufficient when there are many clients to fix, such that fixing them all will take a long time. Additionally, it limits the usefulness of channel bindings, while allowing the acceptor to provide but not enforce would protect against man in the middle attacks (for channel binding aware initiators).

This document proposes a new method for deployment of channel binding that allows the feature to be enabled on the acceptor side before fixing all initiators. If the GSS-API had always had a return flag by which to indicate channel binding state then we could have had a simpler method of deploying channel binding: applications check that return flag and act accordingly (e.g., fail when channel binding is required). We cannot safely introduce this behavior now without an indication of support by the application.

Additionally, there may be applications where it is important for initiators to know that acceptors did use channel binding, and even to know whether a mechanism is capable of indicating as much. We add a request flag and a mechanism attribute for such applications.

1.1. Design and Future directions

The design for signalling application flag support and empty contexts is based on the Java Bindings of the GSS-API [RFC5653]. This document begins introduction of additional context inquiry and mutation functions, which eventually will also allow for simplified stepping to replace the GSS_Init/Accept_sec_context() loop.

1.2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Channel Binding State Extension

We propose a new return flag for GSS_Init_sec_context() and GSS_Accept_sec_context(), as well as a pair of functions for a) creating "empty" security context handles, b) requesting flags and indicating which flags the application understands. We also add a new mechanism attribute for supporting channel binding confirmation.

C bindings of these extensions are provided along the lines of [RFC2744] and [RFC5587].

In the future we might move more of the many input (and output) arguments to GSS_Init_sec_context() and GSS_Accept_sec_context() into mutators on empty security context handles.

2.1. GSS_Create_sec_context()

Inputs:

- o <none>

Outputs:

- o major_status INTEGER
- o minor_status INTEGER -- note: mostly useless, but we should keep it
- o context SECURITY CONTEXT -- "empty" security context

Return major status codes:

- o GSS_S_COMPLETE indicates success.
- o GSS_S_UNAVAILABLE indicates that memory is not available, for example.
- o GSS_S_FAILURE indicates a general failure.

This function creates an "empty" security context handle that can be passed to GSS_Init_sec_context() or GSS_Accept_sec_context() where they expect GSS_C_NO_CONTEXT.

2.1.1. C-Bindings

```
OM_uint32
gss_create_sec_context(OM_uint32 *minor_status,
                       gss_ctx_id_t *context);
```

2.2. GSS_Set_context_flags()

Inputs:

context CONTEXT HANDLE

req_flags FLAGS Requested flags. Applicable to acceptors and initiators.

ret_flags_understood FLAGS The set of return flags understood by the caller.

Outputs:

- major_status INTEGER
- minor_status INTEGER

Return major status codes:

- GSS_S_COMPLETE indicates success.
- GSS_S_FAILURE indicates a general failure.

This function tells the mechanism (when one is eventually chosen and invoked) that the application requests the given req_flags and is prepared to check the flags in the given ret_flags_understood. Mechanisms SHOULD NOT limit flags returned to those in ret_flags_understood, but MAY alter behavior accordingly. Initiators can override the req_flags in their GSS_Init_sec_context() call, but if no flags are requested there then the req_flags set on the empty context will be used. GSS_Accept_sec_context() is not required to perform any action based on req_flags at this time.

NOTE: The abstract GSS-API [RFC2743] uses individual elements--one per-flag--instead of a "FLAGS" type. This is unwieldy, therefore we introduce an abstract type named "FLAGS" to act as a set of all the request/return flags defined for the abstract GSS-API.

2.2.1. C-Bindings

```
OM_uint32
gss_set_context_flags(OM_uint32 *minor_status,
                      gss_ctx_id_t context,
                      uint64_t req_flags,
                      uint64_t ret_flags_understood);
```

2.3. Return Flag for Channel Binding State Signalling

Whenever both the initiator and the acceptor provide matching channel bindings to GSS_Init_sec_context() and GSS_Accept_sec_context(), respectively, then the mechanism SHALL indicate that the context is channel bound via an output flag, ret_channel_bound_flag, for the established context. Note that some mechanisms have no way for the acceptor to signal CB success to the initiator, in which case GSS_Init_sec_context() MUST NOT output the ret_channel_bound_flag.

2.3.1. C-Bindings

```
#define GSS_C_CHANNEL_BOUND_FLAG 2048 /* 0x00000800 */
```

2.4. New Mechanism Attribute

- o We add a new mechanism attribute, GSS_C_MA_CBINDING_CONFIRM, to indicate that the initiator can and always does learn whether the acceptor application supplied channel bindings (assuming mutual auth has been requested). Mechanisms advertising this attribute MUST always indicate acceptor channel bound state to the initiator.

OID assignments TBD.

2.5. Request Flag for Acceptor Confirmation of Channel Binding

We add a new request flag for GSS_Init_sec_context(), req_cb_confirmation_flag, to be used by initiators that insist on acceptors providing channel bindings. If set, the mechanism MUST prefer establishment of contexts which provide channel binding confirmation. It SHOULD NOT fail to negotiate just because it cannot provide the GSS_C_MA_CBINDING_CONFIRM attribute.

2.5.1. C-Bindings

Because GSS_C_CHANNEL_BOUND_FLAG is a return flag only, and this flag is a request flag only, and to save on precious flag bits, we use the same flag bit assignment for both flags:

```
#define GSS_C_CB_CONFIRM_FLAG 2048 /* 0x00000800 */
```

2.6. Handling Empty Contexts in Other GSS-API Functions

GSS_Init_sec_context() and GSS_Accept_sec_context() operate on empty security contexts as specified above (i.e., examining flags).

All other GSS-API functions MUST treat empty contexts as they would GSS_C_NO_CONTEXT as well. For most functions, this will result in returning GSS status code GSS_S_NO_CONTEXT.

GSS_Delete_sec_context() MUST NOT output a context deletion token when applied to empty security contexts.

3. Modified Channel Binding Semantics

The channel binding semantics of the base GSS-API are modified as follows:

- o Whenever both the initiator and acceptor have provided input_channel_bindings to GSS_Init/Accept_sec_context() and the channel bindings do not match, then the mechanism MUST fail to establish a security context token. (This is a restatement of an existing requirement in the base specification.)

- o Whenever the acceptor application has a) provided channel bindings to GSS_Accept_sec_context(), and b) not indicated support for the ret_channel_bound_flag flag, then the mechanism MUST fail to establish a security context if the initiator did not provide channel bindings data. This requirement is critical for security purposes, to make applications predating this document secure, and this requirement reflects actual implementations as deployed.

- o Whenever the initiator application has a) provided channel bindings to GSS_Init_sec_context(), and b) not indicated support for the ret_channel_bound_flag flag, then the mechanism SHOULD NOT fail to establish a security context just because the acceptor failed to provide channel bindings data. This strong sugestion is for interoperability purposes, and reflects actual implementations that have been deployed.

- o Whenever the application has a) provided channel bindings to GSS_Init_sec_context() or GSS_Accept_sec_context(), and b) indicated support for the ret_channel_bound_flag flag, then the mechanism SHOULD NOT fail to establish a security context just because the peer did not provide channel bindings data. The mechanism MUST output the ret_channel_bound_flag if both peers provided the same input_channel_bindings to GSS_Init_sec_context() and GSS_Accept_sec_context(). The mechanism MUST NOT output the ret_channel_bound_flag if either (or both) peer did not provide input_channel_bindings to GSS_Init/Accept_sec_context(). This requirement restores the original base GSS-API specified behavior, with the addition of the ret_channel_bound_flag flag.

4. Security Considerations

This document deals with security. There are no security considerations that should be documented separately in this section. To recap, this document fixes a significant flaw in the base GSS-API [RFC2743] specification that fortunately has not been implemented, and it adds a feature (that should have been in the base specification) for improved negotiation of use of channel binding [RFC5056].

5. IANA Considerations

The GSS-API mechanism attribute is to be added to the "SMI Security for Mechanism gsscma Codes" registry established by RFC5587 [RFC5587]. See Section 2.4.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <https://www.rfc-editor.org/info/rfc2743>.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, DOI 10.17487/RFC2744, January 2000, <https://www.rfc-editor.org/info/rfc2744>.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <https://www.rfc-editor.org/info/rfc5056>.

[RFC5587] Williams, N., "Extended Generic Security Service Mechanism Inquiry APIs", RFC 5587, DOI 10.17487/RFC5587, July 2009, <https://www.rfc-editor.org/info/rfc5587>.

6.2. Informative References

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <https://www.rfc-editor.org/info/rfc4121>.

[RFC5653] Upadhyay, M. and S. Malkani, "Generic Security Service API Version 2: Java Bindings Update", RFC 5653, DOI 10.17487/RFC5653, August 2009, <https://www.rfc-editor.org/info/rfc5653>.

Authors' Addresses

Robbie Harwood
Red Hat, Inc.

Email: rharwood@redhat.com

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 23, 2015

B. Kaduk
MIT

February 19, 2015

Structure of the GSS Negotiation Loop
draft-ietf-kitten-gss-loop-05

Abstract

This document specifies the generic structure of the negotiation loop to establish a GSS security context between initiator and acceptor. The control flow of the loop is indicated for both parties, including error conditions, and indications are given for where application-specific behavior must be specified.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Generic Security Service Application Program Interface version 2 [RFC2743] provides a generic interface for security services, in the form of an abstraction layer over the underlying security mechanisms that an application may use. A GSS initiator and acceptor exchange messages, called tokens, until a security context is established. Such a security context allows for each party to authenticate the other, the passing of confidential and/or integrity-protected messages between the initiator and acceptor, the generation of identical pseudo-random bit strings by both participants [RFC4401], and more.

During context establishment, security context tokens are exchanged synchronously, one at a time; the initiator sends the first context token. The number of tokens which must be exchanged between initiator and acceptor in order to establish the security context is dependent on the underlying mechanism as well as the desired properties of the security context, and is in general not known to the application. Accordingly, the application’s control flow must include a loop within which GSS security context tokens are

exchanged, which terminates upon successful establishment of a security context or an error condition. The GSS-API, together with its security mechanisms, specifies the format and encoding of the context tokens themselves, but the application protocol must specify the necessary framing for the application to determine what octet strings constitute GSS security context tokens and pass them into the GSS-API implementation as appropriate.

The GSS-API C bindings [RFC2744] provide some example code for such a negotiation loop, but this code does not specify the application's behavior on unexpected or error conditions. As such, individual application protocol specifications have had to specify the structure of their GSS negotiation loops, including error handling, on a per-protocol basis. [RFC4462], [RFC3645], [RFC5801], [RFC4752], [RFC2203] This represents a substantial duplication of effort, and the various specifications go into different levels of detail and describe different possible error conditions. It is therefore preferable to have the structure of the GSS negotiation loop, including error conditions and token passing, described in a single specification, which can then be referred to from other documents in lieu of repeating the structure of the loop each time. This document will perform that role.

The necessary requirements for correctly performing a GSS negotiation loop are essentially all included in [RFC2743], but they are scattered in many different places. This document brings all the requirements together into one place for the convenience of implementors, even though the normative requirements remain in [RFC2743]. In a few places, this document notes additional behavior which is useful for applications but is not mandated by [RFC2743].

2. Application Protocol Requirements

Part of the purpose of this document is to guide the development of new application protocols using the GSS-API, as well as the development of new application software using such protocols. The following list is features which are necessary or useful in such an application protocol:

- o A way to frame and identify security context negotiation tokens in the loop.

- o Error tokens should generally also get special framing, as the recipient may have no other way to distinguish unexpected error context tokens from per-message tokens.

- o Failing that, a way to indicate error status from one peer to the other, possibly accompanied by a descriptive string.

- o A protocol may use the negotiated GSS security context for per-message operations; in such cases, the protocol will need a way to frame and identify those per-message tokens and the nature of their contents. For example, a protocol message may be accompanied by the output of GSS_GetMIC() over that message; the protocol must identify the location and size of that MIC token, and indicate that it is a MIC token and what cleartext it corresponds to.

- o Applications are responsible for authorization of the authenticated peer principal names which are supplied by the GSS-API. Such names are mechanism-specific, and may come from a different portion of a federated identity scheme. Application protocols may need to supply additional information to support the authorization of access to a given resource, such as the SSHv2 "username" parameter.

3. Loop Structure

The loop is begun by the appropriately named initiator, which calls GSS_Init_sec_context() with an empty (zero-length) input_token and a fixed set of input flags containing the desired attributes for the security context. The initiator should not change any of the input parameters to GSS_Init_sec_context() between calls to it during the loop, with the exception of the input_token parameter, which will contain a message from the acceptor after the initial call, and the input_context_handle, which must be the result returned in the output_context_handle of the previous call to GSS_Init_sec_context() (GSS_C_NO_CONTEXT for the first call). (In the C bindings, there is only a single read/modify context_handle argument, so the same variable should be passed for each call in the loop.) RFC 2743 only requires that the claimant_cred_handle argument remain constant over all calls in the loop, but the other non-excepted arguments should also remain fixed for reliable operation.

The following subsections will describe the various steps of the loop, without special consideration to whether a call to GSS_Init_sec_context() or GSS_Accept_sec_context() is the first such call in the loop.

3.1. Anonymous Initiators

If the initiator is requesting anonymity by setting the anon_req_flag input to GSS_Init_sec_context(), then on non-error returns from GSS_Init_sec_context() (that is, when the major status is GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED), the initiator must verify that the output value of anon_state from GSS_Init_sec_context() is true before sending the security context token to the acceptor.

Failing to perform this check could cause the initiator to lose anonymity.

3.2. GSS_Init_sec_context

The initiator calls GSS_Init_sec_context(), using the input_context_handle for the current security context being established and its fixed set of input parameters, and the input_token received from the acceptor (if this is not the first iteration of the loop). The presence or absence of a nonempty output_token and the value of the major status code are the indicators for how to proceed:

- If the major status code is GSS_S_COMPLETE and the output_token is empty, then the context negotiation is fully complete and ready for use by the initiator with no further actions.

- If the major status code is GSS_S_COMPLETE and the output_token is nonempty, then the initiator’s portion of the security context negotiation is complete but the acceptor’s is not. The initiator must send the output_token to the acceptor so that the acceptor can establish its half of the security context.

- If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is nonempty, the context negotiation is incomplete. The initiator must send the output_token to the acceptor and await another input_token from the acceptor.

- If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is empty, the mechanism has produced an output which is not compliant with [RFC2743]. However, there are some known implementations of certain mechanisms such as NTLMSSP [NTLMSSP] which do produce empty context negotiation tokens. For maximum interoperability, applications should be prepared to accept such tokens, and should transmit them to the acceptor if they are generated.

- If the major status code is any other value, the context negotiation has failed. If the output_token is nonempty, it is an error token, and the initiator should send it to the acceptor. If the output_token is empty, then the initiator should indicate the failure to the acceptor if an appropriate application-protocol channel to do so is available.

3.3. Sending from Initiator to Acceptor

The establishment of a GSS security context between initiator and acceptor requires some communication channel by which to exchange the context negotiation tokens. The nature of this channel is not specified by the GSS specification -- it could be a dedicated TCP channel, a UDP-based RPC protocol, or any other sort of channel. In many cases, the channel will be multiplexed with non-GSS application data; the application protocol must always provide some means by which the GSS context tokens can be identified (e.g., length and start location) and passed through to the mechanism accordingly. The application protocol may also include a facility for indicating errors from one party to the other, which can be used to convey errors resulting from GSS-API calls, when appropriate (such as when no error token was generated by the GSS-API implementation). Note that GSS major and minor status codes are specified by language bindings, not the abstract API; sending a major status code and optionally the display form of the two error codes may be the best that can be done in this case.

However, even the presence of a communication channel does not necessarily indicate that it is appropriate for the initiator to indicate such errors. For example, if the acceptor is a stateless or near-stateless UDP server, there is probably no need for the initiator to explicitly indicate its failure to the acceptor. Conditions such as this can be treated in individual application protocol specifications.

If a regular security context output_token is produced by the call to GSS_Init_sec_context(), the initiator must transmit this token to the acceptor over the application’s communication channel. If the call to GSS_Init_sec_context() returns an error token as output_token, it is recommended that the initiator transmit this token to the acceptor over the application’s communication channel.

3.4. Acceptor Sanity Checking

The acceptor’s half of the negotiation loop is triggered by the receipt of a context token from the initiator. Before calling GSS_Accept_sec_context(), the acceptor may find it useful to perform some sanity checks on the state of the negotiation loop.

If the acceptor receives a context token but was not expecting such a token (for example, if the acceptor’s previous call to GSS_Accept_sec_context() returned GSS_S_COMPLETE), this is probably an error condition indicating that the initiator’s state is invalid. See Section 4.3 for some exceptional cases. It is likely appropriate

for the acceptor to report this error condition to the initiator via the application's communication channel.

If the acceptor is expecting a context token (e.g., if the previous call to GSS_Accept_sec_context() returned GSS_S_CONTINUE_NEEDED), but does not receive such a token within a reasonable amount of time after transmitting the previous output_token to the initiator, the acceptor should assume that the initiator's state is invalid (time out) and fail the GSS negotiation. Again, it is likely appropriate for the acceptor to report this error condition to the initiator via the application's communication channel.

3.5. GSS_Accept_sec_context

The GSS acceptor responds to the actions of an initiator; as such, there should always be a nonempty input_token to calls to GSS_Accept_sec_context(). The input_context_handle parameter will always be given as the output_context_handle from the previous call to GSS_Accept_sec_context() in a given negotiation loop, or GSS_C_NO_CONTEXT on the first call, but the acceptor_cred_handle and chan_bindings arguments should remain fixed over the course of a given GSS negotiation loop. [RFC2743] only requires that the acceptor_cred_handle remain fixed throughout the loop, but the chan_bindings argument should also remain fixed for reliable operation.

The GSS acceptor calls GSS_Accept_sec_context(), using the input_context_handle for the current security context being established and the input_token received from the initiator. The presence or absence of a nonempty output_token and the value of the major status code are the indicators for how to proceed:

- If the major status code is GSS_S_COMPLETE and the output_token is empty, then the context negotiation is fully complete and ready for use by the acceptor with no further actions.

- If the major status code is GSS_S_COMPLETE and the output_token is nonempty, then the acceptor's portion of the security context negotiation is complete but the initiator's is not. The acceptor must send the output_token to the initiator so that the initiator can establish its half of the security context.

- If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is nonempty, the context negotiation is incomplete. The acceptor must send the output_token to the initiator and await another input_token from the initiator.

- o If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is empty, the mechanism has produced an output which is not compliant with [RFC2743]. However, there are some known implementations of certain mechanisms such as NTLMSSP [NTLMSSP] which do produce empty context negotiation tokens. For maximum interoperability, applications should be prepared to accept such tokens, and should transmit them to the initiator if they are generated.

- o If the major status code is any other value, the context negotiation has failed. If the output_token is nonempty, it is an error token, and the acceptor should send it to the initiator. If the output_token is empty, then the acceptor should indicate the failure to the initiator if an appropriate application-protocol channel to do so is available.

3.6. Sending from Acceptor to Initiator

The mechanism for sending the context token from acceptor to initiator will depend on the nature of the communication channel between the two parties. For a synchronous bidirectional channel, it can be just another piece of data sent over the link, but for a stateless UDP RPC acceptor, the token will probably end up being sent as an RPC output parameter. Application protocol specifications will need to specify the nature of this behavior.

If the application protocol has the initiator driving the application’s control flow, it is particularly helpful for the acceptor to indicate a failure to the initiator, as mentioned in some of the above cases conditional on "an appropriate application-protocol channel to do so".

If a regular security context output_token is produced by the call to GSS_Accept_sec_context(), the acceptor must transmit this token to the initiator over the application’s communication channel. If the call to GSS_Accept_sec_context() returns an error token as output_token, it is recommended that the acceptor transmit this token to the initiator over the application’s communication channel.

3.7. Initiator input validation

The initiator’s half of the negotiation loop is triggered (after the first call) by receipt of a context token from the acceptor. Before calling GSS_Init_sec_context(), the initiator may find it useful to perform some sanity checks on the state of the negotiation loop.

If the initiator receives a context token but was not expecting such a token (for example, if the initiator’s previous call to

GSS_Init_sec_context() returned GSS_S_COMPLETE), this is probably an error condition indicating that the acceptor's state is invalid. See Section 4.3 for some exceptional cases. It may be appropriate for the initiator to report this error condition to the acceptor via the application's communication channel.

If the initiator is expecting a context token (that is, the previous call to GSS_Init_sec_context() returned GSS_S_CONTINUE_NEEDED), but does not receive such a token within a reasonable amount of time after transmitting the previous output_token to the acceptor, the initiator should assume that the acceptor's state is invalid and fail the GSS negotiation. Again, it may be appropriate for the initiator to report this error condition to the acceptor via the application's communication channel.

3.8. Continue the Loop

If the loop is in neither a success or failure condition, then the loop must continue. Control flow returns to Section 3.2.

4. After Security Context Negotiation

Once a party has completed its half of the security context and fulfilled its obligations to the other party, the context is complete, but it is not necessarily ready and appropriate for use. In particular, the security context flags may not be appropriate for the given application's use. In some cases the context may be ready for use before the negotiation is complete, see Section 4.2.

The initiator specifies as part of its fixed set of inputs to GSS_Init_sec_context() values for all defined request flag booleans, among them: deleg_req_flag, mutual_req_flag, replay_det_req_flag, sequence_req_flag, conf_req_flag, and integ_req_flag. Upon completion of the security context negotiation, the initiator must verify that the values of the deleg_state, mutual_state, replay_det_state, sequence_state, conf_avail, and integ_avail (and any additional flags added by extensions) from the last call to GSS_Init_sec_context() correspond to the requested flags. If a flag was requested but is not available, and that feature is necessary for the appplication protocol, the initiator must destroy the security context and not use the security context for application traffic.

Application protocol specifications citing this document should indicate which context flags are required for their application protocol.

The acceptor receives as output the following booleans: deleg_state, mutual_state, replay_det_state, sequence_state, anon_state,

trans_state, conf_avail, and integ_avail, and any additional flags added by extensions to the GSS-API. The acceptor must verify that any flags necessary for the application protocol are set. If a necessary flag is not set, the acceptor must destroy the security context and not use the security context for application traffic.

4.1. Authorization Checks

The acceptor receives as one of the outputs of GSS_Accept_sec_context() the name of the initiator which has authenticated during the security context negotiation. Applications need to implement authorization checks on this received name ('client_name' in the sample code) before providing access to restricted resources. In particular, security context negotiation can be successful when the client is anonymous or is from a different identity realm than the acceptor, depending on the details of the mechanism used by the GSS-API to establish the security context. Acceptor applications can check which target name was used by the initiator, but the details are out of scope for this document. See [RFC2743] sections 2.2.6 and 1.1.5. Additional information can be available in GSS-API Naming Extensions, [RFC6680].

4.2. Using Partially Complete Security Contexts

For mechanism/flag combinations that require multiple token exchanges, the GSS-API specification [RFC2743] provides a prot_ready_state output value from GSS_Init_sec_context() and GSS_Accept_sec_context(), which indicates when per-message operations are available. However, many mechanism implementations do not provide this functionality, and the analysis of the security consequences of its use is rather complicated, so it is not expected to be useful in most application protocols.

In particular, mutual authentication, replay protection, and other services (if requested) are services which will be active when GSS_S_COMPLETE is returned, but which are not necessarily active before the security context is fully established.

4.3. Additional Context Tokens

Under some conditions, a context token will be received by a party to a security context negotiation after that party has completed the negotiation (i.e., after GSS_Init_sec_context() or GSS_Accept_sec_context() has returned GSS_S_COMPLETE). Such tokens must be passed to GSS_Process_context_token() for processing. It may not always be necessary for a mechanism implementation to generate an error token on the initiator side, or for an initiator application to transmit that token to the acceptor; such decisions are out of scope

for this document. Both peers should always be prepared to process such tokens, and application protocols should provide means by which they can be transmitted.

Such tokens can be security context deletion tokens, emitted when the remote party called GSS_Delete_sec_context() with a non-null output_context_token parameter, or error tokens, emitted when the remote party experiences an error processing the last token in a security context negotiation exchange. Errors experienced when processing tokens earlier in the negotiation would be transmitted as normal security context tokens and processed by GSS_Init_sec_context() or GSS_Accept_sec_context(), as appropriate. With the GSS-API version 2, it is not recommended to use security context deletion tokens, so error tokens are expected to be the most common form of additional context token for new application protocols.

GSS_Process_context_token() may indicate an error in its major_status field if an error is encountered locally during token processing, or to indicate that an error was encountered on the peer and conveyed in an error token. See [RFC2743] Errata #4151. Regardless of the major_status output of GSS_Process_context_token(), GSS_Inquire_context() should be used after processing the extra token, to query the status of the security context and whether it can supply the features necessary for the application protocol.

At present, all tokens which should be handled by GSS_Process_context_token() will lead to the security context being effectively unusable. Future extensions to the GSS-API may allow for applications to continue to function after a call to GSS_Process_context_token(), and it is expected that the outputs of GSS_Inquire_context() will indicate whether it is safe to do so. However, since there are no such extensions at present (error tokens and deletion tokens both result in the security context being essentially unusable), there is no guidance to give to applications regarding this possibility at this time.

Even if GSS_Process_context_token() processes an error or deletion token which renders the context essentially unusable, the resources associated with the context must eventually be freed with a call to GSS_Delete_sec_context(), just as would be needed if GSS_Init_sec_context() or GSS_Accept_sec_context() had returned an error while processing an input context token and the input_context_handle was not GSS_C_NO_CONTEXT. RFC 2743 has some text which is slightly ambiguous in this regard, but the best practice is to always call GSS_Delete_sec_context().

5. Sample Code

This section gives sample code for the GSS negotiation loop, both for a regular application and for an application where the initiator wishes to remain anonymous. Since the code for the two cases is very similar, the anonymous-specific additions are wrapped in a conditional check; that check and the conditional code may be ignored if anonymous processing is not needed.

Since the communication channel between the initiator and acceptor is a matter for individual application protocols, it is inherently unspecified at the GSS-API level, which can lead to examples that are less satisfying than may be desired. For example, the sample code in [RFC2744] uses an unspecified send_token_to_peer() routine. Fully correct and general code to frame and transmit tokens requires a substantial amount of error checking and would detract from the core purpose of this document, so we only present the function signature for one example of what such functions might be, and leave some comments in the otherwise-empty function bodies.

This sample code is written in C, using the GSS-API C bindings [RFC2744]. It uses the macro GSS_ERROR() to help unpack the various sorts of information which can be stored in the major status field; supplementary information does not necessarily indicate an error. Applications written in other languages will need to exercise care that checks against the major status value are written correctly.

This sample code should be compilable as a standalone program, linked against a GSS-API library. In addition to supplying implementations for the token transmission/receipt routines, in order for the program to successfully run when linked against most GSS-API libraries, the initiator will need to specify an explicit target name for the acceptor, which must match the credentials available to the acceptor. A skeleton for how this may be done is provided, using a dummy name.

This sample code assumes v2 of the GSS-API. Applications wishing to remain compatible with v1 of the GSS-API may need to perform additional checks in some locations.

5.1. GSS Application Sample Code

```
#include <unistd.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gssapi/gssapi.h>
```

```
/*
 * This helper is used only on buffers that we allocate ourselves (e.g.,
 * from receive_token()).  Buffers allocated by GSS routines must use
 * gss_release_buffer().
 */
static void
release_buffer(gss_buffer_t buf)
{
    free(buf->value);
    buf->value = NULL;
    buf->length = 0;
}

/*
 * Helper to send a token on the specified fd.
 *
 * If errors are encountered, this routine must not directly cause
 * termination of the process, because compliant GSS applications
 * must release resources allocated by the GSS library before
 * exiting.
 *
 * Returns 0 on success, non-zero on failure.
 */
static int
send_token(int fd, gss_buffer_t token)
{
    /*
     * Supply token framing and transmission code here.
     *
     * It is advisable for the application protocol to specify the
     * length of the token being transmitted, unless the underlying
     * transit does so implicitly.
     *
     * In addition to checking for error returns from whichever
     * syscall(s) are used to send data, applications should have
     * a loop to handle EINTR returns.
     */
    return 1;
}

/*
 * Helper to receive a token on the specified fd.
 *
 * If errors are encountered, this routine must not directly cause
 * termination of the process, because compliant GSS applications
 * must release resources allocated by the GSS library before
 * exiting.
 *
```

```
 * Returns 0 on success, non-zero on failure.
 */
static int
receive_token(int fd, gss_buffer_t token)
{
    /*
     * Supply token framing and transmission code here.
     *
     * In addition to checking for error returns from whichever
     * syscall(s) are used to receive data, applications should have
     * a loop to handle EINTR returns.
     *
     * This routine is assumed to allocate memory for the local copy
     * of the received token, which must be freed with release_buffer().
     */
    return 1;
}

static void
do_initiator(int readfd, int writefd, int anon)
{
    int initiator_established = 0, ret;
    gss_ctx_id_t ctx = GSS_C_NO_CONTEXT;
    OM_uint32 major, minor, req_flags, ret_flags;
    gss_buffer_desc input_token = GSS_C_EMPTY_BUFFER;
    gss_buffer_desc output_token = GSS_C_EMPTY_BUFFER;
    gss_buffer_desc name_buf = GSS_C_EMPTY_BUFFER;
    gss_name_t target_name = GSS_C_NO_NAME;

    /* Applications should set target_name to a real value. */
    name_buf.value = "<service>@<hostname.domain>";
    name_buf.length = strlen(name_buf.value);
    major = gss_import_name(&minor, &name_buf,
                            GSS_C_NT_HOSTBASED_SERVICE, &target_name);
    if (GSS_ERROR(major)) {
        warnx(1, "Could not import name\n");
        goto cleanup;
    }

    /* Mutual authentication will require a token from acceptor to
     * initiator, and thus a second call to gss_init_sec_context(). */
    req_flags = GSS_C_MUTUAL_FLAG | GSS_C_CONF_FLAG | GSS_C_INTEG_FLAG;
    if (anon)
        req_flags |= GSS_C_ANON_FLAG;

    while (!initiator_established) {
        /* The initiator_cred_handle, mech_type, time_req,
         * input_chan_bindings, actual_mech_type, and time_rec
```

```
         * parameters are not needed in many cases.  We pass
         * GSS_C_NO_CREDENTIAL, GSS_C_NO_OID, 0, NULL, NULL, and NULL
         * for them, respectively. */
        major = gss_init_sec_context(&minor, GSS_C_NO_CREDENTIAL, &ctx,
                                     target_name, GSS_C_NO_OID,
                                     req_flags, 0, NULL, &input_token,
                                     NULL, &output_token, &ret_flags,
                                     NULL);
        /* This was allocated by receive_token() and is no longer
         * needed.  Free it now to avoid leaks if the loop continues. */
        release_buffer(&input_token);
        if (anon) {
            /* Initiators which wish to remain anonymous must check
             * whether their request has been honored before sending
             * each token. */
            if (!(ret_flags & GSS_C_ANON_FLAG)) {
                warnx("Anonymous requested but not available\n");
                goto cleanup;
            }
        }
        /* Always send a token if we are expecting another input token
         * (GSS_S_CONTINUE_NEEDED is set) or if it is nonempty. */
        if ((major & GSS_S_CONTINUE_NEEDED) ||
            output_token.length > 0) {
            ret = send_token(writefd, &output_token);
            if (ret != 0)
                goto cleanup;
        }
        /* Check for errors after sending the token so that we will send
         * error tokens. */
        if (GSS_ERROR(major)) {
            warnx("gss_init_sec_context() error major 0x%x\n", major);
            goto cleanup;
        }
        /* Free the output token’s storage; we don’t need it anymore.
         * gss_release_buffer() is safe to call on the output buffer
         * from gss_int_sec_context(), even if there is no storage
         * associated with that buffer. */
        (void)gss_release_buffer(&minor, &output_token);

        if (major & GSS_S_CONTINUE_NEEDED) {
            ret = receive_token(readfd, &input_token);
            if (ret != 0)
                goto cleanup;
        } else if (major == GSS_S_COMPLETE) {
            initiator_established = 1;
        } else {
            /* This situation is forbidden by RFC 2743.  Bail out. */
```

```
            warnx("major not complete or continue but not error\n");
            goto cleanup;
        }
    }   /* while (!initiator_established) */
    if ((ret_flags & req_flags) != req_flags) {
        warnx("Negotiated context does not support requested flags\n");
        goto cleanup;
    }
    printf("Initiator's context negotiation successful\n");
cleanup:
    /* We are required to release storage for nonzero-length output
     * tokens.  gss_release_buffer() zeros the length, so we are
     * will not attempt to release the same buffer twice. */
    if (output_token.length > 0)
        (void)gss_release_buffer(&minor, &output_token);
    /* Do not request a context deletion token; pass NULL. */
    (void)gss_delete_sec_context(&minor, &ctx, NULL);
    (void)gss_release_name(&minor, &target_name);
}

/*
 * Perform authorization checks on the initiator's GSS name object.
 *
 * Returns 0 on success (the initiator is authorized) and nonzero
 * when the initiator is not authorized.
 */
static int
check_authz(gss_name_t client_name)
{
    /*
     * Supply authorization checking code here.
     *
     * Options include bitwise comparison of the exported name against
     * a local database, and introspection against name attributes.
     */
    return 0;
}

static void
do_acceptor(int readfd, int writefd)
{
    int acceptor_established = 0, ret;
    gss_ctx_id_t ctx = GSS_C_NO_CONTEXT;
    OM_uint32 major, minor, ret_flags;
    gss_buffer_desc input_token = GSS_C_EMPTY_BUFFER;
    gss_buffer_desc output_token = GSS_C_EMPTY_BUFFER;
    gss_name_t client_name;
```

```
    major = GSS_S_CONTINUE_NEEDED;

    while (!acceptor_established) {
        if (major & GSS_S_CONTINUE_NEEDED) {
            ret = receive_token(readfd, &input_token);
            if (ret != 0)
                goto cleanup;
        } else if (major == GSS_S_COMPLETE) {
            acceptor_established = 1;
            break;
        } else {
            /* This situation is forbidden by RFC 2743.  Bail out. */
            warnx("major not complete or continue but not error\n");
            goto cleanup;
        }
        /* We can use the default behavior or do not need the returned
         * information for the parameters acceptor_cred_handle,
         * input_chan_bindings, mech_type, time_rec, and
         * delegated_cred_handle and pass the values
         * GSS_C_NO_CREDENTIAL, NULL, NULL, NULL, and NULL,
         * respectively.  In some cases the src_name will not be
         * needed, but most likely it will be needed for some
         * authorization or logging functionality. */
        major = gss_accept_sec_context(&minor, &ctx,
                                       GSS_C_NO_CREDENTIAL,
                                       &input_token, NULL,
                                       &client_name, NULL,
                                       &output_token, &ret_flags, NULL,
                                       NULL);
        /* This was allocated by receive_token() and is no longer
         * needed.  Free it now to avoid leaks if the loop continues. */
        release_buffer(&input_token);
        /* Always send a token if we are expecting another input token
         * (GSS_S_CONTINUE_NEEDED is set) or if it is nonempty. */
        if ((major & GSS_S_CONTINUE_NEEDED) ||
            output_token.length > 0) {
            ret = send_token(writefd, &output_token);
            if (ret != 0)
                goto cleanup;
        }
        /* Check for errors after sending the token so that we will send
         * error tokens. */
        if (GSS_ERROR(major)) {
            warnx("gss_accept_sec_context() error major 0x%x\n", major);
            goto cleanup;
        }
        /* Free the output token’s storage; we don’t need it anymore.
         * gss_release_buffer() is safe to call on the output buffer
```

```
         * from gss_accept_sec_context(), even if there is no storage
         * associated with that buffer. */
        (void)gss_release_buffer(&minor, &output_token);
    }   /* while (!acceptor_established) */
    if (!(ret_flags & GSS_C_INTEG_FLAG)) {
        warnx("Negotiated context does not support integrity\n");
        goto cleanup;
    }
    printf("Acceptor’s context negotiation successful\n");
    ret = check_authz(client_name);
    if (ret != 0)
        printf("Client is not authorized; rejecting access\n");
cleanup:
    release_buffer(&input_token);
    /* We are required to release storage for nonzero-length output
     * tokens.  gss_release_buffer() zeros the length, so we are
     * will not attempt to release the same buffer twice. */
    if (output_token.length > 0)
        (void)gss_release_buffer(&minor, &output_token);
    /* Do not request a context deletion token, pass NULL. */
    (void)gss_delete_sec_context(&minor, &ctx, NULL);
    (void)gss_release_name(&minor, &client_name);
}

int
main(void)
{
    pid_t pid;
    int fd1 = -1, fd2 = -1;

    /* Create fds for reading/writing here. */
    pid = fork();
    if (pid == 0)
        do_initiator(fd1, fd2, 0);
    else if (pid > 0)
        do_acceptor(fd2, fd1);
    else
        err(1, "fork() failed\n");
    exit(0);
}
```

6. IANA Considerations

This document makes no request of IANA.

7. Security Considerations

This document provides a (reasonably) concise description and example for correct construction of the GSS-API security context negotiation loop. Since everything relating to the construction and use of a GSS security context is security-related, there are security-relevant considerations throughout the document. It is useful to call out a few things in this section, though.

The GSS-API uses a request-and-check model for features. An application using the GSS-API requests certain features (confidentiality protection for messages, or anonymity), but such a request does not require the GSS implementation to provide that feature. The application must check the returned flags to verify whether a requested feature is present; if the feature was non-optional for the application, the application must generate an error. Phrased differently, the GSS-API will not generate an error if it is unable to satisfy the features requested by the application.

In many cases it is convenient for GSS acceptors to accept security contexts using multiple acceptor names (such as by using the default credential set, as happens when GSS_C_NO_CREDENTIAL is passed to GSS_Accept_sec_context()). This allows acceptors to use any credentials to which it has access for accepting security contexts, which may not be the desired behavior for a given application. (For example, sshd may only wish to accept only using GSS_C_NT_HOSTBASED credentials of the form host@<hostname>, and not nfs@<hostname>.) Acceptor applications can check which target name was used by the initiator, but the details are out of scope for this document. See [RFC2743] sections 2.2.6 and 1.1.5.

The C sample code uses the macro GSS_ERROR() to assess the return value of gss_init_sec_context() and gss_accept_sec_context(). This is done to indicate where checks are needed in writing code for other languages and what the nature of those checks might be. The C code could be made simpler by omitting that macro. In applications expecting to receive protected octet streams, this macro should not be used on the result of per-message operations, as it omits checking for supplementary status values such as GSS_S_DUPLICATE_TOKEN, GSS_S_OLD_TOKEN, etc.. Use of the GSS_ERROR() macro on the results of GSS-API per-message operations has resulted in security vulnerabilities in existing software!

The security considerations from RFCs 2743 and 2744 remain applicable to consumers of this document.

8. References

8.1. Normative References

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, January 2000.

8.2. Informational References

[RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API Extension for the Generic Security Service Application Program Interface (GSS-API)", RFC 4401, February 2006.

[RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch, "Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol", RFC 4462, May 2006.

[RFC3645] Kwan, S., Garg, P., Gilroy, J., Esibov, L., Westhead, J., and R. Hall, "Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG)", RFC 3645, October 2003.

[RFC5801] Josefsson, S. and N. Williams, "Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family", RFC 5801, July 2010.

[RFC4752] Melnikov, A., "The Kerberos V5 ("GSSAPI") Simple Authentication and Security Layer (SASL) Mechanism", RFC 4752, November 2006.

[RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, September 1997.

[NTLMSSP] Microsoft Corporation, "[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol", May 2014.

[RFC6680] Williams, N., Johansson, L., Hartman, S., and S. Josefsson, "Generic Security Service Application Programming Interface (GSS-API) Naming Extensions", RFC 6680, August 2012.

Appendix A. Acknowledgements

Thanks to Nico Williams and Jeff Hutzleman for prompting me to write this document.

Author’s Address

Benjamin Kaduk
MIT Kerberos Consortium

Email: kaduk@mit.edu

NETWORK WORKING GROUP
Internet-Draft
Intended status: Standards Track
Expires: October 1, 2017

N. Williams
Cryptonector LLC
A. Melnikov
Isode Ltd

March 30, 2017

Namespace Considerations and Registries for GSS-API Extensions
draft-ietf-kitten-gssapi-extensions-iana-11.txt

Abstract

This document describes the ways in which the GSS-API may be extended and directs the creation of an IANA registry for various GSS-API namespaces.

Status of This Memo

Copyright Notice

Table of Contents

1. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

There is a need for private-use and mechanism-specific extensions to the Generic Security Services Application Programming Interface (GSS-API). As such extensions are designed and standardized (or not), both at the IETF and elsewhere, there is a non-trivial risk of namespace pollution and conflicts. To avoid this we set out guidelines for extending the GSS-API and direct the creation of an IANA registry for GSS-API namespaces.

Registrations of individual items and sub-namespaces are allowed. Each sub-namespace may provide different rules for registration, e.g., for mechanism-specific and private-use extensions.

3. Extensions to the GSS-API

Extensions to the GSS-API can be categorized as follows:

- o Abstract API extensions

- o Implementation-specific
- o Mechanism-specific
- o Language binding-specific

Extensions to the GSS-API may be purely semantic, without effect on the GSS-API's namespaces. Or they may introduce new functions, constants, types, etc...; these clearly affect the GSS-API namespaces.

Extensions that affect the GSS-API namespaces should be registered with the IANA as described herein.

4. Generic GSS-API Namespaces

The abstract API namespaces for the GSS-API are:

- o Type names
- o Function names
- o Constant names for various types
- o Constant values for various types
- o Name types (OID, type name and syntaxes)

Additionally we have namespaces associates with the OBJECT IDENTIFIER (OID) type. The IANA already maintains a registry of such OIDs:

- o Mechanism OIDs
- o Name Type OIDs

5. Language Binding-Specific GSS-API Namespaces

Language binding specific namespaces include, among others:

- o Header/interface module names
- o Object classes and/or types
- o Methods and/or functions
- o Constant names
- o Constant values

6. Extension-Specific GSS-API Namespaces

Extensions to the GSS-API may create additional namespaces. See Section 8.2.

7. Registration Form

Registrations for GSS-API namespaces SHALL take the following form:

Registration Field	Possible Values	Description
Bindings	'Generic', 'C-bindings', 'Java', 'C#', <programming language name>	Indicates the name of the programming language that this registration involves, or, if 'Generic', that this is an entry for the generic abstract GSS-API (i.e., not specific to any programming language).
Registration type	'Instance', 'Sub-Namespace'	Indicates whether this entry reserves a given symbol name (and possibly, constant value), or whether it reserves an entire sub-namespace (the name is a pattern) or constant value range.
Object Type	<Symbol> defined by the binding language (for example 'Data-Type', 'Function', 'Method', 'Integer', 'String', 'OID', 'Context-Flag', 'Name-Type', 'Macro', 'Header-File-Name', 'Module-Name', 'Class')	Indicates the type of the object whose symbolic name or constant value this entry registers. The possible values of this field depend on the programming language in question, therefore they are not all specified here.
Symbol Name/Prefix	<Symbol name or name pattern>	The name of a symbol or symbol sub-namespace being

		registered. See Section 8.2.1
Binding of	<Name of abstract API element of which this object is a binding>	If the registration is for a specific language binding of the GSS-API, then this names the abstract API element of which it is a binding (OPTIONAL).
Constant Value/Range	<Constant value> or <constant value range>	The value of the constant named by the <Symbol Name/Prefix>. This field is present only for Instance and Sub-namespace registrations of Constant object types.
Description	<Text>	Description of the registration. Multiple instances of this field may result (see Section 8.2.3).
Registration Rules	<Reference> to an IANA registration Policy defined in [RFC5226] (or an RFC that updates it), for instance 'IESG Approval', 'Expert Review', 'First Come First Served', 'Private Use'.	Describes the rules for allocation of items that fall in this sub-namespace, for entries with Registration Type of Sub-namespace (OPTIONAL). For private use sub-namespaces the submitter MUST provide the e-mail address of a responsible contact. If this field is not specified for a sub-namespace, the default registration rules specified in Section 8.2 apply.
Reference	<Reference>	Reference to a document that describes the registration, if any (OPTIONAL). Multiple instances of this field are allowed, with one reference each.
Expert Reviewer	<Name of expert reviewers, possibly	OPTIONAL, see Section 8.2.2. Multiple instances of this

	WG names>	field are allowed, with one expert reviewer per-instance. Leave this field blank when requesting a registration. It will be filled in by the Expert who reviews the registration.
Expert Review Notes	<Notes from the expert review>	Expert reviewers may request that some comments be included with the registration, e.g., regarding security considerations of the registered extension.
Status	'Registered' or 'Obsoleted'	Status of the registration.
Obsoleting Reference	<Reference>	Reference to a document, if any, that obsoletes this registration. Multiple instances of this field are allowed, with one reference each. (OPTIONAL)

The IANA should create a single GSS-API namespace registry, or multiple registries, one for symbolic names and one for constant values, and/or it may create a registry per-programming language, at its convenience.

Entries in these registries should consist of all the fields from their corresponding registration entries.

Entries should be sorted by: programming language, registration type, object type, and symbol name/pattern.

8. IANA Considerations

This document deals with IANA considerations throughout. Specifically it creates a single registry of various kinds of things, though the IANA may instead create multiple registries, each for one of those kinds of things. Of particular interest may be that IANA will now be the registration authority for the GSS-API name type OID space.

8.1. Initial Namespace Registrations

Initial registry content corresponding to the items defined in [RFC2743], [RFC2744], [RFC2853], [RFC1964] and [RFC4121] and others will be supplied during the IANA review portion of the RFC publishing process. [[Note to RFC Editor: Delete the following sentence before publication:]] The KITTEN WG chairs MUST indicate that such content has been reviewed by the WG and that there is WG consensus that the entries are in agreement with those RFCs.

8.1.1. Example registrations

In order to sanity check recommended IANA registration templates, this section registers several entries.

Registration Field	Possible Values
Bindings	C-bindings
Registration type	Instance
Object Type	Function
Symbol Name	gss_init_sec_context
Binding of	GSS_Init_sec_context
Constant Value/Range	N/A
Description	Create a security context by initiator
Registration Rules	N/A
Reference	RFC 2744
Expert Reviewer	Kitten WG
Expert Review Notes	
Status	Registered
Obsoleting Reference	N/A

Registration Field	Possible Values
Bindings	C-bindings
Registration type	Instance
Object Type	Function
Symbol Name	gss_accept_sec_context
Binding of	GSS_Accept_sec_context
Constant Value/Range	N/A
Description	Accept a security context from initiator
Registration Rules	N/A
Reference	RFC 2744
Expert Reviewer	Kitten WG
Expert Review Notes	
Status	Registered
Obsoleting Reference	N/A

Registration Field	Possible Values
Bindings	C-bindings
Registration type	Instance
Object Type	Context-Flag
Symbol Name	GSS_C_DELEG_FLAG
Binding of	deleg_state or deleg_req_flag
Constant Value/Range	1
Description	On output (if set): Delegated credentials are available via the delegated_cred_handle parameter of GSS_Accept_sec_context. On input (if set): With the call to GSS_Init_sec_context, delegate credentials to the acceptor.
Registration Rules	N/A
Reference	RFC 2744
Expert Reviewer	Kitten WG
Expert Review Notes	
Status	Registered
Obsoleting Reference	N/A

8.2. Registration Maintenance Guidelines

Standards-Track RFCs can create new items with any non-conflicting Symbol Name/Prefix value for this registry by virtue of IESG approval to publish as a Standards-Track RFC -- that is, without additional expert review.

Standards-Track RFCs can mark existing entries as obsolete, and can even create conflicting entries if explicitly stated (the IESG, of course, should review conflicts carefully, and may reject them).

IANA shall also consider submissions from individuals, and via Informational and Experimental RFCs, subject to Expert Review. IANA SHALL allow such registrations if a) they are not conflicting, b) provided that the registration is for object types other than Context-Flags, and c) subject to expert review. Guidelines for expert reviews are given below.

8.2.1. Sub-Namespace Symbol Pattern Matching

Sub-namespace registrations must provide a pattern for matching symbols for which the sub-namespace's registration rules apply. The pattern consists of a string with the following special tokens:

- '*', meaning "match any string."

- "%m", meaning "match any mechanism family short-hand name."

- "%i", meaning "match any implementor vanity short-hand name."

For example, "GSS_%m_*" matches "GSS_krb5_foo" since "krb5" is a common short-hand for the Kerberos V GSS-API mechanism [RFC1964]. But "GSS_%m_*" does not match "GSS_foo_bar" unless "foo" is asserted to be a short-hand for some mechanism.

8.2.2. Expert Reviews of Individual Submissions

[[The following paragraph should be deleted from the document before publication, as it will not age well. It should be moved to the shepherding write-up.]]

Expert review selection SHALL be done as follows. If, at the time that the IANA receives an individual submission for registration in this registry, there are any IETF Working Groups chartered to produce GSS-API-related documents, then the IANA SHALL ask the chairs of such WGs to be expert reviewers or to name one. If there are no such WGs at that time, then the IANA SHALL ask past chairs of the KITTEN WG and the author/editor of this RFC to act as expert reviewers or name an alternate.

Expert reviewers of individual registration submissions with Registration Type == Sub-namespace should check that the registration request has a suitable description (which doesn't need to be sufficiently detailed for others to implement) and that the Symbol Name/Prefix is sufficiently descriptive of the purpose of the sub-namespace or reflective of the name of the submitter or associated company.

Expert reviewers of individual registration submissions with

Registration Type == Instance should check that the Symbol Name falls under a sub-namespace controlled by the submitter. Registration of such entries which do not fall under such a sub-namespace may be allowed provided that they correspond to long existing non-standard extensions to the GSS-API and this can be easily checked or demonstrated, otherwise IESG Protocol Action is REQUIRED (see previous section). Also, reviewers should check that any registration of constant values have a detailed description that is suitable for other implementors to reproduce, and that they don't conflict with other usages or are otherwise dangerous in the reviewers estimation.

Expert reviewers should review impact on mechanisms, security and interoperability, and may reject or annotate registrations which can have mechanism impact that requires IESG protocol action. Consider, for example, new versions of GSS_Init_sec_context() and/or GSS_Accept_sec_context which have new input and/or output parameters which imply changes on the wire or in behaviour that may result in interoperability issues. A reviewer could choose to add notes to the registration describing such issues, or the reviewer might conclude that the danger to Internet interoperability is sufficient to warrant rejecting the registration.

8.2.3. Change Control

Registered entries may be marked obsoleted using the same expert review process as for registering new entries. Obsoleted entries are not, however, to be deleted, but merely marked having Obsoleted Status. Note that entries may be created as obsoleted to record the fact that the given symbol(s) have been used before, even though continued use of them is discouraged.

Registered entries may also be updated in two other ways: additional references, obsoleting references, and descriptions may be added.

All changes are subject to expert review, except for changes to registrations in a sub-namespace which are subject to the rules of the relevant sub-namespace. The submitter of a change request need not be the same as the original submitter.

Registrations may be modified by addition, but under no circumstance may any fields be modified except for the Status field or Contact Address, or to correct for transcription errors in filing or processing registration requests.

The IANA SHALL add a field describing the date that a an addition or modification was made, and a description of the change.

9. Security Considerations

General security considerations relating to IANA registration services apply; see [RFC5226].

Also, expert reviewers should look for and may document security related issues with submitters' GSS-API extensions, to the best of the reviewers' ability given the information furnished by the submitter. Reviewers may add comments regarding their limited ability to review a submission for security problems if the submitter is unwilling to provide sufficient documentation.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <http://www.rfc-editor.org/info/rfc2743>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

10.2. Informative References

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, DOI 10.17487/RFC1964, June 1996, <http://www.rfc-editor.org/info/rfc1964>.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, DOI 10.17487/RFC2744, January 2000, <http://www.rfc-editor.org/info/rfc2744>.

[RFC2853] Kabat, J. and M. Upadhyay, "Generic Security Service API Version 2 : Java Bindings", RFC 2853, DOI 10.17487/RFC2853, June 2000, <http://www.rfc-editor.org/info/rfc2853>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <http://www.rfc-editor.org/info/rfc4121>.

Authors' Addresses

Nicolas Williams
Cryptonector LLC

Email: nico@cryptonector.com

Alexey Melnikov
Isode Ltd
5 Castle Business Village
36 Station Road
Hampton, Middlesex TW12 2BX
UK

Email: Alexey.Melnikov@isode.com

NETWORK WORKING GROUP
Internet-Draft
Updates: 4120,4121 (if approved)
Intended status: Standards Track
Expires: October 1, 2017

B. Kaduk, Ed.
Akamai
J. Schaad, Ed.
Soaring Hawk Consulting
L. Zhu
Microsoft Corporation
J. Altman
Secure Endpoints
March 30, 2017

Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API (IAKERB)
draft-ietf-kitten-iakerb-03

Abstract

This document defines extensions to the Kerberos protocol and the GSS-API Kerberos mechanism that enable a GSS-API Kerberos client to exchange messages with the KDC by using the GSS-API acceptor as a proxy, encapsulating the Kerberos messages inside GSS-API tokens. With these extensions a client can obtain Kerberos tickets for services where the KDC is not accessible to the client, but is accessible to the application server.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

When authenticating using Kerberos V5, clients obtain tickets from a KDC and present them to services. This model of operation cannot work if the client does not have access to the KDC. For example, in remote access scenarios, the client must initially authenticate to an access point in order to gain full access to the network. Here the client may be unable to directly contact the KDC either because it does not have an IP address, or the access point packet filter does not allow the client to send packets to the Internet before it authenticates to the access point. The Initial and Pass Through Authentication Using Kerberos (IAKERB) mechanism allows for the use of Kerberos in such scenarios where the client is unable to directly contact the KDC, by using the service to pass messages between the client and the KDC. This allows the client to obtain tickets from the KDC and present them to the service, as in normal Kerberos operation.

Recent advancements in extending Kerberos permit Kerberos authentication to complete with the assistance of a proxy. The

Kerberos [RFC4120] pre-authentication framework [RFC6113] prevents the exposure of weak client keys over the open network. The Kerberos support of anonymity [RFC6112] provides for privacy and further complicates traffic analysis. The kdc-referrals option defined in [RFC6113] may reduce the number of messages exchanged while obtaining a ticket to exactly two even in cross-realm authentications.

Building upon these Kerberos extensions, this document extends [RFC4120] and [RFC4121] such that the client can communicate with the KDC using a Generic Security Service Application Program Interface (GSS-API) [RFC2743] acceptor as a message-passing proxy. (This is completely unrelated to the type of proxy specified in [RFC4120].) The client acts as a GSS-API initiator, and the GSS-API acceptor relays the KDC request and reply messages between the client and the KDC, transitioning to normal [RFC4121] GSS-krb5 messages once the client has obtained the necessary credentials. Consequently, IAKERB as defined in this document requires the use of the GSS-API.

The GSS-API acceptor, when relaying these Kerberos messages, is called an IAKERB proxy.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. GSS-API Encapsulation

The GSS-API mechanism Objection Identifier (OID) for IAKERB is id-kerberos-iakerb:

```
id-kerberos-iakerb ::=
{ iso(1) org(3) dod(6) internet(1) security(5) kerberosV5(2)
iakerb(5) }
```

All context establishment tokens of IAKERB MUST have the token framing described in section 4.1 of [RFC4121] with the mechanism OID being id-kerberos-iakerb. MIT implemented an earlier draft of this specification; details on how to interoperate with that implementation can be found in Appendix A.

The client starts by constructing a ticket request, as if it is being made directly to the KDC. Instead of contacting the KDC directly, the client encapsulates the request message into the output token of the GSS_Init_security_context() call and returns GSS_S_CONTINUE_NEEDED [RFC2743], indicating that at least one more token is required in order to establish the context. The output

token is then passed over the application protocol for use as the input token to the GSS_Accept_sec_context() call in accordance with GSS-API. The GSS-API acceptor extracts the Kerberos request from the input token, locates the target KDC, and sends the request on behalf of the client. After receiving the KDC reply, the GSS-API acceptor then encapsulates the reply message into the output token of GSS_Accept_sec_context(). The GSS-API acceptor returns GSS_S_CONTINUE_NEEDED [RFC2743] indicating that at least one more token is required in order to establish the context. The output token is passed to the initiator over the application protocol in accordance with GSS-API.

```
+--------+           +--------------+           +-----+
|        | --------> |              | --------> |     |
| Client |           | IAKERB proxy |           | KDC |
|        | <-------- |              | <-------- |     |
+--------+           +--------------+           +-----+
```

For all context tokens generated by the IAKERB mechanism, the innerToken described in section 4.1 of [RFC4121] has the following format: it starts with a two-octet token-identifier (TOK_ID), which is followed by an IAKERB message or a Kerberos message.

Only one IAKERB specific message, namely the IAKERB_PROXY message, is defined in this document. The TOK_ID values for Kerberos messages are the same as defined in [RFC4121].

Token	TOK_ID Value in Hex
IAKERB_PROXY	05 01

The content of the IAKERB_PROXY message is defined as an IAKERB-HEADER structure immediately followed by a Kerberos message, which is optional. The Kerberos message can be an AS-REQ, an AS-REP, a TGS-REQ, a TGS-REP, or a KRB-ERROR as defined in [RFC4120].

```
IAKERB-HEADER ::= SEQUENCE {
    -- Note that the tag numbers start at 1, not 0, which would
    -- be more conventional for Kerberos.
    target-realm      [1] UTF8String,
       -- The name of the target realm.
    cookie            [2] OCTET STRING OPTIONAL,
       -- Opaque data, if sent by the server,
       -- MUST be copied by the client verbatim into
       -- the next IAKRB_PROXY message.
    ...
}
```

The IAKERB-HEADER structure and all the Kerberos messages MUST be encoded using Abstract Syntax Notation One (ASN.1) Distinguished Encoding Rules (DER) [CCITT.X680.2002] [CCITT.X690.2002].

The client fills out the IAKERB-HEADER structure as follows: the target-realm contains the realm name the ticket request is addressed to. In the initial message from the client, the cookie field is absent. The client MAY send a completely empty IAKERB_PROXY message (consisting solely of the octets 05 01 and an IAKERB_HEADER with zero-length target-realm) in order to query the Kerberos realm of the acceptor, see Section 3.1. In all other cases, the client MUST specify a target-realm. This can be the realm of the client’s host, if no other realm information is available. client’s host.

Upon receipt of the IAKERB_PROXY message, the GSS-API acceptor inspects the target-realm field in the IAKERB_HEADER, locates a KDC for that realm, and sends the ticket request to that KDC. The IAKERB proxy MAY engage in fallback behavior, retransmitting packets to a given KDC and/or sending the request to other KDCs in that realm if the initial transmission does not receive a reply, as would be done if the proxy was making requests on its own behalf.

The GSS-API acceptor encapsulates the KDC reply message in the returned IAKERB message. It fills out the target realm using the realm sent by the client and the KDC reply message is included immediately following the IAKERB-HEADER header.

When the GSS-API acceptor is unable to obtain an IP address for a KDC in the client’s realm, it sends a KRB_ERROR message with the code KRB_AP_ERR_IAKERB_KDC_NOT_FOUND to the client in place of an actual reply from the KDC, and the context fails to establish. There is no accompanying error data defined in this document for this error code.

```
KRB_AP_ERR_IAKERB_KDC_NOT_FOUND      85
    -- The IAKERB proxy could not find a KDC.
```

When the GSS-API acceptor has an IP address for at least one KDC in the target realm, but does not receive a response from any KDC in the realm (including in response to retries), it sends a KRB_ERROR message with the code KRB_AP_ERR_IAKERB_KDC_NO_RESPONSE to the client and the context fails to establish. There is no accompanying error data defined in this document for this error code.

```
KRB_AP_ERR_IAKERB_KDC_NO_RESPONSE    86
    -- The KDC did not respond to the IAKERB proxy.
```

The IAKERB proxy can send opaque data in the cookie field of the IAKERB-HEADER structure in the server reply to the client, in order

to, for example, minimize the amount of state information kept by the GSS-API acceptor. The content and the encoding of the cookie field is a local matter of the IAKERB proxy. Whenever the cookie is present in a token received by the initiator, the initiator MUST copy the cookie verbatim into its subsequent response tokens which contain IAKERB_PROXY messages.

The client and the server can repeat the sequence of sending and receiving the IAKERB messages as described above for an arbitrary number of message exchanges, in order to allow the client to interact with the KDC through the IAKERB proxy, and to obtain Kerberos tickets as needed to authenticate to the acceptor.

Once the client has obtained the service ticket needed to authenticate to the acceptor, subsequent GSS-API context tokens are of type KRB_AP_REQ, not IAKERB_PROXY, and the client performs the client-server application exchange as defined in [RFC4120] and [RFC4121].

For implementations conforming to this specification, both the authenticator subkey and the GSS_EXTS_FINISHED extension as defined in Section 4 MUST be present in the AP-REQ authenticator. This checksum provides integrity protection for the IAKERB messages previously exchanged, including the unauthenticated clear texts in the IAKERB-HEADER structure.

If the pre-authentication data is encrypted in the long-term password-based key of the principal, the risk of security exposures is significant. Implementations SHOULD utilize the AS_REQ armoring as defined in [RFC6113] unless an alternative protection is deployed. In addition, the anonymous Kerberos FAST option is RECOMMENDED for the client to complicate traffic analysis.

3.1. Enterprise principal names

The introduction of principal name canonicalization by [RFC6806] created the possibility for a client to have a principal name (of type NT-ENTERPRISE) for which it is trying to obtain credentials, but no information about what realm’s KDC to contact to obtain those credentials. A Kerberos client not using IAKERB would typically resolve the NT-ENTERPRISE name to a principal name by starting from the realm of the client’s host and finding out the true realm of the enterprise principal based on referrals [RFC6806].

A client using IAKERB may not have any realm information, even for the realm of the client’s host, or may know that the client host’s realm is not appropriate for a given enterprise principal name. In such cases, the client can retrieve the realm of the GSS-API acceptor

as follows: the client returns GSS_S_CONTINUE_NEEDED with the output token containing an IAKERB message with an empty target-realm in the IAKERB-HEADER and no Kerberos message following the IAKERB-HEADER structure. Upon receipt of the realm request, the GSS-API acceptor fills out an IAKERB_PROXY response message, filling the target-realm field with the realm of the acceptor, and returns GSS_S_CONTINUE_NEEDED with the output token containing the IAKERB message with the server's realm and no Kerberos message following the IAKERB-HEADER header. The GSS-API initiator can then use the returned realm in subsequent IAKERB messages to resolve the NT-ENTERPRISE name type. Since the GSS-API acceptor can act as a Kerberos acceptor, it always has an associated Kerberos realm.

4. Finish Message

For implementations conforming to this specification, the authenticator subkey in the AP-REQ MUST alway be present, and the Exts field in the GSS-API authenticator [RFC6542] MUST contain an extension of type GSS_EXTS_FINISHED with extension data containing the ASN.1 DER encoding of the structure KRB-FINISHED.

```
GSS_EXTS_FINISHED             2
     --- Data type for the IAKERB checksum.

KRB-FINISHED ::= {
     -- Note that the tag numbers start at 1, not 0, which would be
     -- more conventional for Kerberos.
     gss-mic [1] Checksum,
         -- Contains the checksum [RFC3961] of the GSS-API tokens
         -- exchanged between the initiator and the acceptor,
         -- and prior to the containing AP_REQ GSS-API token.
         -- The checksum is performed over the GSS-API tokens
         -- exactly as they were transmitted and received,
         -- in the order that the tokens were sent.
      ...
}
```

The gss-mic field in the KRB-FINISHED structure contains a Kerberos checksum [RFC3961] of all the preceding context tokens of this GSS-API context (including the generic token framing of the GSSAPI-Token type from [RFC4121]), concatenated in chronological order (note that GSS-API context token exchanges are synchronous). The checksum type is the required checksum type of the enctype of the subkey in the authenticator, the protocol key for the checksum operation is the authenticator subkey, and the key usage number is KEY_USAGE_FINISHED.

```
KEY_USAGE_FINISHED            41
```

The GSS-API acceptor MUST then verify the checksum contained in the GSS_EXTS_FINISHED extension. This checksum provides integrity protection for the messages exchanged including the unauthenticated clear texts in the IAKERB-HEADER structure.

5. Addresses in Tickets

In IAKERB, the machine sending requests to the KDC is the GSS-API acceptor and not the client. As a result, the client should not include its addresses in any KDC requests for two reasons. First, the KDC may reject the forwarded request as being from the wrong client. Second, in the case of initial authentication for a dial-up client, the client machine may not yet possess a network address. Hence, as allowed by [RFC4120], the addresses field of the AS-REQ and TGS-REQ requests SHOULD be blank.

6. Security Considerations

The IAKERB proxy is a man-in-the-middle for the client’s Kerberos exchanges. The Kerberos protocol is designed to be used over an untrusted network, so this is not a critical flaw, but it does expose to the IAKERB proxy all information sent in cleartext over those exchanges, such as the principal names in requests. Since the typical usage involves the client obtaining a service ticket for the service operating the proxy, which will receive the client principal as part of normal authentication, this is also not a serious concern. However, an IAKERB client not using an armored FAST channel [RFC6113] sends an AS_REQ with pre-authentication data encrypted in the long-term keys of the user, even before the acceptor is authenticated. This subjects the user’s long-term key to an offline attack by the proxy. To mitigate this threat, the client SHOULD use FAST [RFC6113] and its KDC authentication facility to protect the user’s credentials.

Similarly, the client principal name is in cleartext in the AS and TGS exchanges, whereas in the AP exchanges embedded in GSS context tokens for the regular krb5 mechanism, the client principal name is present only in encrypted form. Thus, more information is exposed over the network path between initiator and acceptor when IAKERB is used than when the krb5 mechanism is used, unless FAST armor is employed. (This information would be exposed in other traffic from the initiator when the krb5 mech is used.) As such, to complicate traffic analysis and provide privacy for the client, the client SHOULD request the anonymous Kerberos FAST option [RFC6113].

Similar to other network access protocols, IAKERB allows an unauthenticated client (possibly outside the security perimeter of an organization) to send messages that are proxied to servers inside the

perimeter. To reduce the attack surface, firewall filters can be applied to restrict from which hosts client requests can be proxied, and the proxy can further restrict the set of realms to which requests can be proxied.

In the intended use scenario, the client uses the proxy to obtain a TGT and then a service ticket for the service it is authenticating to (possibly preceeded by exchanges to produce FAST armor). However, the protocol allows arbitrary KDC-REQs to be passed through, and there is no limit to the number of exchanges that may be proxied. The client can send KDC-REQs unrelated to the current authentication, and obtain service tickets for other service principals in the database of the KDC being contacted.

In a scenario where DNS SRV RR’s are being used to locate the KDC, IAKERB is being used, and an external attacker can modify DNS responses to the IAKERB proxy, there are several countermeasures to prevent arbitrary messages from being sent to internal servers:

1. KDC port numbers can be statically configured on the IAKERB proxy. In this case, the messages will always be sent to KDC’s. For an organization that runs KDC’s on a static port (usually port 88) and does not run any other servers on the same port, this countermeasure would be easy to administer and should be effective.

2. The proxy can do application level sanity checking and filtering. This countermeasure should eliminate many of the above attacks.

3. DNS security can be deployed. This countermeasure is probably overkill for this particular problem, but if an organization has already deployed DNS security for other reasons, then it might make sense to leverage it here. Note that Kerberos could be used to protect the DNS exchanges. The initial DNS SRV KDC lookup by the proxy will be unprotected, but an attack here is at most a denial of service (the initial lookup will be for the proxy’s KDC to facilitate Kerberos protection of subsequent DNS exchanges between itself and the DNS server).

7. Acknowledgements

Jonathan Trostle, Michael Swift, Bernard Aboba and Glen Zorn wrote earlier revision of this document.

The hallway conversations between Larry Zhu and Nicolas Williams formed the basis of this document.

8. Assigned Numbers

The value for the error code KRB_AP_ERR_IAKERB_KDC_NOT_FOUND is 85.

The value for the error code KRB_AP_ERR_IAKERB_KDC_NO_RESPONSE is 86.

The key usage number KEY_USAGE_FINISHED is 41.

The key usage number KEY_USAGE_IAKERB_FINISHED is 42.

9. IANA Considerations

IANA is requested to make a modification in the "Kerberos GSS-API Token Type Identifiers" registry.

The following data to the table:

ID	Description	Reference
05 01	IAKERB_PROXY	[THIS RFC]

10. References

10.1. Normative References

[CCITT.X680.2002]
International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.

[CCITT.X690.2002]
International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <http://www.rfc-editor.org/info/rfc2743>.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, DOI 10.17487/RFC3961, February 2005, <http://www.rfc-editor.org/info/rfc3961>.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <http://www.rfc-editor.org/info/rfc4121>.

[RFC6542] Emery, S., "Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Channel Binding Hash Agility", RFC 6542, DOI 10.17487/RFC6542, March 2012, <http://www.rfc-editor.org/info/rfc6542>.

10.2. Informative references

[RFC6112] Zhu, L., Leach, P., and S. Hartman, "Anonymity Support for Kerberos", RFC 6112, DOI 10.17487/RFC6112, April 2011, <http://www.rfc-editor.org/info/rfc6112>.

[RFC6113] Hartman, S. and L. Zhu, "A Generalized Framework for Kerberos Pre-Authentication", RFC 6113, DOI 10.17487/RFC6113, April 2011, <http://www.rfc-editor.org/info/rfc6113>.

[RFC6806] Hartman, S., Ed., Raeburn, K., and L. Zhu, "Kerberos Principal Name Canonicalization and Cross-Realm Referrals", RFC 6806, DOI 10.17487/RFC6806, November 2012, <http://www.rfc-editor.org/info/rfc6806>.

Appendix A. Interoperate with Previous MIT version

MIT implemented an early draft version of this document. This section gives a method for detecting and interoperating with that version.

Initiators behave as follows:

- o If the first acceptor token begins with generic token framing as described in section 3.1 of [RFC2743], then use the protocol as defined in this document.

- o If the first acceptor token is missing the generic token framing (i.e., the token begins with the two-byte token ID 05 01), then

  - * When creating the finish message, the value of one (1) should be used in place of GSS_EXTS_FINISHED.

  - * When computing the checksum, the value of KEY_USAGE_IAKERB_FINISHED should be used in place of KEY_USAGE_FINISHED.

```
KEY_USAGE_IAKERB_FINISHED            42
```

Acceptors behave as follows:

- o After the first initiator token, allow initiator tokens to omit generic token framing. This allowance is required only for IAKERB_PROXY messages (those using token ID 05 01), not for tokens defined in [RFC4121].

- o If the AP-REQ authenticator contains an extension of type 1 containing a KRB-FINISHED message, then process the extension as if it were of type GSS_EXTS_FINISHED, except with a key usage of KEY_USAGE_IAKERB_FINISHED (42) instead of KEY_USAGE_FINISHED (41).

Authors' Addresses

Benjamin Kaduk (editor)
Akamai

Email: kaduk@mit.edu

Jim Schaad (editor)
Soaring Hawk Consulting

Email: ietf@augustcellars.com

Larry Zhu
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

Email: lzhu@microsoft.com

Jeffery Altman
Secure Endpoints
255 W 94th St
New York, NY 10025
US

Email: jaltman@secure-endpoints.com

Internet Engineering Task Force
Internet-Draft
Updates: 4120 (if approved)
Intended status: Standards Track
Expires: August 13, 2017

A. Jain
Georgia Tech
N. Kinder
N. McCallum
Red Hat, Inc.

February 9, 2017

Authentication Indicator in Kerberos Tickets
draft-ietf-kitten-krb-auth-indicator-07

Abstract

This document updates RFC 4120 in order to specify an extension in the Kerberos protocol. It defines a new authorization data type AD-AUTHENTICATION-INDICATOR. The purpose of introducing this data type is to include an indicator of the strength of a client's authentication in service tickets so that application services can use it as an input into policy decisions.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Kerberos [RFC4120] allows secure interaction among users and services over a network. It supports a variety of authentication mechanisms using its pre-authentication framework [RFC6113]. The Kerberos authentication service has been architected to support password-based authentication as well as multi-factor authentication using one-time password devices, public-key cryptography and other pre-authentication schemes. Implementations that offer pre-authentication mechanisms supporting significantly different strengths of client authentication may choose to keep track of the strength of the authentication that was used, for use as an input into policy decisions.

This document specifies a new authorization data type to convey authentication strength information to application services. Elements of this type appear within an AD-CAMMAC (authorization data type Container Authenticated by Multiple Message Authentication Codes) [RFC7751] container.

2. Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. AD Type Specification

The Key Distribution Center (KDC) MAY include authorization data of ad-type 97, wrapped in AD-CAMMAC, in initial credentials. The KDC MAY copy it from a ticket-granting ticket into service tickets.

The corresponding ad-data field contains the DER encoding [X.690] of the following ASN.1 [X.680] type:

```
AD-AUTHENTICATION-INDICATOR ::= SEQUENCE OF UTF8String
```

Each UTF8String value is a short string that indicates that a particular set of requirements was met during the initial authentication. These strings are intended to be compared against known values. They are not intended to store structured data. Each string MUST be either:

* A URI which references a Level of Assurance Profile [RFC6711]

* A site-defined string, which MUST NOT contain a colon, whose meaning is determined by the realm administrator.

Authorization data elements of type AD-AUTHENTICATION-INDICATOR MUST be included in an AD-CAMMAC container so that their contents can be verified as originating from the KDC. Elements of type AD-AUTHENTICATION-INDICATOR MAY safely be ignored by applications and KDCs that do not implement this element.

4. Assigned Numbers

RFC 4120 [RFC4120] is updated in the following way:

- o The ad-type number 97 is assigned for AD-AUTHENTICATION-INDICATOR, updating the table in Section 7.5.4 of RFC 4120 [RFC4120].

- o The table in Section 5.2.6 of RFC 4120 [RFC4120] is updated to map the ad-type 97 to "DER encoding of AD-AUTHENTICATION-INDICATOR".

5. Security Considerations

Elements of type AD-AUTHENTICATION-INDICATOR are wrapped in AD-CAMMAC containers. AD-CAMMAC supersedes AD-KDC-ISSUED, and allows both application services and the KDC to verify the authenticity of the contained authorization data.

KDC implementations MUST use AD-CAMMAC verifiers as described in the the security considerations of RFC 7751 [RFC7751] to ensure that AD-AUTHENTICATION-INDICATOR elements are not modified by an attacker.

Application servers MUST validate the AD-CAMMAC container before making authorization decisions based on AD-AUTHENTICATION-INDICATOR elements. Application servers MUST NOT make authorization decisions based on AD-AUTHENTICATION-INDICATOR elements which appear outside of AD-CAMMAC containers.

Using multiple strings in AD-AUTHENTICATION-INDICATOR may lead to ambiguity when a service tries to make a decision based on the AD-AUTHENTICATION-INDICATOR values. This ambiguity can be avoided if indicator values are always used as a positive indication of certain requirements being met during the initial authentication. For example, if a "without-password" indicator is inserted whenever authentication occurs without a password, a service might assume this is an indication that a higher-strength client authentication occurred. However, this indicator might also be inserted when no authentication occurred at all (such as anonymous PKINIT).

Service evaluation of site-defined indicators MUST consider the realm of original authentication in order to avoid cross-realm indicator collisions. Failure to enforce this property can result in invalid authorization.

6. IANA Considerations

This document has no actions for IANA.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC6113] Hartman, S. and L. Zhu, "A Generalized Framework for Kerberos Pre-Authentication", RFC 6113, DOI 10.17487/RFC6113, April 2011, <http://www.rfc-editor.org/info/rfc6113>.

[RFC7751] Sorce, S. and T. Yu, "Kerberos Authorization Data Container Authenticated by Multiple Message Authentication Codes (MACs)", RFC 7751, DOI 10.17487/RFC7751, March 2016, <http://www.rfc-editor.org/info/rfc7751>.

[X.680] ITU-T, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation -- ITU-T Recommendation X.680 (ISO/IEC International Standard 8824-1:2008)", 2008.

[X.690] ITU-T, "Information technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) -- ITU-T Recommendation X.690 (ISO/IEC International Standard 8825-1:2008)", 2008.

7.2. Informative References

[RFC6711] Johansson, L., "An IANA Registry for Level of Assurance (LoA) Profiles", RFC 6711, DOI 10.17487/RFC6711, August 2012, <http://www.rfc-editor.org/info/rfc6711>.

Appendix A. ASN.1 Module

```
KerberosV5AuthenticationIndicators {
        iso(1) identified-organization(3) dod(6) internet(1)
        security(5) kerberosV5(2) modules(4)
        authentication-indicators(9)
} DEFINITIONS EXPLICIT TAGS ::= BEGIN

AD-AUTHENTICATION-INDICATOR ::= SEQUENCE OF UTF8String

END
```

Appendix B. Acknowledgements

Dmitri Pal (Red Hat)
Simo Sorce (Red Hat)
Greg Hudson (MIT)

Authors' Addresses

Anupam Jain
Georgia Tech
225 North Ave NW
Atlanta, GA 30332
USA

EMail: ajain323@gatech.edu

Nathan Kinder
Red Hat, Inc.
444 Castro St.
Suite 500
Mountain View, CA 94041
USA

EMail: nkinder@redhat.com

Nathaniel McCallum
Red Hat, Inc.
100 East Davie Street
Raleigh, NC 27601
USA

EMail: npmccallum@redhat.com

Kitten Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 24, 2016

M. Short, Ed.
S. Moore
P. Miller
Microsoft Corporation

May 23, 2016

Public Key Cryptography for Initial Authentication in Kerberos (PKINIT) Freshness Extension
draft-ietf-kitten-pkinit-freshness-07

Abstract

This document describes how to further extend the Public Key Cryptography for Initial Authentication in Kerberos (PKINIT) extension [RFC4556] to exchange an opaque data blob that a KDC can validate to ensure that the client is currently in possession of the private key during a PKINIT AS exchange.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Kerberos PKINIT extension [RFC4556] defines two schemes for using asymmetric cryptography in a Kerberos preauthenticator. One uses Diffie-Hellman key exchange and the other depends on public key encryption. The public key encryption scheme is less commonly used for two reasons:

- o Elliptic Curve Cryptography (ECC) Support for PKINIT [RFC5349] only specified Elliptic Curve Diffie-Hellman (ECDH) key agreement, so it cannot be used for public key encryption.

- o Public key encryption requires certificates with an encryption key, that is not deployed on many existing smart cards.

In the Diffie-Hellman exchange, the client uses its private key only to sign the AuthPack structure (specified in Section 3.2.1 of [RFC4556]), that is performed before any traffic is sent to the KDC. Thus a client can generate requests with future times in the PKAuthenticator, and then send those requests at those future times. Unless the time is outside the validity period of the client's certificate, the KDC will validate the PKAuthenticator and return a TGT the client can use without possessing the private key.

As a result, a client performing PKINIT with the Diffie-Hellman key exchange does not prove current possession of the private key being used for authentication. It proves only prior use of that key. Ensuring that the client has current possession of the private key requires that the signed PKAuthenticator data include information that the client could not have predicted.

1.1. Kerberos message flow using KRB_AS_REQ without pre-authentication

Today, password-based AS exchanges [RFC4120] often begin with the client sending a KRB_AS_REQ without pre-authentication. When the principal requires pre-authentication, the KDC responds with a KRB_ERROR containing information needed to complete an AS exchange, such as the supported encryption types and salt values. This message flow is illustrated below:

```
KDC                     Client

              <----     AS-REQ without pre-authentication
KRB-ERROR     ---->

              <----     AS-REQ
AS-REP        ---->

              <----     TGS-REQ
TGS-REP       ---->
```

Figure 1

We can use a similar message flow with PKINIT, allowing the KDC to provide a token for the client to include in its KRB_AS_REQ to ensure that the PA_PK_AS_REQ [RFC4556] was not pregenerated.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Message Exchanges

The following summarizes the message flow with extensions to [RFC4120] and [RFC4556] required to support a KDC-provided freshness token during the initial request for a ticket:

1. The client generates a KRB_AS_REQ as specified in Section 2.9.3 of [RFC4120] that contains no PA_PK_AS_REQ and includes a freshness token request.

2. The KDC generates a KRB_ERROR as specified in Section 3.1.3 of [RFC4120] providing a freshness token.

3. The client receives the error as specified in Section 3.1.4 of [RFC4120], extracts the freshness token, and includes it as part of the KRB_AS_REQ as specified in [RFC4120] and [RFC4556].

4. The KDC receives and validates the KRB_AS_REQ as specified in Section 3.2.2 of [RFC4556], then additionally validates the freshness token.

5. The KDC and client continue as specified in [RFC4120] and [RFC4556].

2.1. Generation of KRB_AS_REQ Message

The client indicates support of freshness tokens by adding a padata element with padata-type PA_AS_FRESHNESS and padata-value of an empty octet string.

2.2. Generation of KRB_ERROR Message

The KDC will respond with a KRB_ERROR [RFC4120] message with the error-code KDC_ERR_PREAUTH_REQUIRED [RFC4120] adding a padata element with padata-type PA_AS_FRESHNESS and padata-value of the freshness token to the METHOD-DATA object.

2.3. Generation of KRB_AS_REQ Message

After the client receives the KRB-ERROR message containing a freshness token, it extracts the PA_AS_FRESHNESS padata-value field of the PA-DATA structure as an opaque data blob. The PA_AS_FRESHNESS padata-value field of the PA-DATA structure SHALL then be added as an opaque blob in the freshnessToken field when the client generates the PKAuthenticator specified in Section 4 for the PA_PK_AS_REQ message. This ensures that the freshness token value will be included in the signed data portion of the KRB_AS_REQ value.

2.4. Receipt of KRB_AS_REQ Message

If the realm requires freshness and the PA_PK_AS_REQ message does not contain the freshness token, the KDC MUST return a KRB_ERROR [RFC4120] message with the error-code KDC_ERR_PREAUTH_FAILED [RFC4120] with a padata element with padata-type PA_AS_FRESHNESS and padata-value of the freshness token to the METHOD-DATA object.

When the PA_PK_AS_REQ message contains a freshness token, after validating the PA_PK_AS_REQ message normally, the KDC will validate

the freshnessToken value in the PKAuthenticator in an implementation-specific way. If the freshness token is not valid, the KDC MUST return a KRB_ERROR [RFC4120] message with the error-code KDC_ERR_PREAUTH_EXPIRED [RFC6113]. The e-data field of the error contains a METHOD-DATA object [RFC4120] which specifies a valid PA_AS_FRESHNESS padata-value. Since the freshness tokens are validated by KDCs in the same realm, standardizing the contents of the freshness token is not a concern for interoperability.

2.5. Receipt of second KRB_ERROR Message

If a client receives a KDC_ERR_PREAUTH_EXPIRED KRB_ERROR message that includes a freshness token, it SHOULD retry using the new freshness token.

3. PreAuthentication Data Types

The following are the new PreAuthentication data types:

Padata and Data Type	Padata-type Value
PA_AS_FRESHNESS	150

4. Extended PKAuthenticator

The PKAuthenticator structure specified in Section 3.2.1 of [RFC4556] is extended to include a new freshnessToken as follows:

```
PKAuthenticator ::= SEQUENCE {
   cusec        [0] INTEGER (0..999999),
   ctime        [1] KerberosTime,
             -- cusec and ctime are used as in [RFC4120], for
             -- replay prevention.
   nonce        [2] INTEGER (0..4294967295),
             -- Chosen randomly;  this nonce does not need to
             -- match with the nonce in the KDC-REQ-BODY.
   paChecksum   [3] OCTET STRING OPTIONAL,
             -- MUST be present.
             -- Contains the SHA1 checksum, performed over
             -- KDC-REQ-BODY.
   ...,
   freshnessToken     [4] OCTET STRING OPTIONAL,
             -- PA_AS_FRESHNESS padata value as recieved from the
             -- KDC. MUST be present if sent by KDC
   ...
}
```

5. Acknowledgements

Douglas E. Engert, Sam Hartman, Henry B. Hotz, Nikos Mavrogiannopoulos, Martin Rex, Nico Williams, and Tom Yu were key contributors to the discovery of the freshness issue in PKINIT.

Sam Hartman, Greg Hudson, Jeffrey Hutzelman, Nathan Ide, Benjamin Kaduk, Bryce Nordgren, Magnus Nystrom, Nico Williams and Tom Yu reviewed the document and provided suggestions for improvements.

6. IANA Considerations

IANA is requested to assign numbers for PA_AS_FRESHNESS listed in the Kerberos Parameters registry Pre-authentication and Typed Data as follows:

Type	Value	Reference
150	PA_AS_FRESHNESS	[This RFC]

7. Security Considerations

The freshness token SHOULD include signing, encrypting or sealing data from the KDC to determine authenticity and prevent tampering.

Freshness tokens serve to guarantee that the client had the key when constructing the AS-REQ. They are not required to be single use tokens or bound to specific AS exchanges. Part of the reason the token is opaque is to allow KDC implementers the freedom to add additional functionality as long as the "freshness" guarantee remains.

8. Interoperability Considerations

Since the client treats the KDC-provided data blob as opaque, changing the contents will not impact existing clients. Thus extensions to the freshness token do not impact client interoperability.

Clients SHOULD NOT reuse freshness tokens across multiple exchanges. There is no guarantee that a KDC will allow a once-valid token to be used again. Thus clients that do not retry with a new freshness token may not be compatible with KDCs, depending on how they choose to implement freshness validation.

Since upgrading clients takes time, implementers may consider allowing both freshness-token based exchanges and "legacy" exchanges without use of freshness tokens. However, until freshness tokens are required by the realm, the existing risks of pre-generated PKAuthenticators will remain.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC4556] Zhu, L. and B. Tung, "Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 4556, DOI 10.17487/RFC4556, June 2006, <http://www.rfc-editor.org/info/rfc4556>.

[RFC5349] Zhu, L., Jaganathan, K., and K. Lauter, "Elliptic Curve Cryptography (ECC) Support for Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 5349, DOI 10.17487/RFC5349, September 2008, <http://www.rfc-editor.org/info/rfc5349>.

[RFC6113] Hartman, S. and L. Zhu, "A Generalized Framework for Kerberos Pre-Authentication", RFC 6113, DOI 10.17487/RFC6113, April 2011, <http://www.rfc-editor.org/info/rfc6113>.

Authors' Addresses

Michiko Short (editor)
Microsoft Corporation
USA

Email: michikos@microsoft.com

Seth Moore
Microsoft Corporation
USA

Email: sethmo@microsoft.com

Paul Miller
Microsoft Corporation
USA

Email: paumil@microsoft.com

NETWORK WORKING GROUP
Internet-Draft
Obsoletes: 4402 (if approved)
Intended status: Standards Track
Expires: June 13, 2016

S. Emery
Oracle
N. Williams
Cryptonector

December 11, 2015

A Pseudo-Random Function (PRF) for the Kerberos V Generic Security Service Application Program Interface (GSS-API) Mechanism
draft-ietf-kitten-rfc4402bis-02

Abstract

This document defines the Pseudo-Random Function (PRF) for the Kerberos V mechanism for the Generic Security Service Application Program Interface (GSS-API), based on the PRF defined for the Kerberos V cryptographic framework, for keying application protocols given an established Kerberos V GSS-API security context.

This document obsoletes RFC 4402 and reclassifies that document as historic. RFC 4402 starts the PRF+ counter at 1, however a number of implementations starts the counter at 0. As a result, the original specification would not be interoperable with existing implementations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document specifies the Kerberos V GSS-API mechanism's [RFC4121] pseudo-random function corresponding to [RFC4401]. The function is a "PRF+" style construction. For more information see [RFC4401], [RFC2743], [RFC2744] and [RFC4121].

This document obsoletes RFC 4402 and reclassifies that document as historic. RFC 4402 starts the PRF+ counter at 1, however a number of implementations starts the counter at 0. As a result, the original specification would not be interoperable with existing implementations.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Kerberos V GSS Mechanism PRF

The GSS-API PRF [RFC4401] function for the Kerberos V mechanism [RFC4121] shall be the output of a PRF+ function based on the encryption type's PRF function keyed with the negotiated session key of the security context corresponding to the 'prf_key' input parameter of GSS_Pseudo_random().

This PRF+ MUST be keyed with the key indicated by the 'prf_key' input parameter as follows:

- o GSS_C_PRF_KEY_FULL -- use the sub-session key asserted by the acceptor, if any exists, or the sub-session asserted by the initiator, if any exists, or the Ticket's session key

- o GSS_C_PRF_KEY_PARTIAL -- use the sub-session key asserted by the initiator, if any exists, or the Ticket's session key

The PRF+ function is a simple counter-based extension of the Kerberos V pseudo-random function [RFC3961] for the encryption type of the security context's keys:

```
PRF+(K, L, S) = truncate(L, T0 || T1 || .. || Tn)

Tn = pseudo-random(K, n || S)
```

where K is the key indicated by the 'prf_key' parameter, where '||' is the concatenation operator, 'n' is encoded as a network byte order 32-bit unsigned binary number, truncate(L, S) truncates the input octet string S to length L, and pseudo-random() is the Kerberos V pseudo-random function [RFC3961].

The maximum output size of the Kerberos V mechanism's GSS-API PRF then is, necessarily, 2^32 times the output size of the pseudo-random() function for the encryption type of the given key.

When the input size is longer than 2^14 octets as per [RFC4401] and exceeds an implementation's resources, then the mechanism MUST return GSS_S_FAILURE and GSS_KRB5_S_KG_INPUT_TOO_LONG as the minor status code.

4. IANA Considerations

This document has no IANA considerations currently. If and when a relevant IANA registry of GSS-API symbols and constants is created, then the GSS_KRB5_S_KG_INPUT_TOO_LONG minor status code should be added to such a registry.

5. Security Considerations

Kerberos V encryption types' PRF functions use a key derived from contexts' session keys and should preserve the forward security properties of the mechanisms' key exchanges.

Legacy Kerberos V encryption types may be weak, particularly the single-DES encryption types.

See also [RFC4401] for generic security considerations of GSS_Pseudo_random().

See also [RFC3961] for generic security considerations of the Kerberos V cryptographic framework.

Use of Ticket session keys, rather than sub-session keys, when initiators and acceptors fail to assert sub-session keys, is dangerous as ticket reuse can lead to key reuse; therefore, initiators should assert sub-session keys always, and acceptors should assert sub-session keys at least when initiators fail to do so.

The computational cost of computing this PRF+ may vary depending on the Kerberos V encryption types being used, but generally the computation of this PRF+ gets more expensive as the input and output octet string lengths grow (note that the use of a counter in the PRF+ construction allows for parallelization).

6. Acknowledgements

This document is an update to Nico Williams' RFC. Greg Hudson has provided the test vectors based on MIT's implementation.

7. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <http://www.rfc-editor.org/info/rfc2743>.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, DOI 10.17487/RFC2744, January 2000, <http://www.rfc-editor.org/info/rfc2744>.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, DOI 10.17487/RFC3961, February 2005, <http://www.rfc-editor.org/info/rfc3961>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <http://www.rfc-editor.org/info/rfc4121>.

[RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API Extension for the Generic Security Service Application Program Interface (GSS-API)", RFC 4401, DOI 10.17487/RFC4401, February 2006, <http://www.rfc-editor.org/info/rfc4401>.

Appendix A. Test Vectors

Here are some test vectors from the MIT implementation provided by Greg Hudson. Test cases used include input string lengths of 0 and 61 bytes, and an output length of 44 bytes. 61 bytes of input is just enough to produce a partial second MD5 or SHA1 hash block with the four-byte counter prefix. 44 bytes of output requires two full and one partial RFC 3961 PRF output for all existing enctypes. All keys were randomly generated.

```
Enctype: des-cbc-crc
Key: E607FE9DABB57AE0
Input: (empty string)
Output: 803C4121379FC4B87CE413B67707C4632EBED2C6D6B7
        2A55E878836E35E21600D915D590DED5B6D77BB30A1F

Enctype: des-cbc-crc
Key: 54758316B6257A75
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: 279E4105F7ADC9BD6EF28ABE31D89B442FE0058388BA
        33264ACB5729562DC637950F6BD144B654BE7700B2D6

Enctype: des3-cbc-sha1
Key: 70378A19CD64134580C27C0115D6B34A1CF2FEECEF9886A2
Input: (empty string)
Output: 9F8D127C520BB826BFF3E0FE5EF352389C17E0C073D9
        AC4A333D644D21BA3EF24F4A886D143F85AC9F6377FB

Enctype: des3-cbc-sha1
Key: 3452A167DF1094BA1089E0A20E9E51ABEF1525922558B69E
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: 6BF24FABC858F8DD9752E4FCD331BB831F238B5BE190
        4EEA42E38F7A60C588F075C5C96A67E7F8B7BD0AECF4

Enctype: rc4-hmac
Key: 3BB3AE288C12B3B9D06B208A4151B3B6
Input: (empty string)
Output: 9AEA11A3BCF3C53F1F91F5A0BA2132E2501ADF5F3C28
        3C8A983AB88757CE865A22132D6100EAD63E9E291AFA

Enctype: rc4-hmac
Key: 6DB7B33A01BD2B72F7655CB7B3D5FA0B
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: CDA9A544869FC84873B692663A82AFDA101C8611498B
        A46138B01E927C9B95EEC953B562807434037837DDDF

Enctype: aes128-cts-hmac-sha1-96
Key: 6C742096EB896230312B73972FA28B5D
```

```
Input: (empty string)
Output: 94208D982FC1BB7778128BDD77904420B45C9DA699F3
        117BCE66E39602128EF0296611A6D191A5828530F20F

Enctype: aes128-cts-hmac-sha1-96
Key: FA61138C109D834A477D24C7311BE6DA
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: 0FAEDF0F842CC834FEE750487E1B622739286B975FE5
        B7F45AB053143C75CA0DF5D3D4BBB80F6A616C7C9027

Enctype: aes256-cts-hmac-sha1-96
Key: 08FCDAFD5832611B73BA7B497FEBFF8C954B4B58031CAD9B977C3B8C25192FD6
Input: (empty string)
Output: E627EFC14EF5B6D629F830C7109DEA0D3D7D36E8CD57
        A1F301C5452494A1928F05AFFBEE3360232209D3BE0D

Enctype: aes256-cts-hmac-sha1-96
Key: F5B68B7823D8944F33F41541B4E4D38C9B2934F8D16334A796645B066152B4BE
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: 112F2B2D878590653CCC7DE278E9F0AA46FA5A380B62
        59F774CB7C134FCD37F61A50FD0D9F89BF8FE1A6B593

Enctype: camellia128-cts-cmac
Key: 866E0466A178279A32AC0BDA92B72AEB
Input: (empty string)
Output: 97FBB354BF341C3A160DCC86A7A910FDA824601DF677
        68797BACEEBF5D250AE929DEC9760772084267F50A54

Enctype: camellia128-cts-cmac
Key: D4893FD37DA1A211E12DD1E03E0F03B7
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: 1DEE2FF126CA563A2A2326B9DD3F0095013257414C83
        FAD4398901013D55F367C82681186B7B2FE62F746BA4

Enctype: camellia256-cts-cmac
Key: 203071B1AE77BD3D6FCE70174AF95C225B1CED46B35CF52B6479EFEB47E6B063
Input: (empty string)
Output: 9B30020634C10FDA28420CEE7B96B70A90A771CED43A
        D8346554163E5949CBAE2FB8EF36AFB6B32CE75116A0

Enctype: camellia256-cts-cmac
Key: A171AD582C1AFBBAD52ABD622EE6B6A14D19BF95C6914B2BA40FFD99A88EC660
Input: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789
Output: A47CBB6E104DCC77E4DB48A7A474B977F2FB6A7A1AB6
        52317D50508AE72B7BE2E4E4BA24164E029CBACF786B
```

Authors' Addresses

Shawn Emery
Oracle Corporation
500 Eldorado Blvd Bldg 1
Broomfield, CO 78727
US

EMail: shawn.emery@oracle.com

Nicolas Williams
Cryptonector, LLC

EMail: nico@cryptonector.com

Network Working Group
Internet-Draft
Obsoletes: 5653 (if approved)
Intended status: Standards Track
Expires: August 13, 2018

M. Upadhyay
Google
S. Malkani
ActivIdentity
W. Wang
Oracle

February 9, 2018

Generic Security Service API Version 2: Java Bindings Update
draft-ietf-kitten-rfc5653bis-07

Abstract

The Generic Security Services Application Program Interface (GSS-API) offers application programmers uniform access to security services atop a variety of underlying cryptographic mechanisms. This document updates the Java bindings for the GSS-API that are specified in "Generic Security Service API Version 2 : Java Bindings Update" (RFC 5653). This document obsoletes RFC 5653 by adding a new output token field to the GSSException class so that when the initSecContext or acceptSecContext methods of the GSSContext class fails it has a chance to emit an error token which can be sent to the peer for debugging or informational purpose. The stream-based GSSContext methods are also removed in this version.

The GSS-API is described at a language-independent conceptual level in "Generic Security Service Application Program Interface Version 2, Update 1" (RFC 2743). The GSS-API allows a caller application to authenticate a principal identity, to delegate rights to a peer, and to apply security services such as confidentiality and integrity on a per-message basis. Examples of security mechanisms defined for GSS-API are "The Simple Public-Key GSS-API Mechanism" (RFC 2025) and "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2" (RFC 4121).

Status of This Memo

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 13, 2018.

Copyright Notice

Table of Contents

1. Introduction

This document specifies Java language bindings for the Generic Security Services Application Programming Interface version 2 (GSS-API). GSS-API version 2 is described in a language-independent format in RFC 2743 [RFC2743]. The GSS-API allows a caller application to authenticate a principal identity, to delegate rights to a peer, and to apply security services such as confidentiality and integrity on a per-message basis.

This document and its predecessor, RFC 2853 [RFC2853] and RFC 5653 [RFC5653], leverage the work done by the working group (WG) in the area of RFC 2743 [RFC2743] and the C-bindings of RFC 2744 [RFC2744]. Whenever appropriate, text has been used from the C-bindings document (RFC 2744) to explain generic concepts and provide direction to the implementors.

The design goals of this API have been to satisfy all the functionality defined in RFC 2743 [RFC2743] and to provide these services in an object-oriented method. The specification also aims to satisfy the needs of both types of Java application developers, those who would like access to a "system-wide" GSS-API implementation, as well as those who would want to provide their own "custom" implementation.

A system-wide implementation is one that is available to all applications in the form of a library package. It may be the standard package in the Java runtime environment (JRE) being used or it may be additionally installed and accessible to any application via the CLASSPATH.

A custom implementation of the GSS-API, on the other hand, is one that would, in most cases, be bundled with the application during distribution. It is expected that such an implementation would be

meant to provide for some particular need of the application, such as support for some specific mechanism.

The design of this API also aims to provide a flexible framework to add and manage GSS-API mechanisms. GSS-API leverages the Java Cryptography Architecture (JCA) provider model to support the plugability of mechanisms. Mechanisms can be added on a system-wide basis, where all users of the framework will have them available. The specification also allows for the addition of mechanisms per-instance of the GSS-API.

Lastly, this specification presents an API that will naturally fit within the operation environment of the Java platform. Readers are assumed to be familiar with both the GSS-API and the Java platform.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. GSS-API Operational Paradigm

"Generic Security Service Application Programming Interface, Version 2" [RFC2743] defines a generic security API to calling applications. It allows a communicating application to authenticate the user associated with another application, to delegate rights to another application, and to apply security services such as confidentiality and integrity on a per-message basis.

There are four stages to using GSS-API:

1) The application acquires a set of credentials with which it may prove its identity to other processes. The application’s credentials vouch for its global identity, which may or may not be related to any local username under which it may be running.

2) A pair of communicating applications establish a joint security context using their credentials. The security context encapsulates shared state information, which is required in order that per-message security services may be provided. Examples of state information that might be shared between applications as part of a security context are cryptographic keys and message sequence numbers. As part of the establishment of a security context, the context initiator is authenticated to the responder, and may require that the responder is authenticated back to the

initiator. The initiator may optionally give the responder the right to initiate further security contexts, acting as an agent or delegate of the initiator. This transfer of rights is termed "delegation", and is achieved by creating a set of credentials, similar to those used by the initiating application, but which may be used by the responder.

A GSSContext object is used to establish and maintain the shared information that makes up the security context. Certain GSSContext methods will generate a token, which applications treat as cryptographically protected, opaque data. The caller of such a GSSContext method is responsible for transferring the token to the peer application, encapsulated if necessary in an application-to-application protocol. On receipt of such a token, the peer application should pass it to a corresponding GSSContext method which will decode the token and extract the information, updating the security context state information accordingly.

3) Per-message services are invoked on a GSSContext object to apply either:

integrity and data origin authentication, or

confidentiality, integrity and data origin authentication

to application data, which are treated by GSS-API as arbitrary octet-strings. An application transmitting a message that it wishes to protect will call the appropriate GSSContext method (getMIC or wrap) to apply protection, and send the resulting token to the receiving application. The receiver will pass the received token (and, in the case of data protected by getMIC, the accompanying message-data) to the corresponding decoding method of the GSSContext interface (verifyMIC or unwrap) to remove the protection and validate the data.

4) At the completion of a communications session (which may extend across several transport connections), each application uses a GSSContext method to invalidate the security context and release any system or cryptographic resources held. Multiple contexts may also be used (either successively or simultaneously) within a single communications association, at the discretion of the applications.

4. Additional Controls

This section discusses the OPTIONAL services that a context initiator may request of the GSS-API before the context establishment. Each of these services is requested by calling the appropriate mutator method

in the GSSContext object before the first call to init is performed. Only the context initiator can request context flags.

The OPTIONAL services defined are:

Delegation: The (usually temporary) transfer of rights from initiator to acceptor, enabling the acceptor to authenticate itself as an agent of the initiator.

Mutual Authentication: In addition to the initiator authenticating its identity to the context acceptor, the context acceptor SHOULD also authenticate itself to the initiator.

Replay Detection: In addition to providing message integrity services, GSSContext per-message operations of getMIC and wrap SHOULD include message numbering information to enable verifyMIC and unwrap to detect if a message has been duplicated.

Out-of-Sequence Detection: In addition to providing message integrity services, GSSContext per-message operations (getMIC and wrap) SHOULD include message sequencing information to enable verifyMIC and unwrap to detect if a message has been received out of sequence.

Anonymous Authentication: The establishment of the security context SHOULD NOT reveal the initiator’s identity to the context acceptor.

Some mechanisms may not support all OPTIONAL services, and some mechanisms may only support some services in conjunction with others. The GSSContext interface offers query methods to allow the verification by the calling application of which services will be available from the context when the establishment phase is complete. In general, if the security mechanism is capable of providing a requested service, it SHOULD do so even if additional services must be enabled in order to provide the requested service. If the mechanism is incapable of providing a requested service, it SHOULD proceed without the service leaving the application to abort the context establishment process if it considers the requested service to be mandatory.

Some mechanisms MAY specify that support for some services is optional, and that implementors of the mechanism need not provide it. This is most commonly true of the confidentiality service, often because of legal restrictions on the use of data-encryption, but may apply to any of the services. Such mechanisms are required to send at least one token from acceptor to initiator during context establishment when the initiator indicates a desire to use such a

service, so that the initiating GSS-API can correctly indicate whether the service is supported by the acceptor's GSS-API.

4.1. Delegation

The GSS-API allows delegation to be controlled by the initiating application via the requestCredDeleg method before the first call to init has been issued. Some mechanisms do not support delegation, and for such mechanisms, attempts by an application to enable delegation are ignored.

The acceptor of a security context, for which the initiator enabled delegation, can check if delegation was enabled by using the getCredDelegState method of the GSSContext interface. In cases when it is enabled, the delegated credential object can be obtained by calling the getDelegCred method. The obtained GSSCredential object may then be used to initiate subsequent GSS-API security contexts as an agent or delegate of the initiator. If the original initiator's identity is "A" and the delegate's identity is "B", then, depending on the underlying mechanism, the identity embodied by the delegated credential may be either "A" or "B acting for A".

For many mechanisms that support delegation, a simple boolean does not provide enough control. Examples of additional aspects of delegation control that a mechanism might provide to an application are duration of delegation, network addresses from which delegation is valid, and constraints on the tasks that may be performed by a delegate. Such controls are presently outside the scope of the GSS-API. GSS-API implementations supporting mechanisms offering additional controls SHOULD provide extension routines that allow these controls to be exercised (perhaps by modifying the initiator's GSS-API credential object prior to its use in establishing a context). However, the simple delegation control provided by GSS-API SHOULD always be able to override other mechanism-specific delegation controls. If the application instructs the GSSContext object that delegation is not desired, then the implementation MUST NOT permit delegation to occur. This is an exception to the general rule that a mechanism may enable services even if they are not requested -- delegation may only be provided at the explicit request of the application.

4.2. Mutual Authentication

Usually, a context acceptor will require that a context initiator authenticate itself so that the acceptor may make an access-control decision prior to performing a service for the initiator. In some cases, the initiator may also request that the acceptor authenticate itself. GSS-API allows the initiating application to request this

mutual authentication service by calling the requestMutualAuth method of the GSSContext interface with a "true" parameter before making the first call to init. The initiating application is informed as to whether or not the context acceptor has authenticated itself. Note that some mechanisms may not support mutual authentication, and other mechanisms may always perform mutual authentication, whether or not the initiating application requests it. In particular, mutual authentication may be required by some mechanisms in order to support replay or out-of-sequence message detection, and for such mechanisms, a request for either of these services will automatically enable mutual authentication.

4.3. Replay and Out-of-Sequence Detection

The GSS-API MAY provide detection of mis-ordered messages once a security context has been established. Protection MAY be applied to messages by either application, by calling either getMIC or wrap methods of the GSSContext interface, and verified by the peer application by calling verifyMIC or unwrap for the peer’s GSSContext object.

The getMIC method calculates a cryptographic checksum (authentication tag) of an application message, and returns that checksum in a token. The application SHOULD pass both the token and the message to the peer application, which presents them to the verifyMIC method of the peer’s GSSContext object.

The wrap method calculates a cryptographic checksum of an application message, and places both the checksum and the message inside a single token. The application SHOULD pass the token to the peer application, which presents it to the unwrap method of the peer’s GSSContext object to extract the message and verify the checksum.

Either pair of routines may be capable of detecting out-of-sequence message delivery or the duplication of messages. Details of such mis-ordered messages are indicated through supplementary query methods of the MessageProp object that is filled in by each of these routines.

A mechanism need not maintain a list of all tokens that have been processed in order to support these status codes. A typical mechanism might retain information about only the most recent "N" tokens processed, allowing it to distinguish duplicates and missing tokens within the most recent "N" messages; the receipt of a token older than the most recent "N" would result in the isOldToken method of the instance of MessageProp to return "true".

4.4. Anonymous Authentication

In certain situations, an application may wish to initiate the authentication process to authenticate a peer, without revealing its own identity. As an example, consider an application providing access to a database containing medical information and offering unrestricted access to the service. A client of such a service might wish to authenticate the service (in order to establish trust in any information retrieved from it), but might not wish the service to be able to obtain the client's identity (perhaps due to privacy concerns about the specific inquiries, or perhaps simply to avoid being placed on mailing-lists).

In normal use of the GSS-API, the initiator's identity is made available to the acceptor as a result of the context establishment process. However, context initiators may request that their identity not be revealed to the context acceptor. Many mechanisms do not support anonymous authentication, and for such mechanisms, the request will not be honored. An authentication token will still be generated, but the application is always informed if a requested service is unavailable, and has the option to abort context establishment if anonymity is valued above the other security services that would require a context to be established.

In addition to informing the application that a context is established anonymously (via the isAnonymous method of the GSSContext class), the getSrcName method of the acceptor's GSSContext object will, for such contexts, return a reserved internal-form name, defined by the implementation.

The toString method for a GSSName object representing an anonymous entity will return a printable name. The returned value will be syntactically distinguishable from any valid principal name supported by the implementation. The associated name-type object identifier will be an oid representing the value of NT_ANONYMOUS. This name-type oid will be defined as a public, static Oid object of the GSSName class. The printable form of an anonymous name SHOULD be chosen such that it implies anonymity, since this name may appear in, for example, audit logs. For example, the string "<anonymous>" might be a good choice, if no valid printable names supported by the implementation can begin with "<" and end with ">".

When using the equal method of the GSSName interface, and one of the operands is a GSSName instance representing an anonymous entity, the method MUST return "false".

4.5. Integrity and Confidentiality

If a GSSContext supports the integrity service, getMic method may be used to create message integrity check tokens on application messages.

If a GSSContext supports the confidentiality service, wrap method may be used to encrypt application messages. Messages are selectively encrypted, under the control of the setPrivacy method of the MessageProp object used in the wrap method. Confidentiality will be applied if the privacy state is set to true.

4.6. Inter-process Context Transfer

GSS-APIv2 provides functionality that allows a security context to be transferred between processes on a single machine. These are implemented using the export method of GSSContext and a byte array constructor of the same class. The most common use for such a feature is a client-server design where the server is implemented as a single process that accepts incoming security contexts, which then launches child processes to deal with the data on these contexts. In such a design, the child processes must have access to the security context object created within the parent so that they can use per-message protection services and delete the security context when the communication session ends.

Since the security context data structure is expected to contain sequencing information, it is impractical in general to share a context between processes. Thus, the GSSContext interface provides an export method that the process, which currently owns the context, can call to declare that it has no intention to use the context subsequently, and to create an inter-process token containing information needed by the adopting process to successfully recreate the context. After successful completion of export, the original security context is made inaccessible to the calling process by GSS-API, and any further usage of this object will result in failures. The originating process transfers the inter-process token to the adopting process, which creates a new GSSContext object using the byte array constructor. The properties of the context are equivalent to that of the original context.

The inter-process token MAY contain sensitive data from the original security context (including cryptographic keys). Applications using inter-process tokens to transfer security contexts MUST take appropriate steps to protect these tokens in transit.

Implementations are not required to support the inter-process transfer of security contexts. Calling the isTransferable method of

the GSSContext interface will indicate if the context object is transferable.

4.7. The Use of Incomplete Contexts

Some mechanisms may allow the per-message services to be used before the context establishment process is complete. For example, a mechanism may include sufficient information in its initial context-level tokens for the context acceptor to immediately decode messages protected with wrap or getMIC. For such a mechanism, the initiating application need not wait until subsequent context-level tokens have been sent and received before invoking the per-message protection services.

An application can invoke the isProtReady method of the GSSContext class to determine if the per-message services are available in advance of complete context establishment. Applications wishing to use per-message protection services on partially established contexts SHOULD query this method before attempting to invoke wrap or getMIC.

5. Calling Conventions

Java provides the implementors with not just a syntax for the language, but also an operational environment. For example, memory is automatically managed and does not require application intervention. These language features have allowed for a simpler API and have led to the elimination of certain GSS-API functions.

Moreover, the JCA defines a provider model that allows for implementation-independent access to security services. Using this model, applications can seamlessly switch between different implementations and dynamically add new services. The GSS-API specification leverages these concepts by the usage of providers for the mechanism implementations.

5.1. Package Name

The classes and interfaces defined in this document reside in the package called "org.ietf.jgss". Applications that wish to make use of this API should import this package name as shown in section 8.

5.2. Provider Framework

The Java security API’s use a provider architecture that allows applications to be implementation independent and security API implementations to be modular and extensible. The java.security.Provider class is an abstract class that a vendor extends. This class maps various properties that represent different

security services that are available to the names of the actual vendor classes that implement those services. When requesting a service, an application simply specifies the desired provider and the API delegates the request to service classes available from that provider.

Using the Java security provider model insulates applications from implementation details of the services they wish to use. Applications can switch between providers easily and new providers can be added as needed, even at runtime.

The GSS-API may use providers to find components for specific underlying security mechanisms. For instance, a particular provider might contain components that will allow the GSS-API to support the Kerberos v5 mechanism [RFC4121] and another might contain components to support the Simple Public-Key GSS-API Mechanism (SPKM) [RFC2025]. By delegating mechanism-specific functionality to the components obtained from providers, the GSS-API can be extended to support an arbitrary list of mechanism.

How the GSS-API locates and queries these providers is beyond the scope of this document and is being deferred to a Service Provider Interface (SPI) specification. The availability of such an SPI specification is not mandatory for the adoption of this API specification nor is it mandatory to use providers in the implementation of a GSS-API framework. However, by using the provider framework together with an SPI specification, one can create an extensible and implementation-independent GSS-API framework.

5.3. Integer Types

All numeric values are declared as "int" primitive Java type. The Java specification guarantees that this will be a 32-bit two’s complement signed number.

Throughout this API, the "boolean" primitive Java type is used wherever a boolean value is required or returned.

5.4. Opaque Data Types

Java byte arrays are used to represent opaque data types that are consumed and produced by the GSS-API in the form of tokens. Java arrays contain a length field that enables the users to easily determine their size. The language has automatic garbage collection that alleviates the need by developers to release memory and simplifies buffer ownership issues.

5.5. Strings

The String object will be used to represent all textual data. The Java String object transparently treats all characters as two-byte Unicode characters, which allows support for many locals. All routines returning or accepting textual data will use the String object.

5.6. Object Identifiers

An Oid object will be used to represent Universal Object Identifiers (Oids). Oids are ISO-defined, hierarchically globally interpretable identifiers used within the GSS-API framework to identify security mechanisms and name formats. The Oid object can be created from a string representation of its dot notation (e.g., "1.3.6.1.5.6.2") as well as from its ASN.1 DER encoding. Methods are also provided to test equality and provide the DER representation for the object.

An important feature of the Oid class is that its instances are immutable -- i.e., there are no methods defined that allow one to change the contents of an Oid. This property allows one to treat these objects as "statics" without the need to perform copies.

Certain routines allow the usage of a default oid. A "null" value can be used in those cases.

5.7. Object Identifier Sets

The Java bindings represent object identifier sets as arrays of Oid objects. All Java arrays contain a length field, which allows for easy manipulation and reference.

In order to support the full functionality of RFC 2743 [RFC2743], the Oid class includes a method that checks for existence of an Oid object within a specified array. This is equivalent in functionality to gss_test_oid_set_member. The use of Java arrays and Java’s automatic garbage collection has eliminated the need for the following routines: gss_create_empty_oid_set, gss_release_oid_set, and gss_add_oid_set_member. Java GSS-API implementations will not contain them. Java’s automatic garbage collection and the immutable property of the Oid object eliminates the memory management issues of the C counterpart.

Whenever a default value for an Object Identifier Set is required, a "null" value can be used. Please consult the detailed method description for details.

5.8. Credentials

GSS-API credentials are represented by the GSSCredential interface. The interface contains several constructs to allow for the creation of most common credential objects for the initiator and the acceptor. Comparisons are performed using the interface's "equals" method. The following general description of GSS-API credentials is included from the C-bindings specification:

> GSS-API credentials can contain mechanism-specific principal authentication data for multiple mechanisms. A GSS-API credential is composed of a set of credential-elements, each of which is applicable to a single mechanism. A credential may contain at most one credential-element for each supported mechanism. A credential-element identifies the data needed by a single mechanism to authenticate a single principal, and conceptually contains two credential-references that describe the actual mechanism-specific authentication data, one to be used by GSS-API for initiating contexts, and one to be used for accepting contexts. For mechanisms that do not distinguish between acceptor and initiator credentials, both references would point to the same underlying mechanism-specific authentication data.

Credentials describe a set of mechanism-specific principals, and give their holder the ability to act as any of those principals. All principal identities asserted by a single GSS-API credential SHOULD belong to the same entity, although enforcement of this property is an implementation-specific matter. A single GSSCredential object represents all the credential elements that have been acquired.

The creation of an GSSContext object allows the value of "null" to be specified as the GSSCredential input parameter. This will indicate a desire by the application to act as a default principal. While individual GSS-API implementations are free to determine such default behavior as appropriate to the mechanism, the following default behavior by these routines is RECOMMENDED for portability:

For the initiator side of the context:

1) If there is only a single principal capable of initiating security contexts for the chosen mechanism that the application is authorized to act on behalf of, then that principal shall be used; otherwise,

2) If the platform maintains a concept of a default network-identity for the chosen mechanism, and if the application is authorized to act on behalf of that identity for the purpose of initiating

security contexts, then the principal corresponding to that identity shall be used; otherwise,

3) If the platform maintains a concept of a default local identity, and provides a means to map local identities into network-identities for the chosen mechanism, and if the application is authorized to act on behalf of the network-identity image of the default local identity for the purpose of initiating security contexts using the chosen mechanism, then the principal corresponding to that identity shall be used; otherwise,

4) A user-configurable default identity should be used.

For the acceptor side of the context:

1) If there is only a single authorized principal identity capable of accepting security contexts for the chosen mechanism, then that principal shall be used; otherwise,

2) If the mechanism can determine the identity of the target principal by examining the context-establishment token processed during the accept method, and if the accepting application is authorized to act as that principal for the purpose of accepting security contexts using the chosen mechanism, then that principal identity shall be used; otherwise,

3) If the mechanism supports context acceptance by any principal, and if mutual authentication was not requested, any principal that the application is authorized to accept security contexts under using the chosen mechanism may be used; otherwise,

4) A user-configurable default identity shall be used.

The purpose of the above rules is to allow security contexts to be established by both initiator and acceptor using the default behavior whenever possible. Applications requesting default behavior are likely to be more portable across mechanisms and implementations than ones that instantiate an GSSCredential object representing a specific identity.

5.9. Contexts

The GSSContext interface is used to represent one end of a GSS-API security context, storing state information appropriate to that end of the peer communication, including cryptographic state information. The instantiation of the context object is done differently by the initiator and the acceptor. After the context has been instantiated, the initiator MAY choose to set various context options that will

determine the characteristics of the desired security context. When all the application-desired characteristics have been set, the initiator will call the initSecContext method, which will produce a token for consumption by the peer's acceptSecContext method. It is the responsibility of the application to deliver the authentication token(s) between the peer applications for processing. Upon completion of the context-establishment phase, context attributes can be retrieved, by both the initiator and acceptor, using the accessor methods. These will reflect the actual attributes of the established context and might not match the initiator-requested values. If any retrieved attribute does not match the desired value but it is necessary for the application protocol, the application SHOULD destroy the security context and not use it for application traffic. Otherwise, at this point, the context can be used by the application to apply cryptographic services to its data.

5.10. Authentication Tokens

A token is a caller-opaque type that GSS-API uses to maintain synchronization between each end of the GSS-API security context. The token is a cryptographically protected octet-string, generated by the underlying mechanism at one end of a GSS-API security context for use by the peer mechanism at the other end. Encapsulation (if required) within the application protocol and transfer of the token are the responsibility of the peer applications.

Java GSS-API uses byte arrays to represent authentication tokens.

5.11. Inter-Process Tokens

Certain GSS-API routines are intended to transfer data between processes in multi-process programs. These routines use a caller-opaque octet-string, generated by the GSS-API in one process for use by the GSS-API in another process. The calling application is responsible for transferring such tokens between processes. Note that, while GSS-API implementors are encouraged to avoid placing sensitive information within inter-process tokens, or to cryptographically protect them, many implementations will be unable to avoid placing key material or other sensitive data within them. It is the application's responsibility to ensure that inter-process tokens are protected in transit, and transferred only to processes that are trustworthy. An inter-process token is represented using a byte array emitted from the export method of the GSSContext interface. The receiver of the inter-process token would initialize an GSSContext object with this token to create a new context. Once a context has been exported, the GSSContext object is invalidated and is no longer available.

5.12. Error Reporting

RFC 2743 [RFC2743] defined the usage of major and minor status values for the signaling of GSS-API errors. The major code, also called GSS status code, is used to signal errors at the GSS-API level, independent of the underlying mechanism(s). The minor status value or Mechanism status code, is a mechanism-defined error value indicating a mechanism-specific error code.

Java GSS-API uses exceptions implemented by the GSSException class to signal both minor and major error values. Both mechanism-specific errors and GSS-API level errors are signaled through instances of this class. The usage of exceptions replaces the need for major and minor codes to be used within the API calls. The GSSException class also contains methods to obtain textual representations for both the major and minor values, which is equivalent to the functionality of gss_display_status. A GSSException object MAY also include an output token that SHOULD be sent to the peer.

If an exception is thrown during context establishment, the context negotiation has failed and the GSSContext object MUST be abandoned. If it is thrown in a per-message call, the context MAY remain useful.

5.12.1. GSS Status Codes

GSS status codes indicate errors that are independent of the underlying mechanism(s) used to provide the security service. The errors that can be indicated via a GSS status code are generic API routine errors (errors that are defined in the GSS-API specification). These bindings take advantage of the Java exceptions mechanism, thus, eliminating the need for calling errors.

A GSS status code indicates a single fatal generic API error from the routine that has thrown the GSSException. Using exceptions announces that a fatal error has occurred during the execution of the method. The GSS-API operational model also allows for the signaling of supplementary status information from the per-message calls. These need to be handled as return values since using exceptions is not appropriate for informatory or warning-like information. The methods that are capable of producing supplementary information are the two per-message methods GSSContext.verifyMIC() and GSSContext.unwrap(). These methods fill the supplementary status codes in the MessageProp object that was passed in.

A GSSException object, along with providing the functionality for setting of the various error codes and translating them into textual representation, also contains the definitions of all the numeric

error values. The following table lists the definitions of error codes:

Table: GSS Status Codes

Name	Value	Meaning
BAD_BINDINGS	1	Incorrect channel bindings were supplied.
BAD_MECH	2	An unsupported mechanism was requested.
BAD_NAME	3	An invalid name was supplied.
BAD_NAMETYPE	4	A supplied name was of an unsupported type.
BAD_STATUS	5	An invalid status code was supplied.
BAD_MIC	6	A token had an invalid MIC.
CONTEXT_EXPIRED	7	The context has expired.
CREDENTIALS_EXPIRED	8	The referenced credentials have expired.
DEFECTIVE_CREDENTIAL	9	A supplied credential was invalid.
DEFECTIVE_TOKEN	10	A supplied token was invalid.
FAILURE	11	Miscellaneous failure, unspecified at the GSS-API level.
NO_CONTEXT	12	Invalid context has been supplied.
NO_CRED	13	No credentials were supplied, or the credentials were unavailable or inaccessible.
BAD_QOP	14	The quality-of-protection (QOP) requested could not be provided.
UNAUTHORIZED	15	The operation is forbidden by the local security policy.

UNAVAILABLE	16	The operation or option is unavailable.
DUPLICATE_ELEMENT	17	The requested credential element already exists.
NAME_NOT_MN	18	The provided name was not a mechanism name.

The following four status codes (DUPLICATE_TOKEN, OLD_TOKEN, UNSEQ_TOKEN, and GAP_TOKEN) are contained in a GSSException only if detected during context establishment, in which case it is a fatal error. (During per-message calls, these values are indicated as supplementary information contained in the MessageProp object.) They are:

Name	Value	Meaning
DUPLICATE_TOKEN	19	The token was a duplicate of an earlier version.
OLD_TOKEN	20	The token’s validity period has expired.
UNSEQ_TOKEN	21	A later token has already been processed.
GAP_TOKEN	22	The expected token was not received.

The GSS major status code of FAILURE is used to indicate that the underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code can provide more details about the error.

The different major status codes that can be contained in the GSSException object thrown by the methods in this specification are the same as the major status codes returned by the corresponding calls in RFC 2743 [RFC2743].

5.12.2. Mechanism-Specific Status Codes

Mechanism-specific status codes are communicated in two ways, they are part of any GSSException thrown from the mechanism-specific layer to signal a fatal error, or they are part of the MessageProp object that the per-message calls use to signal non-fatal errors.

A default value of 0 in either the GSSException object or the MessageProp object will be used to represent the absence of any mechanism-specific status code.

5.12.3. Supplementary Status Codes

Supplementary status codes are confined to the per-message methods of the GSSContext interface. Because of the informative nature of these errors it is not appropriate to use exceptions to signal them. Instead, the per-message operations of the GSSContext interface return these values in a MessageProp object.

The MessageProp class defines query methods that return boolean values indicating the following supplementary states:

Table: Supplementary Status Methods

Method Name	Meaning when "true" is returned
isDuplicateToken	The token was a duplicate of an earlier token.
isOldToken	The token’s validity period has expired.
isUnseqToken	A later token has already been processed.
isGapToken	An expected per-message token was not received.

A "true" return value for any of the above methods indicates that the token exhibited the specified property. The application MUST determine the appropriate course of action for these supplementary values. They are not treated as errors by the GSS-API.

5.13. Names

A name is used to identify a person or entity. GSS-API authenticates the relationship between a name and the entity claiming the name.

Since different authentication mechanisms may employ different namespaces for identifying their principals, GSS-API’s naming support is necessarily complex in multi-mechanism environments (or even in some single-mechanism environments where the underlying mechanism supports multiple namespaces).

Two distinct conceptual representations are defined for names:

1) A GSS-API form represented by implementations of the GSSName interface: A single GSSName object MAY contain multiple names from different namespaces, but all names SHOULD refer to the same entity. An example of such an internal name would be the name returned from a call to the getName method of the GSSCredential interface, when applied to a credential containing credential elements for multiple authentication mechanisms employing different namespaces. This GSSName object will contain a distinct name for the entity for each authentication mechanism.

   For GSS-API implementations supporting multiple namespaces, GSSName implementations MUST contain sufficient information to determine the namespace to which each primitive name belongs.

2) Mechanism-specific contiguous byte array and string forms: Different GSSName initialization methods are provided to handle both byte array and string formats and to accommodate various calling applications and name types. These formats are capable of containing only a single name (from a single namespace). Contiguous string names are always accompanied by an object identifier specifying the namespace to which the name belongs, and their format is dependent on the authentication mechanism that employs that name. The string name forms are assumed to be printable, and may therefore be used by GSS-API applications for communication with their users. The byte array name formats are assumed to be in non-printable formats (e.g., the byte array returned from the export method of the GSSName interface).

A GSSName object can be converted to a contiguous representation by using the toString method. This will guarantee that the name will be converted to a printable format. Different initialization methods in the GSSName interface are defined allowing support for multiple syntaxes for each supported namespace, and allowing users the freedom to choose a preferred name representation. The toString method SHOULD use an implementation-chosen printable syntax for each supported name type. To obtain the printable name type, getStringNameType method can be used.

There is no guarantee that calling the toString method on the GSSName interface will produce the same string form as the original imported string name. Furthermore, it is possible that the name was not even constructed from a string representation. The same applies to namespace identifiers, which may not necessarily survive unchanged after a journey through the internal name form. An example of this might be a mechanism that authenticates X.500 names, but provides an algorithmic mapping of Internet DNS names into X.500. That mechanism’s implementation of GSSName might, when presented with a DNS name, generate an internal name that contained both the original

DNS name and the equivalent X.500 name. Alternatively, it might only store the X.500 name. In the latter case, the toString method of GSSName would most likely generate a printable X.500 name, rather than the original DNS name.

The context acceptor can obtain a GSSName object representing the entity performing the context initiation (through the usage of getSrcName method). Since this name has been authenticated by a single mechanism, it contains only a single name (even if the internal name presented by the context initiator to the GSSContext object had multiple components). Such names are termed internal-mechanism names (or MNs), and the names emitted by GSSContext interface in the getSrcName and getTargName are always of this type. Since some applications may require MNs without wanting to incur the overhead of an authentication operation, creation methods are provided that take not only the name buffer and name type, but also the mechanism oid for which this name should be created. When dealing with an existing GSSName object, the canonicalize method may be invoked to convert a general internal name into an MN.

GSSName objects can be compared using their equal method, which returns "true" if the two names being compared refer to the same entity. This is the preferred way to perform name comparisons instead of using the printable names that a given GSS-API implementation may support. Since GSS-API assumes that all primitive names contained within a given internal name refer to the same entity, equal can return "true" if the two names have at least one primitive name in common. If the implementation embodies knowledge of equivalence relationships between names taken from different namespaces, this knowledge may also allow successful comparisons of internal names containing no overlapping primitive elements. However, applications SHOULD note that to avoid surpising behavior, it is best to ensure that the names being compared are either both mechanism names for the same mechanism, or both internal names that are not mechanism names. This holds whether the equals method is used directly, or the export method is used to generate byte strings that are then compared byte-by-byte.

When used in large access control lists, the overhead of creating a GSSName object on each name and invoking the equal method on each name from the Access Control List (ACL) may be prohibitive. As an alternative way of supporting this case, GSS-API defines a special form of the contiguous byte array name, which MAY be compared directly (byte by byte). Contiguous names suitable for comparison are generated by the export method. Exported names MAY be re-imported by using the byte array constructor and specifying the NT_EXPORT_NAME as the name type object identifier. The resulting GSSName name will also be a MN.

The GSSName interface defines public static Oid objects representing the standard name types. Structurally, an exported name object consists of a header containing an OID identifying the mechanism that authenticated the name, and a trailer containing the name itself, where the syntax of the trailer is defined by the individual mechanism specification. Detailed description of the format is specified in the language-independent GSS-API specification [RFC2743].

Note that the results obtained by using the equals method will in general be different from those obtained by invoking canonicalize and export, and then comparing the byte array output. The first series of operation determines whether two (unauthenticated) names identify the same principal; the second whether a particular mechanism would authenticate them as the same principal. These two operations will in general give the same results only for MNs.

It is important to note that the above are guidelines as to how GSSName implementations SHOULD behave, and are not intended to be specific requirements of how name objects must be implemented. The mechanism designers are free to decide on the details of their implementations of the GSSName interface as long as the behavior satisfies the above guidelines.

5.14. Channel Bindings

GSS-API supports the use of user-specified tags to identify a given context to the peer application. These tags are intended to be used to identify the particular communications channel that carries the context. Channel bindings are communicated to the GSS-API using the ChannelBinding object. The application MAY use byte arrays to specify the application data to be used in the channel binding as well as using instances of the InetAddress. The InetAddress for the initiator and/or acceptor can be used within an instance of a ChannelBinding. ChannelBinding can be set for the GSSContext object using the setChannelBinding method before the first call to init or accept has been performed. Unless the setChannelBinding method has been used to set the ChannelBinding for a GSSContext object, "null" ChannelBinding will be assumed. InetAddress is currently the only address type defined within the Java platform and as such, it is the only one supported within the ChannelBinding class. Applications that use other types of addresses can include them as part of the application-specific data.

Conceptually, the GSS-API concatenates the initiator and acceptor address information, and the application-supplied byte array to form an octet-string. The mechanism calculates a Message Integrity Code (MIC) over this octet-string and binds the MIC to the context

establishment token emitted by the init method of the GSSContext interface. The same bindings are set by the context acceptor for its GSSContext object and during processing of the accept method, a MIC is calculated in the same way. The calculated MIC is compared with that found in the token, and if the MICs differ, accept will throw a GSSException with the major code set to BAD_BINDINGS, and the context will not be established. Some mechanisms may include the actual channel binding data in the token (rather than just a MIC); applications SHOULD therefore not use confidential data as channel-binding components.

Individual mechanisms may impose additional constraints on addresses that may appear in channel bindings. For example, a mechanism may verify that the initiator address field of the channel binding contains the correct network address of the host system. Portable applications SHOULD therefore ensure that they either provide correct information for the address fields, or omit the setting of the addressing information.

5.15. Optional Parameters

Whenever the application wishes to omit an optional parameter the "null" value SHALL be used. The detailed method descriptions indicate which parameters are optional. Method overloading has also been used as a technique to indicate default parameters.

6. Introduction to GSS-API Classes and Interfaces

This section presents a brief description of the classes and interfaces that constitute the GSS-API. The implementations of these are obtained from the CLASSPATH defined by the application. If Java GSS becomes part of the standard Java APIs, then these classes will be available by default on all systems as part of the JRE’s system classes.

This section also shows the corresponding RFC 2743 [RFC2743] functionality implemented by each of the classes. Detailed description of these classes and their methods is presented in section 7

6.1. GSSManager Class

This abstract class serves as a factory to instantiate implementations of the GSS-API interfaces and also provides methods to make queries about underlying security mechanisms.

A default implementation can be obtained using the static method getInstance(). Applications that desire to provide their own

implementation of the GSSManager class can simply extend the abstract class themselves.

This class contains equivalents of the following RFC 2743 [RFC2743] routines:

RFC 2743 Routine	Function	Section(s)
gss_import_name	Create an internal name from the supplied information.	7.1.5 - 7.1.8
gss_acquire_cred	Acquire credential for use.	7.1.9 - 7.1.11
gss_import_sec_context	Create a previously exported context.	7.1.14
gss_indicate_mechs	List the mechanisms supported by this GSS-API implementation.	7.1.2
gss_inquire_mechs_for_name	List the mechanisms supporting the specified name type.	7.1.4
gss_inquire_names_for_mech	List the name types supported by the specified mechanism.	7.1.3

6.2. GSSName Interface

GSS-API names are represented in the Java bindings through the GSSName interface. Different name formats and their definitions are identified with Universal Object Identifiers (oids). The format of the names can be derived based on the unique oid of each name type. The following GSS-API routines are provided by the GSSName interface:

RFC 2743 Routine	Function	Section(s)
gss_display_name	Convert internal name representation to text format.	7.2.6
gss_compare_name	Compare two internal names.	7.2.2, 7.2.3
gss_release_name	Release resources associated with the internal name.	N/A
gss_canonicalize_name	Convert an internal name to a mechanism name.	7.2.4
gss_export_name	Convert a mechanism name to export format.	7.2.5
gss_duplicate_name	Create a copy of the internal name.	N/A

The gss_release_name call is not provided as Java does its own garbage collection. The gss_duplicate_name call is also redundant; the GSSName interface has no mutator methods that can change the state of the object so it is safe for sharing across threads.

6.3. GSSCredential Interface

The GSSCredential interface is responsible for the encapsulation of GSS-API credentials. Credentials identify a single entity and provide the necessary cryptographic information to enable the creation of a context on behalf of that entity. A single credential may contain multiple mechanism-specific credentials, each referred to as a credential element. The GSSCredential interface provides the functionality of the following GSS-API routines:

RFC 2743 Routine	Function	Section(s)
gss_add_cred	Constructs credentials incrementally.	7.3.11
gss_inquire_cred	Obtain information about credential.	7.3.3 - 7.3.10
gss_inquire_cred_by_mech	Obtain per-mechanism information about a credential.	7.3.4 - 7.3.9
gss_release_cred	Dispose of credentials after use.	7.3.2

6.4. GSSContext Interface

This interface encapsulates the functionality of context-level calls required for security context establishment and management between peers as well as the per-message services offered to applications. A context is established between a pair of peers and allows the usage of security services on a per-message basis on application data. It is created over a single security mechanism. The GSSContext interface provides the functionality of the following GSS-API routines:

RFC 2743 Routine	Function	Section(s)
gss_init_sec_context	Initiate the creation of a security context with a peer.	7.4.2
gss_accept_sec_context	Accept a security context initiated by a peer.	7.4.3
gss_delete_sec_context	Destroy a security context.	7.4.5
gss_context_time	Obtain remaining context time.	7.4.30
gss_inquire_context	Obtain context characteristics.	7.4.21 - 7.4.35
gss_wrap_size_limit	Determine token-size limit for gss_wrap.	7.4.6
gss_export_sec_context	Transfer security context to another process.	7.4.11
gss_get_mic	Calculate a cryptographic Message Integrity Code (MIC) for a message.	7.4.9
gss_verify_mic	Verify integrity on a received message.	7.4.10
gss_wrap	Attach a MIC to a message and optionally encrypt the message content.	7.4.7
gss_unwrap	Obtain a previously wrapped application message verifying its integrity and optionally decrypting it.	7.4.8

The functionality offered by the gss_process_context_token routine has not been included in the Java bindings specification. The corresponding functionality of gss_delete_sec_context has also been modified to not return any peer tokens. This has been proposed in accordance to the recommendations stated in RFC 2743 [RFC2743]. GSSContext does offer the functionality of destroying the locally stored context information.

6.5. MessageProp Class

This helper class is used in the per-message operations on the context. An instance of this class is created by the application and then passed into the per-message calls. In some cases, the application conveys information to the GSS-API implementation through this object and in other cases the GSS-API returns information to the application by setting it in this object. See the description of the per-message operations wrap, unwrap, getMIC, and verifyMIC in the GSSContext interfaces for details.

6.6. GSSException Class

Exceptions are used in the Java bindings to signal fatal errors to the calling applications. This replaces the major and minor codes used in the C-bindings specification as a method of signaling failures. The GSSException class handles both minor and major codes, as well as their translation into textual representation. All GSS-API methods are declared as throwing this exception.

RFC 2743 Routine	Function	Section
gss_display_status	Retrieve textual representation of error codes.	7.8.5, 7.8.6, 7.8.9, 7.8.10

6.7. Oid Class

This utility class is used to represent Universal Object Identifiers and their associated operations. GSS-API uses object identifiers to distinguish between security mechanisms and name types. This class, aside from being used whenever an object identifier is needed, implements the following GSS-API functionality:

RFC 2743 Routine	Function	Section
gss_test_oid_set_member	Determine if the specified oid is part of a set of oids.	7.7.5

6.8. ChannelBinding Class

An instance of this class is used to specify channel binding information to the GSSContext object before the start of a security context establishment. The application may use a byte array to

specify application data to be used in the channel binding as well as to use instances of the InetAddress. InetAddress is currently the only address type defined within the Java platform and as such, it is the only one supported within the ChannelBinding class. Applications that use other types of addresses can include them as part of the application data.

7. Detailed GSS-API Class Description

This section lists a detailed description of all the public methods that each of the GSS-API classes and interfaces MUST provide.

7.1. public abstract class GSSManager

The GSSManager class is an abstract class that serves as a factory for three GSS interfaces: GSSName, GSSCredential, and GSSContext. It also provides methods for applications to determine what mechanisms are available from the GSS implementation and what name types these mechanisms support. An instance of the default GSSManager subclass MAY be obtained through the static method getInstance(), but applications are free to instantiate other subclasses of GSSManager.

All but one method in this class are declared abstract. This means that subclasses have to provide the complete implementation for those methods. The only exception to this is the static method getInstance(), which will have platform-specific code to return an instance of the default subclass.

Platform providers of GSS are REQUIRED not to add any constructors to this class, private, public, or protected. This will ensure that all subclasses invoke only the default constructor provided to the base class by the compiler.

A subclass extending the GSSManager abstract class MAY be implemented as a modular provider-based layer that utilizes some well-known service provider specification. The GSSManager API provides the application with methods to set provider preferences on such an implementation. These methods also allow the implementation to throw a well-defined exception in case provider-based configuration is not supported. Applications that expect to be portable SHOULD be aware of this and recover cleanly by catching the exception.

It is envisioned that there will be three most common ways in which providers will be used:

1) The application does not care about what provider is used (the default case).

2) The application wants a particular provider to be used preferentially, either for a particular mechanism or all the time, irrespective of the mechanism.

3) The application wants to use the locally configured providers as far as possible, but if support is missing for one or more mechanisms, then it wants to fall back on its own provider.

The GSSManager class has two methods that enable these modes of usage: addProviderAtFront() and addProviderAtEnd(). These methods have the effect of creating an ordered list of <provider, oid> pairs where each pair indicates a preference of provider for a given oid.

The use of these methods does not require any knowledge of whatever service provider specification the GSSManager subclass follows. It is hoped that these methods will serve the needs of most applications. Additional methods MAY be added to an extended GSSManager that could be part of a service provider specification that is standardized later.

When neither of the methods is called, the implementation SHOULD choose a default provider for each mechanism it supports.

7.1.1. getInstance

```
public static GSSManager getInstance()
```

Returns the default GSSManager implementation.

7.1.2. getMechs

```
public abstract Oid[] getMechs()
```

Returns an array of Oid objects indicating the mechanisms available to GSS-API callers. A "null" value is returned when no mechanism are available (an example of this would be when mechanism are dynamically configured, and currently no mechanisms are installed).

7.1.3. getNamesForMech

```
public abstract Oid[] getNamesForMech(Oid mech)
                      throws GSSException
```

Returns name type Oid’s supported by the specified mechanism.

Parameters:

mech The Oid object for the mechanism to query.

7.1.4. getMechsForName

```
public abstract Oid[] getMechsForName(Oid nameType)
```

Returns an array of Oid objects corresponding to the mechanisms that support the specific name type. "null" is returned when no mechanisms are found to support the specified name type.

Parameters:

nameType	The Oid object for the name type.

7.1.5. createName

```
public abstract GSSName createName(String nameStr, Oid nameType)
                throws GSSException
```

Factory method to convert a contiguous string name from the specified namespace to a GSSName object. In general, the GSSName object created will not be an MN; two examples that are exceptions to this are when the namespace type parameter indicates NT_EXPORT_NAME or when the GSS-API implementation is not multi-mechanism.

Parameters:

nameStr	The string representing a printable form of the name to create.
nameType	The Oid specifying the namespace of the printable name is supplied. Note that nameType serves to describe and qualify the interpretation of the input nameStr, it does not necessarily imply a type for the output GSSName implementation. The "null" value can be used to specify that a mechanism-specific default printable syntax SHOULD be assumed by each mechanism that examines nameStr.

7.1.6. createName

```
public abstract GSSName createName(byte[] name, Oid nameType)
                throws GSSException
```

Factory method to convert a contiguous byte array containing a name from the specified namespace to a GSSName object. In general, the GSSName object created will not be an MN; two examples that are exceptions to this are when the namespace type parameter indicates

NT_EXPORT_NAME or when the GSS-API implementation is not multi-mechanism.

Parameters:

name — The byte array containing the name to create.

nameType — The Oid specifying the namespace of the name supplied in the byte array. Note that nameType serves to describe and qualify the interpretation of the input name byte array; it does not necessarily imply a type for the output GSSName implementation. The "null" value can be used to specify that a mechanism-specific default syntax SHOULD be assumed by each mechanism that examines the byte array.

7.1.7. createName

```
public abstract GSSName createName(String nameStr, Oid nameType,
                Oid mech) throws GSSException
```

Factory method to convert a contiguous string name from the specified namespace to a GSSName object that is a mechanism name (MN). In other words, this method is a utility that does the equivalent of two steps: the createName described in section 7.1.5, and then also the GSSName.canonicalize() described in section 7.2.4.

Parameters:

nameStr — The string representing a printable form of the name to create.

nameType — The Oid specifying the namespace of the printable name supplied. Note that nameType serves to describe and qualify the interpretation of the input nameStr; it does not necessarily imply a type for the output GSSName implementation. The "null" value can be used to specify that a mechanism-specific default printable syntax SHOULD be assumed when the mechanism examines nameStr.

mech — Oid specifying the mechanism for which this name should be created.

7.1.8. createName

```
public abstract GSSName createName(byte[] name, Oid nameType,
                Oid mech) throws GSSException
```

Factory method to convert a contiguous byte array containing a name from the specified namespace to a GSSName object that is an MN. In other words, this method is a utility that does the equivalent of two steps: the createName described in section 7.1.6, and then also the GSSName.canonicalize() described in section 7.2.4.

Parameters:

name	The byte array representing the name to create.
nameType	The Oid specifying the namespace of the name supplied in the byte array. Note that nameType serves to describe and qualify the interpretation of the input name byte array, it does not necessarily imply a type for the output GSSName implementation. The "null" value can be used to specify that a mechanism-specific default syntax SHOULD be assumed by each mechanism that examines the byte array.
mech	Oid specifying the mechanism for which this name should be created.

7.1.9. createCredential

```
public abstract GSSCredential createCredential(int usage)
                throws GSSException
```

Factory method for acquiring default credentials. This will cause the GSS-API to use system-specific defaults for the set of mechanisms, name, and a DEFAULT lifetime.

Parameters:

usage	The intended usage for this credential object. The value of this parameter MUST be one of: GSSCredential.INITIATE_AND_ACCEPT(0), GSSCredential.INITIATE_ONLY(1), or GSSCredential.ACCEPT_ONLY(2)

7.1.10. createCredential

```
public abstract GSSCredential createCredential(GSSName aName,
                int lifetime, Oid mech, int usage)
                throws GSSException
```

Factory method for acquiring a single mechanism credential.

Parameters:

aName — Name of the principal for whom this credential is to be acquired. Use "null" to specify the default principal.

lifetime — The number of seconds that credentials should remain valid. Use GSSCredential.INDEFINITE_LIFETIME to request that the credentials have the maximum permitted lifetime. Use GSSCredential.DEFAULT_LIFETIME to request default credential lifetime.

mech — The oid of the desired mechanism. Use "(Oid) null" to request the default mechanism(s).

usage — The intended usage for this credential object. The value of this parameter MUST be one of:

GSSCredential.INITIATE_AND_ACCEPT(0),
GSSCredential.INITIATE_ONLY(1), or
GSSCredential.ACCEPT_ONLY(2)

7.1.11. createCredential

```
public abstract GSSCredential createCredential(GSSName aName,
                int lifetime, Oid[] mechs, int usage)
                throws GSSException
```

Factory method for acquiring credentials over a set of mechanisms. Acquires credentials for each of the mechanisms specified in the array called mechs. To determine the list of mechanisms’ for which the acquisition of credentials succeeded, the caller should use the GSSCredential.getMechs() method.

Parameters:

aName — Name of the principal for whom this credential is to be acquired. Use "null" to specify the default principal.

lifetime — The number of seconds that credentials should remain valid. Use GSSCredential.INDEFINITE_LIFETIME to request that the credentials have the maximum permitted lifetime. Use GSSCredential.DEFAULT_LIFETIME to request default credential lifetime.

mechs — The array of mechanisms over which the credential is to be acquired. Use "(Oid[]) null" for requesting a system-specific default set of mechanisms.

usage — The intended usage for this credential object. The value of this parameter MUST be one of:

GSSCredential.INITIATE_AND_ACCEPT(0), GSSCredential.INITIATE_ONLY(1), or GSSCredential.ACCEPT_ONLY(2)

7.1.12. createContext

```
public abstract GSSContext createContext(GSSName peer, Oid mech,
                GSSCredential myCred, int lifetime)
                throws GSSException
```

Factory method for creating a context on the initiator’s side. Context flags may be modified through the mutator methods prior to calling GSSContext.initSecContext().

Parameters:

peer — Name of the target peer.

mech — Oid of the desired mechanism. Use "(Oid) null" to request the default mechanism.

myCred — Credentials of the initiator. Use "null" to act as a default initiator principal.

lifetime — The request lifetime, in seconds, for the context. Use GSSContext.INDEFINITE_LIFETIME and GSSContext.DEFAULT_LIFETIME to request indefinite or default context lifetime.

7.1.13. createContext

```
public abstract GSSContext createContext(GSSCredential myCred)
                throws GSSException
```

Factory method for creating a context on the acceptor' side. The context's properties will be determined from the input token supplied to the accept method.

Parameters:

myCred	Credentials for the acceptor. Use "null" to act as a default acceptor principal.

7.1.14. createContext

```
public abstract GSSContext createContext(byte[] interProcessToken)
                throws GSSException
```

Factory method for importing a previously exported context. The context properties will be determined from the input token and can't be modified through the set methods.

Parameters:

interProcessToken	The token previously emitted from the export method.

7.1.15. addProviderAtFront

```
public abstract void addProviderAtFront(Provider p, Oid mech)
                throws GSSException
```

This method is used to indicate to the GSSManager that the application would like a particular provider to be used ahead of all others when support is desired for the given mechanism. When a value of "null" is used instead of an Oid for the mechanism, the GSSManager MUST use the indicated provider ahead of all others no matter what the mechanism is. Only when the indicated provider does not support the needed mechanism should the GSSManager move on to a different provider.

Calling this method repeatedly preserves the older settings but lowers them in preference thus forming an ordered list of provider and Oid pairs that grows at the top.

Calling addProviderAtFront with a null Oid will remove all previous preferences that were set for this provider in the GSSManager

instance. Calling addProviderAtFront with a non-null Oid will remove any previous preference that was set using this mechanism and this provider together.

If the GSSManager implementation does not support an SPI with a pluggable provider architecture, it SHOULD throw a GSSException with the status code GSSException.UNAVAILABLE to indicate that the operation is unavailable.

Parameters:

p	The provider instance that should be used whenever support is needed for mech.
mech	The mechanism for which the provider is being set.

7.1.15.1. addProviderAtFront Example Code

Suppose an application desired that the provider A always be checked first when any mechanism is needed, it would call:

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();
// mgr may at this point have its own pre-configured list
// of provider preferences.  The following will prepend to
// any such list:

mgr.addProviderAtFront(A, null);
<CODE ENDS>
```

Now if it also desired that the mechanism of Oid m1 always be obtained from the provider B before the previously set A was checked, it would call:

```
<CODE BEGINS>
mgr.addProviderAtFront(B, m1);
<CODE ENDS>
```

The GSSManager would then first check with B if m1 was needed. In case B did not provide support for m1, the GSSManager would continue on to check with A. If any mechanism m2 is needed where m2 is different from m1, then the GSSManager would skip B and check with A directly.

Suppose, at a later time, the following call is made to the same GSSManager instance:

```
<CODE BEGINS>
mgr.addProviderAtFront(B, null)
<CODE ENDS>
```

then the previous setting with the pair (B, m1) is subsumed by this and SHOULD be removed. Effectively, the list of preferences now becomes {(B, null), (A, null), ... //followed by the pre-configured list}.

Please note, however, that the following call:

```
<CODE BEGINS>
mgr.addProviderAtFront(A, m3)
<CODE ENDS>
```

does not subsume the previous setting of (A, null), and the list will effectively become {(A, m3), (B, null), (A, null), ...}

7.1.16. addProviderAtEnd

```
public abstract void addProviderAtEnd(Provider p, Oid mech)
                 throws GSSException
```

This method is used to indicate to the GSSManager that the application would like a particular provider to be used if no other provider can be found that supports the given mechanism. When a value of "null" is used instead of an Oid for the mechanism, the GSSManager MUST use the indicated provider for any mechanism.

Calling this method repeatedly preserves the older settings, but raises them above newer ones in preference thus forming an ordered list of providers and Oid pairs that grows at the bottom. Thus, the older provider settings will be utilized first before this one is.

If there are any previously existing preferences that conflict with the preference being set here, then the GSSManager SHOULD ignore this request.

If the GSSManager implementation does not support an SPI with a pluggable provider architecture, it SHOULD throw a GSSException with the status code GSSException.UNAVAILABLE to indicate that the operation is unavailable.

Parameters:

p	The provider instance that should be used whenever support is needed for mech.

mech The mechanism for which the provider is being set.

7.1.16.1. addProviderAtEnd Example Code

Suppose an application desired that when a mechanism of Oid m1 is needed, the system default providers always be checked first, and only when they do not support m1 should a provider A be checked. It would then make the call:

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();

mgr.addProviderAtEnd(A, m1);
<CODE ENDS>
```

Now, if it also desired that for all mechanisms the provider B be checked after all configured providers have been checked, it would then call:

```
<CODE BEGINS>
mgr.addProviderAtEnd(B, null);
<CODE ENDS>
```

Effectively, the list of preferences now becomes {..., (A, m1), (B, null)}.

Suppose, at a later time, the following call is made to the same GSSManager instance:

```
<CODE BEGINS>
mgr.addProviderAtEnd(B, m2)
<CODE ENDS>
```

then the previous setting with the pair (B, null) subsumes this; therefore, this request SHOULD be ignored. The same would happen if a request is made for the already existing pairs of (A, m1) or (B, null).

Please note, however, that the following call:

```
<CODE BEGINS>
mgr.addProviderAtEnd(A, null)
<CODE ENDS>
```

is not subsumed by the previous setting of (A, m1) and the list will effectively become {..., (A, m1), (B, null), (A, null)}.

7.1.17. Example Code

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();

// What mechs are available to us?

Oid[] supportedMechs = mgr.getMechs();

// Set a preference for the provider to be used when support
// is needed for the mechanisms:
//  "1.2.840.113554.1.2.2" and "1.3.6.1.5.5.1.1".

Oid krb = new Oid("1.2.840.113554.1.2.2");
Oid spkm1 = new Oid("1.3.6.1.5.5.1.1");

Provider p = (Provider) (new com.foo.security.Provider());

mgr.addProviderAtFront(p, krb);
mgr.addProviderAtFront(p, spkm1);

// What name types does this spkm implementation support?
Oid[] nameTypes = mgr.getNamesForMech(spkm1);
<CODE ENDS>
```

7.2. public interface GSSName

This interface encapsulates a single GSS-API principal entity. Different name formats and their definitions are identified with Universal Object Identifiers (Oids). The format of the names can be derived based on the unique oid of its namespace type.

7.2.1. Static Constants

public static final Oid NT_HOSTBASED_SERVICE

Oid indicating a host-based service name form. It is used to represent services associated with host computers. This name form is constructed using two elements, "service" and "hostname", as follows:

service@hostname

Values for the "service" element are registered with the IANA. It represents the following value: { iso(1) member-body(2) Unites States(840) mit(113554) infosys(1) gssapi(2) generic(1) service_name(4) }

public static final Oid NT_USER_NAME

Name type to indicate a named user on a local system. It represents the following value: { iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) user_name(1) }

public static final Oid NT_MACHINE_UID_NAME

Name type to indicate a numeric user identifier corresponding to a user on a local system (e.g., Uid). It represents the following value: { iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) machine_uid_name(2) }

public static final Oid NT_STRING_UID_NAME

Name type to indicate a string of digits representing the numeric user identifier of a user on a local system. It represents the following value: { iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) string_uid_name(3) }

public static final Oid NT_ANONYMOUS

Name type for representing an anonymous entity. It represents the following value: { iso(1), org(3), dod(6), internet(1), security(5), nametypes(6), gss-anonymous-name(3) }

public static final Oid NT_EXPORT_NAME

Name type used to indicate an exported name produced by the export method. It represents the following value: { iso(1), org(3), dod(6), internet(1), security(5), nametypes(6), gss-api-exported-name(4) }

7.2.2. equals

public boolean equals(GSSName another) throws GSSException

Compares two GSSName objects to determine whether they refer to the same entity. This method MAY throw a GSSException when the names cannot be compared. If either of the names represents an anonymous entity, the method will return "false".

Parameters:

another GSSName object with which to compare.

7.2.3. equals

public boolean equals(Object another)

A variation of the equals method, described in section 7.2.2, that is provided to override the Object.equals() method that the implementing class will inherit. The behavior is exactly the same as that in section 7.2.2 except that no GSSException is thrown; instead, "false" will be returned in the situation where an error occurs. (Note that the Java language specification requires that two objects that are equal according to the equals(Object) method MUST return the same integer result when the hashCode() method is called on them.)

Parameters:

another GSSName object with which to compare.

7.2.4. canonicalize

public GSSName canonicalize(Oid mech) throws GSSException

Creates a mechanism name (MN) from an arbitrary internal name. This is equivalent to using the factory methods described in sections 7.1.7 or 7.1.8 that take the mechanism name as one of their parameters.

Parameters:

mech The oid for the mechanism for which the canonical form of the name is requested.

7.2.5. export

public byte[] export() throws GSSException

Returns a canonical contiguous byte representation of a mechanism name (MN), suitable for direct, byte-by-byte comparison by authorization functions. If the name is not an MN, implementations MAY throw a GSSException with the NAME_NOT_MN status code. If an implementation chooses not to throw an exception, it SHOULD use some system-specific default mechanism to canonicalize the name and then export it. The format of the header of the output buffer is specified in RFC 2743 [RFC2743].

7.2.6. toString

public String toString()

Returns a textual representation of the GSSName object. To retrieve the printed name format, which determines the syntax of the returned string, the getStringNameType method can be used.

7.2.7. getStringNameType

public Oid getStringNameType() throws GSSException

Returns the oid representing the type of name returned through the toString method. Using this oid, the syntax of the printable name can be determined.

7.2.8. isAnonymous

public boolean isAnonymous()

Tests if this name object represents an anonymous entity. Returns "true" if this is an anonymous name.

7.2.9. isMN

public boolean isMN()

Tests if this name object contains only one mechanism element and is thus a mechanism name as defined by RFC 2743 [RFC2743].

7.2.10. Example Code

Included below are code examples utilizing the GSSName interface. The code below creates a GSSName, converts it to a mechanism name (MN), performs a comparison, obtains a printable representation of the name, exports it and then re-imports to obtain a new GSSName.

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();

// create a host-based service name
GSSName name = mgr.createName("service@host",
                GSSName.NT_HOSTBASED_SERVICE);

Oid krb5 = new Oid("1.2.840.113554.1.2.2");

GSSName mechName = name.canonicalize(krb5);

// the above two steps are equivalent to the following
GSSName mechName = mgr.createName("service@host",
                GSSName.NT_HOSTBASED_SERVICE, krb5);

// perform name comparison
if (name.equals(mechName))
        print("Names are equals.");

// obtain textual representation of name and its printable
// name type
print(mechName.toString() +
      mechName.getStringNameType().toString());

// export the name
byte[] exportName = mechName.export();

// create a new name object from the exported buffer
GSSName newName = mgr.createName(exportName,
                  GSSName.NT_EXPORT_NAME);
<CODE ENDS>
```

7.3. public interface GSSCredential implements Cloneable

This interface encapsulates the GSS-API credentials for an entity. A credential contains all the necessary cryptographic information to enable the creation of a context on behalf of the entity that it represents. It MAY contain multiple, distinct, mechanism-specific credential elements, each containing information for a specific security mechanism, but all referring to the same entity.

A credential MAY be used to perform context initiation, acceptance, or both.

GSS-API implementations MUST impose a local access-control policy on callers to prevent unauthorized callers from acquiring credentials to which they are not entitled. GSS-API credential creation is not intended to provide a "login to the network" function, as such a

function would involve the creation of new credentials rather than merely acquiring a handle to existing credentials. Such functions, if required, SHOULD be defined in implementation-specific extensions to the API.

If credential acquisition is time-consuming for a mechanism, the mechanism MAY choose to delay the actual acquisition until the credential is required (e.g., by GSSContext). Such mechanism-specific implementation decisions SHOULD be invisible to the calling application; thus, the query methods immediately following the creation of a credential object MUST return valid credential data, and may therefore incur the overhead of a deferred credential acquisition.

Applications will create a credential object passing the desired parameters. The application can then use the query methods to obtain specific information about the instantiated credential object (equivalent to the gss_inquire routines). When the credential is no longer needed, the application SHOULD call the dispose (equivalent to gss_release_cred) method to release any resources held by the credential object and to destroy any cryptographically sensitive information.

Classes implementing this interface also implement the Cloneable interface. This indicates that the class will support the clone() method that will allow the creation of duplicate credentials. This is useful when called just before the add() call to retain a copy of the original credential.

7.3.1. Static Constants

public static final int INITIATE_AND_ACCEPT

Credential usage flag requesting that it be able to be used for both context initiation and acceptance. The value of this constant is 0.

public static final int INITIATE_ONLY

Credential usage flag requesting that it be able to be used for context initiation only. The value of this constant is 1.

public static final int ACCEPT_ONLY

Credential usage flag requesting that it be able to be used for context acceptance only. The value of this constant is 2.

public static final int DEFAULT_LIFETIME

A lifetime constant representing the default credential lifetime. The value of this constant is 0.

public static final int INDEFINITE_LIFETIME

A lifetime constant representing indefinite credential lifetime. The value of this constant is the maximum integer value in Java - Integer.MAX_VALUE.

7.3.2. dispose

public void dispose() throws GSSException

Releases any sensitive information that the GSSCredential object may be containing. Applications SHOULD call this method as soon as the credential is no longer needed to minimize the time any sensitive information is maintained.

7.3.3. getName

public GSSName getName() throws GSSException

Retrieves the name of the entity that the credential asserts.

7.3.4. getName

public GSSName getName(Oid mechOID) throws GSSException

Retrieves a mechanism name of the entity that the credential asserts. Equivalent to calling canonicalize() on the name returned by section 7.3.3.

Parameters:

mechOID The mechanism for which information should be returned.

7.3.5. getRemainingLifetime

public int getRemainingLifetime() throws GSSException

Returns the remaining lifetime in seconds for a credential. The remaining lifetime is the minimum lifetime for any of the underlying credential mechanisms. A return value of GSSCredential.INDEFINITE_LIFETIME indicates that the credential does not expire. A return value of 0 indicates that the credential is already expired.

7.3.6. getRemainingInitLifetime

```
public int getRemainingInitLifetime(Oid mech) throws GSSException
```

Returns the remaining lifetime in seconds for the credential to remain capable of initiating security contexts under the specified mechanism. A return value of GSSCredential.INDEFINITE_LIFETIME indicates that the credential does not expire for context initiation. A return value of 0 indicates that the credential is already expired.

Parameters:

mechOID	The mechanism for which information should be returned.

7.3.7. getRemainingAcceptLifetime

```
public int getRemainingAcceptLifetime(Oid mech) throws GSSException
```

Returns the remaining lifetime in seconds for the credential to remain capable of accepting security contexts under the specified mechanism. A return value of GSSCredential.INDEFINITE_LIFETIME indicates that the credential does not expire for context acceptance. A return value of 0 indicates that the credential is already expired.

Parameters:

mechOID	The mechanism for which information should be returned.

7.3.8. getUsage

```
public int getUsage() throws GSSException
```

Returns the credential usage flag as a union over all mechanisms. The return value will be one of GSSCredential.INITIATE_AND_ACCEPT(0), GSSCredential.INITIATE_ONLY(1), or GSSCredential.ACCEPT_ONLY(2).

Specifically, GSSCredential.INITIATE_AND_ACCEPT(0) SHOULD be returned as long as there exists one credential element allowing context initiation and one credential element allowing context acceptance. These two credential elements are not necessarily the same one, nor do they need to use the same mechanism(s).

7.3.9. getUsage

```
public int getUsage(Oid mechOID) throws GSSException
```

Returns the credential usage flag for the specified mechanism only. The return value will be one of GSSCredential.INITIATE_AND_ACCEPT(0), GSSCredential.INITIATE_ONLY(1), or GSSCredential.ACCEPT_ONLY(2).

Parameters:

mechOID	The mechanism for which information should be returned.

7.3.10. getMechs

```
public Oid[] getMechs() throws GSSException
```

Returns an array of mechanisms supported by this credential.

7.3.11. add

```
public void add(GSSName aName, int initLifetime, int acceptLifetime,
                Oid mech, int usage) throws GSSException
```

Adds a mechanism-specific credential-element to an existing credential. This method allows the construction of credentials one mechanism at a time.

This routine is envisioned to be used mainly by context acceptors during the creation of acceptance credentials, which are to be used with a variety of clients using different security mechanisms.

This routine adds the new credential element "in-place". To add the element in a new credential, first call clone() to obtain a copy of this credential, then call its add() method.

Parameters:

aName	Name of the principal for whom this credential is to be acquired. Use "null" to specify the default principal.
initLifetime	The number of seconds that credentials should remain valid for initiating of security contexts. Use GSSCredential.INDEFINITE_LIFETIME to request that the credentials have the maximum permitted lifetime. Use GSSCredential.DEFAULT_LIFETIME to request default credential lifetime.

acceptLifetime The number of seconds that credentials should remain valid for accepting of security contexts.

Use GSSCredential.INDEFINITE_LIFETIME to request that the credentials
have the maximum permitted lifetime. Use GSSCredential.DEFAULT_LIFETIME to request default credential lifetime.

mech The mechanisms over which the credential is to be acquired.

usage The intended usage for this credential object. The value of this parameter MUST be one of:

GSSCredential.INITIATE_AND_ACCEPT(0),
GSSCredential.INITIATE_ONLY(1), or
GSSCredential.ACCEPT_ONLY(2)

7.3.12. equals

public boolean equals(Object another)

Tests if this GSSCredential refers to the same entity as the supplied object. The two credentials MUST be acquired over the same mechanisms and MUST refer to the same principal. Returns "true" if the two GSSCredentials refer to the same entity; "false" otherwise. (Note that the Java language specification [JLS] requires that two objects that are equal according to the equals(Object) method MUST return the same integer result when the hashCode() method is called on them.)

Parameters:

another Another GSSCredential object for comparison.

7.3.13. Example Code

This example code demonstrates the creation of a GSSCredential implementation for a specific entity, querying of its fields, and its release when it is no longer needed.

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();

// start by creating a name object for the entity
GSSName name = mgr.createName("userName", GSSName.NT_USER_NAME);

// now acquire credentials for the entity
GSSCredential cred = mgr.createCredential(name,
                     GSSCredential.ACCEPT_ONLY);

// display credential information - name, remaining lifetime,
// and the mechanisms it has been acquired over
print(cred.getName().toString());
print(cred.getRemainingLifetime());

Oid[] mechs = cred.getMechs();
if (mechs != null) {
   for (int i = 0; i < mechs.length; i++)
       print(mechs[i].toString());
}
// release system resources held by the credential
cred.dispose();
<CODE ENDS>
```

7.4. public interface GSSContext

This interface encapsulates the GSS-API security context and provides the security services (wrap, unwrap, getMIC, verifyMIC) that are available over the context. Security contexts are established between peers using locally acquired credentials. Multiple contexts may exist simultaneously between a pair of peers, using the same or different set of credentials. GSS-API functions in a manner independent of the underlying transport protocol and depends on its calling application to transport its tokens between peers.

Before the context establishment phase is initiated, the context initiator may request specific characteristics desired of the established context. These can be set using the set methods. After the context is established, the caller can check the actual characteristic and services offered by the context using the query methods.

The context establishment phase begins with the first call to the init method by the context initiator. During this phase, the initSecContext and acceptSecContext methods will produce GSS-API authentication tokens, which the calling application needs to send to its peer. If an error occurs at any point, an exception will get thrown and the code will start executing in a catch block where the

exception may contain an output token that should be sent to the peer for debugging or informational purpose. If not, the normal flow of code continues and the application can make a call to the isEstablished() method. If this method returns "false" it indicates that a token is needed from its peer in order to continue the context establishment phase. A return value of "true" signals that the local end of the context is established. This may still require that a token be sent to the peer, if one is produced by GSS-API. During the context establishment phase, the isProtReady() method may be called to determine if the context can be used for the per-message operations. This allows applications to use per-message operations on contexts that aren't fully established.

After the context has been established or the isProtReady() method returns "true", the query routines can be invoked to determine the actual characteristics and services of the established context. The application can also start using the per-message methods of wrap and getMIC to obtain cryptographic operations on application supplied data.

When the context is no longer needed, the application SHOULD call dispose to release any system resources the context may be using.

7.4.1. Static Constants

```
public static final int DEFAULT_LIFETIME
```

A lifetime constant representing the default context lifetime. The value of this constant is 0.

```
public static final int INDEFINITE_LIFETIME
```

A lifetime constant representing indefinite context lifetime. The value of this constant is the maximum integer value in Java - Integer.MAX_VALUE.

7.4.2. initSecContext

```
public byte[] initSecContext(byte[] inputBuf, int offset, int len)
              throws GSSException
```

Called by the context initiator to start the context creation process. This method MAY return an output token that the application will need to send to the peer for processing by the accept call. The application can call isEstablished() to determine if the context establishment phase is complete for this peer. A return value of "false" from isEstablished() indicates that more tokens are expected to be supplied to the initSecContext() method. Note that it is

possible that the initSecContext() method will return a token for the peer and isEstablished() will return "true" also. This indicates that the token needs to be sent to the peer, but the local end of the context is now fully established.

Upon completion of the context establishment, the available context options may be queried through the get methods.

A GSSException will be thrown if the call fails. Users SHOULD call its getOutputToken() method to find out if there is a token that can be sent to the acceptor to communicate the reason for the error.

Parameters:

inputBuf	Token generated by the peer. This parameter is ignored on the first call.
offset	The offset within the inputBuf where the token begins.
len	The length of the token within the inputBuf (starting at the offset).

7.4.3. acceptSecContext

```
public byte[] acceptSecContext(byte[] inTok, int offset, int len)
           throws GSSException
```

Called by the context acceptor upon receiving a token from the peer.

This method MAY return an output token that the application will need to send to the peer for further processing by the init call.

The "null" return value indicates that no token needs to be sent to the peer. The application can call isEstablished() to determine if the context establishment phase is complete for this peer. A return value of "false" from isEstablished() indicates that more tokens are expected to be supplied to this method.

Note that it is possible that acceptSecContext() will return a token for the peer and isEstablished() will return "true" also. This indicates that the token needs to be sent to the peer, but the local end of the context is now fully established.

Upon completion of the context establishment, the available context options may be queried through the get methods.

A GSSException will be thrown if the call fails. Users SHOULD call its getOutputToken() method to find out if there is a token that can be sent to the initiator to communicate the reason for the error.

Parameters:

inTok	Token generated by the peer.
offset	The offset within the inTok where the token begins.
len	The length of the token within the inTok (starting at the offset).

7.4.4. isEstablished

```
public boolean isEstablished()
```

Used during context establishment to determine the state of the context. Returns "true" if this is a fully established context on the caller’s side and no more tokens are needed from the peer. Should be called after a call to initSecContext() or acceptSecContext() when no GSSException is thrown.

7.4.5. dispose

```
public void dispose() throws GSSException
```

Releases any system resources and cryptographic information stored in the context object. This will invalidate the context.

7.4.6. getWrapSizeLimit

```
public int getWrapSizeLimit(int qop, boolean confReq,
           int maxTokenSize) throws GSSException
```

Returns the maximum message size that, if presented to the wrap method with the same confReq and qop parameters, will result in an output token containing no more than the maxTokenSize bytes.

This call is intended for use by applications that communicate over protocols that impose a maximum message size. It enables the application to fragment messages prior to applying protection.

GSS-API implementations are RECOMMENDED but not required to detect invalid QOP values when getWrapSizeLimit is called. This routine guarantees only a maximum message size, not the availability of specific QOP values for message protection.

Successful completion of this call does not guarantee that wrap will be able to protect a message of the computed length, since this ability may depend on the availability of system resources at the time that wrap is called. However, if the implementation itself imposes an upper limit on the length of messages that may be processed by wrap, the implementation SHOULD NOT return a value that is greater than this length.

Parameters:

qop	Indicates the level of protection wrap will be asked to provide.
confReq	Indicates if wrap will be asked to provide privacy service.
maxTokenSize	The desired maximum size of the token emitted by wrap.

7.4.7. wrap

```
public byte[] wrap(byte[] inBuf, int offset, int len,
                   MessageProp msgProp) throws GSSException
```

Applies per-message security services over the established security context. The method will return a token with a cryptographic MIC and MAY optionally encrypt the specified inBuf. The returned byte array will contain both the MIC and the message.

The MessageProp object is instantiated by the application and used to specify a QOP value that selects cryptographic algorithms, and a privacy service to optionally encrypt the message. The underlying mechanism that is used in the call may not be able to provide the privacy service. It sets the actual privacy service that it does provide in this MessageProp object, which the caller SHOULD then query upon return. If the mechanism is not able to provide the requested QOP, it throws a GSSException with the BAD_QOP code.

Since some application-level protocols may wish to use tokens emitted by wrap to provide "secure framing", implementations SHOULD support the wrapping of zero-length messages.

The application will be responsible for sending the token to the peer.

Parameters:

inBuf	Application data to be protected.

offset — The offset within the inBuf where the data begins.

len — The length of the data within the inBuf (starting at the offset).

msgProp — Instance of MessageProp that is used by the application to set the desired QOP and privacy state. Set the desired QOP to 0 to request the default QOP. Upon return from this method, this object will contain the actual privacy state that was applied to the message by the underlying mechanism.

7.4.8. unwrap

```
public byte[] unwrap(byte[] inBuf, int offset, int len,
                     MessageProp msgProp) throws GSSException
```

Used by the peer application to process tokens generated with the wrap call. The method will return the message supplied in the peer application to the wrap call, verifying the embedded MIC.

The MessageProp object is instantiated by the application and is used by the underlying mechanism to return information to the caller such as the QOP, whether confidentiality was applied to the message, and other supplementary message state information.

Since some application-level protocols may wish to use tokens emitted by wrap to provide "secure framing", implementations SHOULD support the wrapping and unwrapping of zero-length messages.

Parameters:

inBuf — GSS-API wrap token received from peer.

offset — The offset within the inBuf where the token begins.

len — The length of the token within the inBuf (starting at the offset).

msgProp — Upon return from the method, this object will contain the applied QOP, the privacy state of the message, and supplementary information, described in section 5.12.3, stating whether the token was a duplicate, old, out of sequence, or arriving after a gap.

7.4.9. getMIC

```
public byte[] getMIC(byte[] inMsg, int offset, int len,
                     MessageProp msgProp) throws GSSException
```

Returns a token containing a cryptographic MIC for the supplied message for transfer to the peer application. Unlike wrap, which encapsulates the user message in the returned token, only the message MIC is returned in the output token.

Note that privacy can only be applied through the wrap call.

Since some application-level protocols may wish to use tokens emitted by getMIC to provide "secure framing", implementations SHOULD support derivation of MICs from zero-length messages.

Parameters:

inMsg	Message over which to generate MIC.
offset	The offset within the inMsg where the token begins.
len	The length of the token within the inMsg (starting at the offset).
msgProp	Instance of MessageProp that is used by the application to set the desired QOP. Set the desired QOP to 0 in msgProp to request the default QOP. Alternatively, pass in "null" for msgProp to request default QOP.

7.4.10. verifyMIC

```
public void verifyMIC(byte[] inTok, int tokOffset, int tokLen,
                      byte[] inMsg, int msgOffset, int msgLen,
                      MessageProp msgProp) throws GSSException
```

Verifies the cryptographic MIC, contained in the token parameter, over the supplied message.

The MessageProp object is instantiated by the application and is used by the underlying mechanism to return information to the caller such as the QOP indicating the strength of protection that was applied to the message and other supplementary message state information.

Since some application-level protocols may wish to use tokens emitted by getMIC to provide "secure framing", implementations SHOULD support the calculation and verification of MICs over zero-length messages.

Parameters:

inTok	Token generated by peer’s getMIC method.
tokOffset	The offset within the inTok where the token begins.
tokLen	The length of the token within the inTok (starting at the offset).
inMsg	Application message over which to verify the cryptographic MIC.
msgOffset	The offset within the inMsg where the message begins.
msgLen	The length of the message within the inMsg (starting at the offset).
msgProp	Upon return from the method, this object will contain the applied QOP and supplementary information, described in section 5.12.3, stating whether the token was a duplicate, old, out of sequence, or arriving after a gap. The confidentiality state will be set to "false".

7.4.11. export

public byte[] export() throws GSSException

Provided to support the sharing of work between multiple processes. This routine will typically be used by the context acceptor, in an application where a single process receives incoming connection requests and accepts security contexts over them, then passes the established context to one or more other processes for message exchange.

This method deactivates the security context and creates an inter-process token which, when passed to the byte array constructor of the GSSContext interface in another process, will re-activate the context in the second process. Only a single instantiation of a given context may be active at any one time; a subsequent attempt by a context exporter to access the exported security context will fail.

The implementation MAY constrain the set of processes by which the inter-process token may be imported, either as a function of local security policy, or as a result of implementation decisions. For example, some implementations may constrain contexts to be passed only between processes that run under the same account, or which are part of the same process group.

The inter-process token MAY contain security-sensitive information (for example, cryptographic keys). While mechanisms are encouraged to either avoid placing such sensitive information within inter-process tokens or to encrypt the token before returning it to the application, in a typical GSS-API implementation, this may not be possible. Thus, the application MUST take care to protect the inter-process token, and ensure that any process to which the token is transferred is trustworthy.

7.4.12. requestMutualAuth

```
public void requestMutualAuth(boolean state) throws GSSException
```

Sets the request state of the mutual authentication flag for the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state	Boolean representing if mutual authentication should be requested during context establishment.

7.4.13. requestReplayDet

```
public void requestReplayDet(boolean state) throws GSSException
```

Sets the request state of the replay detection service for the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state	Boolean representing if replay detection is desired over the established context.

7.4.14. requestSequenceDet

```
public void requestSequenceDet(boolean state) throws GSSException
```

Sets the request state for the sequence checking service of the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state Boolean representing if sequence detection is desired over the established context.

7.4.15. requestCredDeleg

public void requestCredDeleg(boolean state) throws GSSException

Sets the request state for the credential delegation flag for the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state Boolean representing if credential delegation is desired.

7.4.16. requestAnonymity

public void requestAnonymity(boolean state) throws GSSException

Requests anonymous support over the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state Boolean representing if anonymity support is requested.

7.4.17. requestConf

public void requestConf(boolean state) throws GSSException

Requests that confidentiality service be available over the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state Boolean indicating if confidentiality services are to be requested for the context.

7.4.18. requestInteg

public void requestInteg(boolean state) throws GSSException

Requests that integrity services be available over the context. This method is only valid before the context creation process begins and only for the initiator.

Parameters:

state	Boolean indicating if integrity services are to be requested for the context.

7.4.19. requestLifetime

public void requestLifetime(int lifetime) throws GSSException

Sets the desired lifetime for the context in seconds. This method is only valid before the context creation process begins and only for the initiator. Use GSSContext.INDEFINITE_LIFETIME and GSSContext.DEFAULT_LIFETIME to request indefinite or default context lifetime.

Parameters:

lifetime	The desired context lifetime in seconds.

7.4.20. setChannelBinding

public void setChannelBinding(ChannelBinding cb) throws GSSException

Sets the channel bindings to be used during context establishment. This method is only valid before the context creation process begins.

Parameters:

cb	Channel bindings to be used.

7.4.21. getCredDelegState

public boolean getCredDelegState()

Returns the state of the delegated credentials for the context. When issued before context establishment is completed or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.22. getMutualAuthState

public boolean getMutualAuthState()

Returns the state of the mutual authentication option for the context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.23. getReplayDetState

public boolean getReplayDetState()

Returns the state of the replay detection option for the context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.24. getSequenceDetState

public boolean getSequenceDetState()

Returns the state of the sequence detection option for the context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.25. getAnonymityState

public boolean getAnonymityState()

Returns "true" if this is an anonymous context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.26. isTransferable

public boolean isTransferable() throws GSSException

Returns "true" if the context is transferable to other processes through the use of the export method. This call is only valid on fully established contexts.

7.4.27. isProtReady

public boolean isProtReady()

Returns "true" if the per-message operations can be applied over the context. Some mechanisms may allow the usage of per-message operations before the context is fully established. This will also indicate that the get methods will return actual context state characteristics instead of the desired ones.

7.4.28. getConfState

public boolean getConfState()

Returns the confidentiality service state over the context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.29. getIntegState

public boolean getIntegState()

Returns the integrity service state over the context. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired state; otherwise, it will indicate the actual state over the established context.

7.4.30. getLifetime

public int getLifetime()

Returns the context lifetime in seconds. When issued before context establishment completes or when the isProtReady method returns "false", it returns the desired lifetime; otherwise, it will indicate the remaining lifetime for the context.

7.4.31. getSrcName

public GSSName getSrcName() throws GSSException

Returns the name of the context initiator. This call is valid only after the context is fully established or the isProtReady method returns "true". It is guaranteed to return an MN.

7.4.32. getTargName

public GSSName getTargName() throws GSSException

Returns the name of the context target (acceptor). This call is valid only after the context is fully established or the isProtReady method returns "true". It is guaranteed to return an MN.

7.4.33. getMech

public Oid getMech() throws GSSException

Returns the mechanism oid for this context. This method MAY be called before the context is fully established, but the mechanism returned MAY change on successive calls in negotiated mechanism case.

7.4.34. getDelegCred

public GSSCredential getDelegCred() throws GSSException

Returns the delegated credential object on the acceptor’s side. To check for availability of delegated credentials call getDelegCredState. This call is only valid on fully established contexts.

7.4.35. isInitiator

public boolean isInitiator() throws GSSException

Returns "true" if this is the initiator of the context. This call is only valid after the context creation process has started.

7.4.36. Example Code

The example code presented below demonstrates the usage of the GSSContext interface for the initiating peer. Different operations on the GSSContext object are presented, including: object instantiation, setting of desired flags, context establishment, query of actual context flags, per-message operations on application data, and finally context deletion.

```
<CODE BEGINS>
GSSManager mgr = GSSManager.getInstance();

// start by creating the name for a service entity
GSSName targetName = mgr.createName("service@host",
                     GSSName.NT_HOSTBASED_SERVICE);
// create a context using default credentials for the above entity
```

```
   // and the implementation-specific default mechanism
   GSSContext context = mgr.createContext(targetName,
                   null,   /* default mechanism */
                   null,   /* default credentials */
                   GSSContext.INDEFINITE_LIFETIME);

   // set desired context options - all others are "false" by default
   context.requestConf(true);
   context.requestMutualAuth(true);
   context.requestReplayDet(true);
   context.requestSequenceDet(true);

   // establish a context between peers - using byte arrays
   byte[]inTok = new byte[0];

   try {
       do {
           byte[] outTok = context.initSecContext(inTok, 0,
                                                  inTok.length);

           // send the token if present
           if (outTok != null)
               sendToken(outTok);

           // check if we should expect more tokens
           if (context.isEstablished())
               break;

           // another token expected from peer
           inTok = readToken();

       } while (true);

   } catch (GSSException e) {
       print("GSSAPI error: " + e.getMessage());

       // If the exception contains an output token,
       // it should be sent to the acceptor.
       byte[] outTok = e.getOutputToken();
       if (outTok != null) {
           sendToken(outTok);
       }

       return;
   }

   // display context information
   print("Remaining lifetime in seconds = " + context.getLifetime());
```

```
print("Context mechanism = " + context.getMech().toString());
print("Initiator = " + context.getSrcName().toString());
print("Acceptor = " + context.getTargName().toString());

if (context.getConfState())
    print("Confidentiality security service available");

if (context.getIntegState())
    print("Integrity security service available");

// perform wrap on an application-supplied message, appMsg,
// using QOP = 0, and requesting privacy service
byte[] appMsg ...

MessageProp mProp = new MessageProp(0, true);

byte[] tok = context.wrap(appMsg, 0, appMsg.length, mProp);

if (mProp.getPrivacy())
    print("Message protected with privacy.");

sendToken(tok);

// release the local end of the context
context.dispose();
<CODE ENDS>
```

7.5. public class MessageProp

This is a utility class used within the per-message GSSContext methods to convey per-message properties.

When used with the GSSContext interface’s wrap and getMIC methods, an instance of this class is used to indicate the desired QOP and to request if confidentiality services are to be applied to caller supplied data (wrap only). To request default QOP, the value of 0 should be used for QOP. A QOP is an integer value defined by an mechanism.

When used with the unwrap and verifyMIC methods of the GSSContext interface, an instance of this class will be used to indicate the applied QOP and confidentiality services over the supplied message. In the case of verifyMIC, the confidentiality state will always be "false". Upon return from these methods, this object will also contain any supplementary status values applicable to the processed token. The supplementary status values can indicate old tokens, out of sequence tokens, gap tokens, or duplicate tokens.

7.5.1. Constructors

public MessageProp(boolean privState)

Constructor that sets QOP to 0 indicating that the default QOP is requested.

Parameters:

privState	The desired privacy state. "true" for privacy and "false" for integrity only.

public MessageProp(int qop, boolean privState)

Constructor that sets the values for the qop and privacy state.

Parameters:

qop	The desired QOP. Use 0 to request a default QOP.
privState	The desired privacy state. "true" for privacy and "false" for integrity only.

7.5.2. getQOP

public int getQOP()

Retrieves the QOP value.

7.5.3. getPrivacy

public boolean getPrivacy()

Retrieves the privacy state.

7.5.4. getMinorStatus

public int getMinorStatus()

Retrieves the minor status that the underlying mechanism might have set.

7.5.5. getMinorString

public String getMinorString()

Returns a string explaining the mechanism-specific error code. "null" will be returned when no mechanism error code has been set.

7.5.6. setQOP

public void setQOP(int qopVal)

Sets the QOP value.

Parameters:

qopVal	The QOP value to be set. Use 0 to request a default QOP value.

7.5.7. setPrivacy

public void setPrivacy(boolean privState)

Sets the privacy state.

Parameters:

privState	The privacy state to set.

7.5.8. isDuplicateToken

public boolean isDuplicateToken()

Returns "true" if this is a duplicate of an earlier token.

7.5.9. isOldToken

public boolean isOldToken()

Returns "true" if the token’s validity period has expired.

7.5.10. isUnseqToken

public boolean isUnseqToken()

Returns "true" if a later token has already been processed.

7.5.11. isGapToken

public boolean isGapToken()

Returns "true" if an expected per-message token was not received.

7.5.12. setSupplementaryStates

```
public void setSupplementaryStates(boolean duplicate,
               boolean old, boolean unseq, boolean gap,
               int minorStatus, String minorString)
```

This method sets the state for the supplementary information flags and the minor status in MessageProp. It is not used by the application but by the GSS implementation to return this information to the caller of a per-message context method.

Parameters:

duplicate	"true" if the token was a duplicate of an earlier token; otherwise, "false".
old	"true" if the token’s validity period has expired; otherwise, "false".
unseq	"true" if a later token has already been processed; otherwise, "false".
gap	"true" if one or more predecessor tokens have not yet been successfully processed; otherwise, "false".
minorStatus	The integer minor status code that the underlying mechanism wants to set.
minorString	The textual representation of the minorStatus value.

7.6. public class ChannelBinding

The GSS-API accommodates the concept of caller-provided channel binding information. Channel bindings are used to strengthen the quality with which peer entity authentication is provided during context establishment. They enable the GSS-API callers to bind the establishment of the security context to relevant characteristics like addresses or to application-specific data.

The caller initiating the security context MUST determine the appropriate channel binding values to set in the GSSContext object. The acceptor MUST provide an identical binding in order to validate that received tokens possess correct channel-related characteristics.

Use of channel bindings is OPTIONAL in GSS-API. Since channel-binding information may be transmitted in context establishment

tokens, applications SHOULD therefore not use confidential data as channel-binding components.

7.6.1. Constructors

```
public ChannelBinding(InetAddress initAddr, InetAddress acceptAddr,
                      byte[] appData)
```

Create a ChannelBinding object with user-supplied address information and data. "null" values can be used for any fields that the application does not want to specify.

Parameters:

initAddr	The address of the context initiator. "null" value can be supplied to indicate that the application does not want to set this value.
acceptAddr	The address of the context acceptor. "null" value can be supplied to indicate that the application does not want to set this value.
appData	Application-supplied data to be used as part of the channel bindings. "null" value can be supplied to indicate that the application does not want to set this value.

```
public ChannelBinding(byte[] appData)
```

Creates a ChannelBinding object without any addressing information.

Parameters:

appData	Application supplied data to be used as part of the channel bindings.

7.6.2. getInitiatorAddress

```
public InetAddress getInitiatorAddress()
```

Returns the initiator’s address for this channel binding. "null" is returned if the address has not been set.

7.6.3. getAcceptorAddress

```
public InetAddress getAcceptorAddress()
```

Returns the acceptor’s address for this channel binding. "null" is returned if the address has not been set.

7.6.4. getApplicationData

public byte[] getApplicationData()

Returns application data being used as part of the ChannelBinding. "null" is returned if no application data has been specified for the channel binding.

7.6.5. equals

public boolean equals(Object obj)

Returns "true" if two channel bindings match. (Note that the Java language specification requires that two objects that are equal according to the equals(Object) method MUST return the same integer result when the hashCode() method is called on them.)

Parameters:

obj Another channel binding with which to compare.

7.7. public class Oid

This class represents Universal Object Identifiers (Oids) and their associated operations.

Oids are hierarchically globally interpretable identifiers used within the GSS-API framework to identify mechanisms and name formats.

The structure and encoding of Oids is defined in ISOIEC-8824 and ISOIEC-8825. For example, the Oid representation of the Kerberos v5 mechanism is "1.2.840.113554.1.2.2".

The GSSName name class contains public static Oid objects representing the standard name types defined in GSS-API.

7.7.1. Constructors

public Oid(String strOid) throws GSSException

Creates an Oid object from a string representation of its integer components (e.g., "1.2.840.113554.1.2.2").

Parameters:

strOid The string representation for the oid.

public Oid(InputStream derOid) throws GSSException

Creates an Oid object from its DER encoding. This refers to the full encoding including tag and length. The structure and encoding of Oids is defined in ISOIEC-8824 and ISOIEC-8825. This method is identical in functionality to its byte array counterpart.

Parameters:

derOid Stream containing the DER-encoded oid.

public Oid(byte[] DEROid) throws GSSException

Creates an Oid object from its DER encoding. This refers to the full encoding including tag and length. The structure and encoding of Oids is defined in ISOIEC-8824 and ISOIEC-8825. This method is identical in functionality to its byte array counterpart.

Parameters:

derOid Byte array storing a DER-encoded oid.

7.7.2. toString

public String toString()

Returns a string representation of the oid’s integer components in dot separated notation (e.g., "1.2.840.113554.1.2.2").

7.7.3. equals

public boolean equals(Object Obj)

Returns "true" if the two Oid objects represent the same oid value. (Note that the Java language specification [JLS] requires that two objects that are equal according to the equals(Object) method MUST return the same integer result when the hashCode() method is called on them.)

Parameters:

obj Another Oid object with which to compare.

7.7.4. getDER

public byte[] getDER()

Returns the full ASN.1 DER encoding for this oid object, which includes the tag and length.

7.7.5. containedIn

public boolean containedIn(Oid[] oids)

A utility method to test if an Oid object is contained within the supplied Oid object array.

Parameters:

oids An array of oids to search.

7.8. public class GSSException extends Exception

This exception is thrown whenever a fatal GSS-API error occurs including mechanism-specific errors. It MAY contain both, the major and minor, GSS-API status codes. The mechanism implementors are responsible for setting appropriate minor status codes when throwing this exception. Aside from delivering the numeric error code(s) to the caller, this class performs the mapping from their numeric values to textual representations. This exception MAY also include an output token that SHOULD be sent to the peer. For example, when an initSecContext call fails due to a fatal error, the mechanism MAY define an error token that SHOULD be sent to the peer for debugging or informational purpose. All Java GSS-API methods are declared throwing this exception.

All implementations are encouraged to use the Java internationalization techniques to provide local translations of the message strings.

7.8.1. Static Constants

All valid major GSS-API error code values are declared as constants in this class.

public static final int BAD_BINDINGS

Channel bindings mismatch error. The value of this constant is 1.

public static final int BAD_MECH

Unsupported mechanism requested error. The value of this constant is 2.

public static final int BAD_NAME

Invalid name provided error. The value of this constant is 3.

public static final int BAD_NAMETYPE

Name of unsupported type provided error. The value of this constant is 4.

public static final int BAD_STATUS

Invalid status code error - this is the default status value. The value of this constant is 5.

public static final int BAD_MIC

Token had invalid integrity check error. The value of this constant is 6.

public static final int CONTEXT_EXPIRED

Specified security context expired error. The value of this constant is 7.

public static final int CREDENTIALS_EXPIRED

Expired credentials detected error. The value of this constant is 8.

public static final int DEFECTIVE_CREDENTIAL

Defective credential error. The value of this constant is 9.

public static final int DEFECTIVE_TOKEN

Defective token error. The value of this constant is 10.

public static final int FAILURE

General failure, unspecified at GSS-API level. The value of this constant is 11.

public static final int NO_CONTEXT

Invalid security context error. The value of this constant is 12.

public static final int NO_CRED

Invalid credentials error. The value of this constant is 13.

public static final int BAD_QOP

Unsupported QOP value error. The value of this constant is 14.

public static final int UNAUTHORIZED

Operation unauthorized error. The value of this constant is 15.

public static final int UNAVAILABLE

Operation unavailable error. The value of this constant is 16.

public static final int DUPLICATE_ELEMENT

Duplicate credential element requested error. The value of this constant is 17.

public static final int NAME_NOT_MN

Name contains multi-mechanism elements error. The value of this constant is 18.

public static final int DUPLICATE_TOKEN

The token was a duplicate of an earlier token. This is contained in an exception only when detected during context establishment, in which case it is considered a fatal error. (Non-fatal supplementary codes are indicated via the MessageProp object.) The value of this constant is 19.

public static final int OLD_TOKEN

The token’s validity period has expired. This is contained in an exception only when detected during context establishment, in which case it is considered a fatal error. (Non-fatal supplementary codes are indicated via the MessageProp object.) The value of this constant is 20.

public static final int UNSEQ_TOKEN

A later token has already been processed. This is contained in an exception only when detected during context establishment, in which case it is considered a fatal error. (Non-fatal supplementary codes

are indicated via the MessageProp object.) The value of this constant is 21.

public static final int GAP_TOKEN

An expected per-message token was not received. This is contained in an exception only when detected during context establishment, in which case it is considered a fatal error. (Non-fatal supplementary codes are indicated via the MessageProp object.) The value of this constant is 22.

7.8.2. Constructors

public GSSException(int majorCode)

Creates a GSSException object with a specified major code.

Calling this constructor is equivalent to calling GSSException(majorCode, null, 0, null, null).

public GSSException(int majorCode, int minorCode, String minorString)

Creates a GSSException object with the specified major code, minor code, and minor code textual explanation. This constructor is to be used when the exception is originating from the security mechanism. It allows to specify the GSS code and the mechanism code.

Calling this constructor is equivalent to calling GSSException(majorCode, null, minorCode, minorString, null).

```
public GSSException(int majorCode, String majorString,
                    int minorCode, String minorString,
                    byte[] outputToken)
```

Creates a GSSException object with the specified major code, major code textual explanation, minor code, minor code textual explanation, and an output token. This is a general-purpose constructor that can be used to create any type of GSSException.

Parameters:

majorCode	The GSS error code causing this exception to be thrown.
majorString	The textual explanation of the GSS error code. If null is provided, a default explanation that matches the majorCode will be set.

minorCode	The mechanism error code causing this exception to be thrown. Can be 0 if no mechanism error code is available.
minorString	The textual explanation of the mechanism error code. Can be null if no textual explanation is available.
outputToken	The output token that SHOULD be sent to the peer. Can be null if no such token is available. It MUST NOT be an empty array. When provided, the array will be cloned to protect against subsequent modifications.

7.8.3. getMajor

```
public int getMajor()
```

Returns the major code representing the GSS error code that caused this exception to be thrown.

7.8.4. getMinor

```
public int getMinor()
```

Returns the mechanism error code that caused this exception. The minor code is set by the underlying mechanism. Value of 0 indicates that mechanism error code is not set.

7.8.5. getMajorString

```
public String getMajorString()
```

Returns a string explaining the GSS major error code causing this exception to be thrown.

7.8.6. getMinorString

```
public String getMinorString()
```

Returns a string explaining the mechanism-specific error code. "null" will be returned when no string explaining the mechanism error code has been set.

7.8.7. getOutputToken

public byte[] getOutputToken

Returns the output token in a new byte array.

If the method (For example, GSSContext#initSecContext) that throws this GSSException needs to generate an output token that SHOULD be sent to the peer, that token will be stored in this GSSException and can be retrieved with this method.

The return value MUST be null if no such token is generated. It MUST NOT be an empty byte array.

7.8.8. setMinor

public void setMinor(int minorCode, String message)

Used internally by the GSS-API implementation and the underlying mechanisms to set the minor code and its textual representation.

Parameters:

minorCode	The mechanism-specific error code.
message	A textual explanation of the mechanism error code.

7.8.9. toString

public String toString()

Returns a textual representation of both the major and minor status codes.

7.8.10. getMessage

public String getMessage()

Returns a detailed message of this exception. Overrides Throwable.getMessage. It is customary in Java to use this method to obtain exception information.

8. Sample Applications

8.1. Simple GSS Context Initiator

```
<CODE BEGINS>
import org.ietf.jgss.*;

/**
 * This is a partial sketch for a simple client program that acts
 * as a GSS context initiator.  It illustrates how to use the Java
 * bindings for the GSS-API specified in
 * Generic Security Service API Version 2 : Java bindings
 *
 *
 * This code sketch assumes the existence of a GSS-API
 * implementation that supports the mechanism that it will need
 * and is present as a library package (org.ietf.jgss) either as
 * part of the standard JRE or in the CLASSPATH the application
 * specifies.
 */

 public class SimpleClient {

     private String serviceName; // name of peer (i.e., server)
     private GSSCredential clientCred = null;
     private GSSContext context = null;
     private Oid mech; // underlying mechanism to use

     private GSSManager mgr = GSSManager.getInstance();

     ...
     ...

     private void clientActions() {
        initializeGSS();
        establishContext();
        doCommunication();
     }

    /**
     * Acquire credentials for the client.
     */
    private void initializeGSS() {

        try {

            clientCred = mgr.createCredential(null /*default princ*/,
                GSSCredential.INDEFINITE_LIFETIME /* max lifetime */,
                mech /* mechanism to use */,
                GSSCredential.INITIATE_ONLY /* init context */);
```

```
                print("GSSCredential created for " +
                        cred.getName().toString());
                print("Credential lifetime (sec)=" +
                        cred.getRemainingLifetime());
            } catch (GSSException e) {
                print("GSS-API error in credential acquisition: "
                      + e.getMessage());
                ...
                ...
            }
            ...
            ...
        }

        /**
         * Does the security context establishment with the
         * server.
         */
        private void establishContext() {

            byte[] inToken = new byte[0];
            byte[] outToken = null;

            try {

                GSSName peer = mgr.createName(serviceName,
                                      GSSName.NT_HOSTBASED_SERVICE);
                context = mgr.createContext(peer, mech, gssCred,
                       GSSContext.INDEFINITE_LIFETIME/*lifetime*/);

                // Will need to support confidentiality
                context.requestConf(true);

                while (!context.isEstablished()) {

                    outToken = context.initSecContext(inToken, 0,
                                                      inToken.length);

                    if (outToken != null)
                        writeGSSToken(outToken);

                    if (!context.isEstablished())
                        inToken = readGSSToken();
                }

                GSSName peer = context.getSrcName();
                print("Security context established with " + peer +
                      " using underlying mechanism " + mech.toString());
```

```
            } catch (GSSException e) {
                 print("GSS-API error during context establishment: "
                       + e.getMessage());

                 // If the exception contains an output token,
                 // it should be sent to the acceptor.
                 byte[] outTok = e.getOutputToken();
                 if (outTok != null) {
                     writeGSSToken(outTok);
                 }
                 ...
                 ...
            }
            ...
            ...
        }

        /**
         * Sends some data to the server and reads back the
         * response.
         */
        private void doCommunication()  {
            byte[] inToken = null;
            byte[] outToken = null;
            byte[] buffer;

            // Container for multiple input-output arguments to and
            // from the per-message routines (e.g., wrap/unwrap).
            MessageProp messgInfo = new MessageProp();

            try {

                /*
                 * Now send some bytes to the server to be
                 * processed.  They will be integrity protected
                 * but not encrypted for privacy.
                 */

                buffer = readFromFile();

                // Set privacy to "false" and use the default QOP
                messgInfo.setPrivacy(false);

                outToken = context.wrap(buffer, 0, buffer.length,
                                        messgInfo);

                writeGSSToken(outToken);
```

```
                /*
                 * Now read the response from the server.
                 */

                inToken = readGSSToken();
                buffer = context.unwrap(inToken, 0,
                             inToken.length, messgInfo);
                // All ok if no exception was thrown!

                GSSName peer = context.getSrcName();

                print("Message from "  + peer.toString()
                      + " arrived.");
                print("Was it encrypted? "  +
                      messgInfo.getPrivacy());
                print("Duplicate Token? "   +
                      messgInfo.isDuplicateToken());
                print("Old Token? "         +
                      messgInfo.isOldToken());
                print("Unsequenced Token? " +
                      messgInfo.isUnseqToken());
                print("Gap Token? "         +
                      messgInfo.isGapToken());
                ...
                ...
            } catch (GSSException e) {
                print("GSS-API error in per-message calls: "
                      + e.getMessage());
                ...
                ...
            }
            ...
            ...
        } // end of doCommunication method

        ...
        ...

   } // end of class SimpleClient
   <CODE ENDS>
```

8.2. Simple GSS Context Acceptor

```
   <CODE BEGINS>
   import org.ietf.jgss.*;

   /**
    * This is a partial sketch for a simple server program that acts
```

```
    * as a GSS context acceptor.  It illustrates how to use the Java
    * bindings for the GSS-API specified in
    * Generic Security Service API Version 2 : Java bindings.
    *
    * This code sketch assumes the existence of a GSS-API
    * implementation that supports the mechanisms that it will need
    * and is present as a library package (org.ietf.jgss) either as
    * part of the standard JRE or in the CLASSPATH the application
    * specifies.
    */

   import org.ietf.jgss.*;

   public class SimpleServer {

       private String serviceName;
       private GSSName name;
       private GSSCredential cred;

       private GSSManager mgr;

       ...
       ...

       /**
        * Wait for client connections, establish security contexts
        * and provide service.
        */
       private void loop() {
           ...
           ...
           mgr = GSSManager.getInstance();

           name = mgr.createName(serviceName,
                     GSSName.NT_HOSTBASED_SERVICE);

           cred = mgr.createCredential(name,
                     GSSCredential.INDEFINITE_LIFETIME,
                     null,
                     GSSCredential.ACCEPT_ONLY);

           // Loop infinitely
           while (true) {
               Socket s = serverSock.accept();

               // Start a new thread to serve this connection
               Thread serverThread = new ServerThread(s);
               serverThread.start();
```

```
            }
        }

        /**
         * Inner class ServerThread whose run() method provides the
         * secure service to a connection.
         */

        private class ServerThread extends Thread {

            ...
            ...

            /**
             * Deals with the connection from one client.  It also
             * handles all GSSException’s thrown while talking to
             * this client.
             */
            public void run() {

                byte[] inToken = null;
                byte[] outToken = null;
                byte[] buffer;

                GSSName peer;

                // Container for multiple input-output arguments to
                // and from the per-message routines
                // (i.e., wrap/unwrap).
                MessageProp supplInfo = new MessageProp();
                GSSContext secContext = null;

                try {
                    // Now do the context establishment loop
                    GSSContext context = mgr.createContext(cred);

                    while (!context.isEstablished()) {

                        inToken = readGSSToken();
                        outToken = context.acceptSecContext(inToken,
                                                  0, inToken.length);
                        if (outToken != null)
                            writeGSSToken(outToken);
                    }

                    // SimpleServer wants confidentiality to be
                    // available.  Check for it.
                    if (!context.getConfState()){
```

```
                    ...
                    ...
                }

                GSSName peer = context.getSrcName();
                Oid mech = context.getMech();
                print("Security context established with " +
                       peer.toString() +
                      " using underlying mechanism " +
                      mech.toString() +
                      " from Provider " +
                      context.getProvider().getName());

                // Now read the bytes sent by the client to be
                // processed.
                inToken = readGSSToken();

                // Unwrap the message
                buffer = context.unwrap(inToken, 0,
                            inToken.length, supplInfo);
                // All ok if no exception was thrown!

                // Print other supplementary per-message status
                // information.

                print("Message from " +
                        peer.toString() + " arrived.");
                print("Was it encrypted? " +
                        supplInfo.getPrivacy());
                print("Duplicate Token? " +
                        supplInfo.isDuplicateToken());
                print("Old Token? "  + supplInfo.isOldToken());
                print("Unsequenced Token? " +
                        supplInfo.isUnseqToken());
                print("Gap Token? "  + supplInfo.isGapToken());

                /*
                 * Now process the bytes and send back an
                 * encrypted response.
                 */

                buffer = serverProcess(buffer);

                // Encipher it and send it across

                supplInfo.setPrivacy(true); // privacy requested
                supplInfo.setQOP(0); // default QOP
                outToken = context.wrap(buffer, 0, buffer.length,
```

```
                                                supplInfo);
                    writeGSSToken(outToken);

                } catch (GSSException e) {
                    print("GSS-API Error: " + e.getMessage());
                    // Alternatively, could call e.getMajorMessage()
                    // and e.getMinorMessage()

                    // If the exception contains an output token,
                    // it should be sent to the initiator.
                    byte[] outTok = e.getOutputToken();
                    if (outTok != null) {
                        writeGSSToken(outTok);
                    }
                    print("Abandoning security context.");
                    ...
                    ...
                }
                ...
                ...
            } // end of run method in ServerThread

        } // end of inner class ServerThread

        ...
        ...

    } // end of class SimpleServer
    <CODE ENDS>
```

9. Security Considerations

The Java language security model allows platform providers to have policy-based fine-grained access control over any resource that an application wants. When using a Java security manager (such as, but not limited to, the case of applets running in browsers) the application code is in a sandbox by default.

Administrators of the platform JRE determine what permissions, if any, are to be given to source from different codebases. Thus, the administrator has to be aware of any special requirements that the GSS provider might have for system resources. For instance, a Kerberos provider might wish to make a network connection to the Key Distribution Center (KDC) to obtain initial credentials. This would not be allowed under the sandbox unless the administrator had granted permissions for this. Also, note that this granting and checking of permissions happens transparently to the application and is outside the scope of this document.

The Java language allows administrators to pre-configure a list of security service providers in the <JRE>/lib/security/java.security file. At runtime, the system approaches these providers in order of preference when looking for security related services. Applications have a means to modify this list through methods in the "Security" class in the "java.security" package. However, since these modifications would be visible in the entire Java Virtual Machine (JVM) and thus affect all code executing in it, this operation is not available in the sandbox and requires special permissions to perform. Thus, when a GSS application has special needs that are met by a particular security provider, it has two choices:

1) To install the provider on a JVM-wide basis using the java.security.Security class and then depend on the system to find the right provider automatically when the need arises. (This would require the application to be granted a "insertProvider SecurityPermission".)

2) To pass an instance of the provider to the local instance of GSSManager so that only factory calls going through that GSSManager use the desired provider. (This would not require any permissions.)

10. IANA Considerations

This document has no actions for IANA.

11. Acknowledgments

This proposed API leverages earlier work performed by the IETF’s CAT WG as outlined in both RFC 2743 [RFC2743] and RFC 2744 [RFC2744]. Many conceptual definitions, implementation directions, and explanations have been included from these documents.

We would like to thank Mike Eisler, Lin Ling, Ram Marti, Michael Saltz, and other members of Sun’s development team for their helpful input, comments, and suggestions.

We would also like to thank Joe Salowey, and Michael Smith for many insightful ideas and suggestions that have contributed to this document.

12. Changes since RFC 5653

This document has following changes:

1) New error token embedded in GSSException

There is a design flaw in the initSecContext and acceptSecContext methods of the GSSContext class defined in Generic Security Service API Version 2: Java Bindings Update [RFC5653].

The methods could either return a token (possibly null if no more tokens are needed) when the call succeeds or throw a GSSException if there is a failure, but NOT both. On the other hand, the C bindings of GSS-API [RFC2744] can return both, that is to say, a call to the GSS_Init_sec_context() function can return a major status code, and at the same time, fill in the output_token argument if there is one.

Without the ability to emit an error token when there is a failure, a Java application has no mechanism to tell the other side what the error is. For example, a "reject" NegTokenResp token can never be transmitted for the SPNEGO mechanism [RFC4178].

While a Java method can never return a value and throw an exception at the same time, we can embed the error token inside the exception so that the caller has a chance to retrieve it. This update adds a new GSSException constructor to include this token inside a GSSException object, and a getOutputToken() method to retrieve the token. The specification for the initSecContext and acceptSecContext methods are updated to describe the new behavior. Various examples are also updated.

New JGSS programs SHOULD make use of this new feature but it is not mandatory. A program that intends to run with both old and new GSS Java bindings can use reflection to check the availability of this new method and call it accordingly.

2) Removing stream-based GSSContext methods

The overloaded methods of GSSContext that use input and output streams as the means to convey authentication and per-message GSS-API tokens as described in Section 5.15 of RFC 5653 [RFC5653] are removed in this update as the wire protocol should be defined by an application and not a library. It’s also impossible to implement these methods correctly when the token has no self-framing (where the end cannot be determined) or the library has no knowledge of the token format (for example, as a bridge talking to another GSS library). These methods include initSecContext (Section 7.4.5 of RFC 5653 [RFC5653]), acceptSecContext (Section 7.4.9 of RFC 5653 [RFC5653]), wrap (Section 7.4.15 of RFC 5653 [RFC5653]), unwrap (Section 7.4.17 of RFC 5653 [RFC5653]), getMIC (Section 7.4.19 of RFC 5653 [RFC5653]), and verifyMIC (Section 7.4.21 of RFC 5653 [RFC5653]).

13. Changes since RFC 2853

This document has following changes:

1) Major GSS Status Code Constant Values

   RFC 2853 listed all the GSS status code values in two different sections: section 4.12.1 defined numeric values for them, and section 6.8.1 defined them as static constants in the GSSException class without assigning any values. Due to an inconsistent ordering between these two sections, all of the GSS major status codes resulted in misalignment, and a subsequent disagreement between deployed implementations.

   This document defines the numeric values of the GSS status codes in both sections, while maintaining the original ordering from section 6.8.1 of RFC 2853 [RFC2853], and obsoletes the GSS status code values defined in section 4.12.1. The relevant sections in this document are sections 5.12.1 and 7.8.1.

2) GSS Credential Usage Constant Values

   RFC 2853 section 6.3.2 defines static constants for the GSSCredential usage flags. However, the values of these constants were not defined anywhere in RFC 2853 [RFC2853].

   This document defines the credential usage values in section 7.3.1. The original ordering of these values from section 6.3.2 of RFC 2853 [RFC2853] is maintained.

3) GSS Host-Based Service Name

   RFC 2853 [RFC2853], section 6.2.2, defines the static constant for the GSS host-based service OID NT_HOSTBASED_SERVICE, using a deprecated OID value.

   This document updates the NT_HOSTBASED_SERVICE OID value in section 7.2.1 to be consistent with the C-bindings in RFC 2744 [RFC2744].

14. References

14.1. Normative References

[RFC2025] Adams, C., "The Simple Public-Key GSS-API Mechanism (SPKM)", RFC 2025, DOI 10.17487/RFC2025, October 1996, <https://www.rfc-editor.org/info/rfc2025>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <https://www.rfc-editor.org/info/rfc2743>.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, DOI 10.17487/RFC2744, January 2000, <https://www.rfc-editor.org/info/rfc2744>.

[RFC2853] Kabat, J. and M. Upadhyay, "Generic Security Service API Version 2 : Java Bindings", RFC 2853, DOI 10.17487/RFC2853, June 2000, <https://www.rfc-editor.org/info/rfc2853>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <https://www.rfc-editor.org/info/rfc4121>.

[RFC4178] Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism", RFC 4178, DOI 10.17487/RFC4178, October 2005, <https://www.rfc-editor.org/info/rfc4178>.

[RFC5653] Upadhyay, M. and S. Malkani, "Generic Security Service API Version 2: Java Bindings Update", RFC 5653, DOI 10.17487/RFC5653, August 2009, <https://www.rfc-editor.org/info/rfc5653>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

14.2. Informative References

[JLS] Gosling, J., Joy, B., Steele, G., and G. Bracha, "The Java Language Specification", Third Edition, 2005, <http://java.sun.com/docs/books/jls/>.

Authors' Addresses

Mayank D. Upadhyay
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA

Email: m.d.upadhyay+ietf@gmail.com

Seema Malkani
ActivIdentity Corp.
6623 Dumbarton Circle
Fremont, California 94555
USA

Email: Seema.Malkani@gmail.com

Wang Weijun
Oracle
Building No. 24, Zhongguancun Software Park
Beijing 100193
China

Email: weijun.wang@oracle.com

Network Working Group
Internet-Draft
Obsoletes: 6112 (if approved)
Updates: 4120, 4121, 4556 (if approved)
Intended status: Standards Track
Expires: May 20, 2017

L. Zhu
P. Leach
Microsoft Corporation
S. Hartman
Painless Security
S. Emery, Ed.
Oracle

November 16, 2016

Anonymity Support for Kerberos
draft-ietf-kitten-rfc6112bis-03

Abstract

This document defines extensions to the Kerberos protocol to allow a Kerberos client to securely communicate with a Kerberos application service without revealing its identity, or without revealing more than its Kerberos realm. It also defines extensions that allow a Kerberos client to obtain anonymous credentials without revealing its identity to the Kerberos Key Distribution Center (KDC). This document updates RFCs 4120, 4121, and 4556. This document obsoletes RFC 6112 and reclassifies that document as historic. RFC 6112 contained errors and the protocol described in that specification is not interoperable with any known implementation. This specification describes a protocol that interoperates with multiple implementations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In certain situations, the Kerberos [RFC4120] client may wish to authenticate a server and/or protect communications without revealing the client's own identity. For example, consider an application that provides read access to a research database and that permits queries by arbitrary requesters. A client of such a service might wish to authenticate the service, to establish trust in the information received from it, but might not wish to disclose the client's identity to the service for privacy reasons.

Extensions to Kerberos are specified in this document by which a client can authenticate the Key Distribution Center (KDC) and request an anonymous ticket. The client can use the anonymous ticket to authenticate the server and protect subsequent client-server communications.

By using the extensions defined in this specification, the client can request an anonymous ticket where the client may reveal the client's identity to the client's own KDC, or the client can hide the client's identity completely by using anonymous Public Key Cryptography for Initial Authentication in Kerberos (PKINIT) as defined in Section 4.1. Using the returned anonymous ticket, the client remains anonymous in subsequent Kerberos exchanges thereafter to KDCs on the cross-realm authentication path and to the server with which it communicates.

In this specification, the client realm in the anonymous ticket is the anonymous realm name when anonymous PKINIT is used to obtain the ticket. The client realm is the client's real realm name if the client is authenticated using the client's long-term keys. Note that a membership in a realm can imply a member of the community represented by the realm.

The interaction with Generic Security Service Application Program Interface (GSS-API) is described after the protocol description.

This specification replaces [RFC6112] to correct technical errors in that specification. RFC 6112 is classified as historic; implementation of RFC 6112 is NOT RECOMMENDED: existing implementations comply with this specification and not RFC 6112.

1.1. Changes Since RFC 6112

In Section 7, the pepper2 string, "KeyExchange", is corrected to comply with the string actually used by implementations.

The requirement for the anonymous option to be used when an anonymous ticket is used in a TGS request is reduced from a MUST to a SHOULD. At least one implementation does not require this and is not necessary that both be used as an indicator of request type.

Corrected the authorization data type name, AD-INITIAL-VERIFIED-CAS, referenced in this document.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Definitions

The anonymous Kerberos realm name is defined as a well-known realm name based on [RFC6111], and the value of this well-known realm name is the literal "WELLKNOWN:ANONYMOUS".

The anonymous Kerberos principal name is defined as a well-known Kerberos principal name based on [RFC6111]. The value of the name-type field is KRB_NT_WELLKNOWN [RFC6111], and the value of the name-string field is a sequence of two KerberosString components: "WELLKNOWN", "ANONYMOUS".

The anonymous ticket flag is defined as bit 16 (with the first bit being bit 0) in the TicketFlags:

```
TicketFlags     ::= KerberosFlags
  -- anonymous(16)
  -- TicketFlags and KerberosFlags are defined in [RFC4120]
```

This is a new ticket flag that is used to indicate that a ticket is an anonymous one.

An anonymous ticket is a ticket that has all of the following properties:

- o The cname field contains the anonymous Kerberos principal name.

- o The crealm field contains the client's realm name or the anonymous realm name.

- o The anonymous ticket contains no information that can reveal the client's identity. However, the ticket may contain the client realm, intermediate realms on the client's authentication path, and authorization data that may provide information related to the client's identity. For example, an anonymous principal that is identifiable only as being in a particular group of users can be implemented using authorization data. Such authorization data, if included in the anonymous ticket, would disclose that the client is a member of the group observed.

- o The anonymous ticket flag is set.

The anonymous KDC option is defined as bit 16 (with the first bit being bit 0) in the KDCOptions:

```
KDCOptions      ::= KerberosFlags
  -- anonymous(16)
  -- KDCOptions and KerberosFlags are defined in [RFC4120]
```

As described in Section 4, the anonymous KDC option is set to request an anonymous ticket in an Authentication Service (AS) request or a Ticket Granting Service (TGS) request.

4. Protocol Description

In order to request an anonymous ticket, the client sets the anonymous KDC option in an AS request or a TGS request.

The rest of this section is organized as follows: it first describes protocol actions specific to AS exchanges, then it describes those of TGS exchanges. These are then followed by the description of protocol actions common to both AS and TGS and those in subsequent exchanges.

4.1. Anonymity Support in AS Exchange

The client requests an anonymous ticket by setting the anonymous KDC option in an AS exchange.

The Kerberos client can use the client's long-term keys, the client's X.509 certificates [RFC4556], or any other pre-authentication data,

to authenticate to the KDC and request an anonymous ticket in an AS exchange where the client's identity is known to the KDC.

If the client in the AS request is anonymous, the anonymous KDC option MUST be set in the request. Otherwise, the KDC MUST return a KRB-ERROR message with the code KDC_ERR_BADOPTION.

If the client is anonymous and the KDC does not have a key to encrypt the reply (this can happen when, for example, the KDC does not support PKINIT [RFC4556]), the KDC MUST return an error message with the code KDC_ERR_NULL_KEY [RFC4120].

When policy allows, the KDC issues an anonymous ticket. If the client name in the request is the anonymous principal, the client realm (crealm) in the reply is the anonymous realm, otherwise, the client realm is the realm of the AS. As specified by [RFC4120], the client name and the client realm in the EncTicketPart of the reply MUST match with the corresponding client name and the client realm of the KDC reply; the client MUST use the client name and the client realm returned in the KDC-REP in subsequent message exchanges when using the obtained anonymous ticket.

The KDC MUST NOT reveal the client's identity in the authorization data of the returned ticket when populating the authorization data in a returned anonymous ticket.

The AD_INITIAL_VERIFIED_CAS authorization data, as defined in [RFC4556], contains the issuer name of the client certificate. This authorization is not applicable and MUST NOT be present in the returned anonymous ticket when anonymous PKINIT is used. When the client is authenticated (i.e., anonymous PKINIT is not used), if it is undesirable to disclose such information about the client's identity, the AD_INITIAL_VERIFIED_CAS authorization data SHOULD be removed from the returned anonymous ticket.

The client can use the client's key to mutually authenticate with the KDC and request an anonymous Ticket Granting Ticket (TGT) in the AS request. In that case, the reply key is selected as normal, according to Section 3.1.3 of [RFC4120].

4.1.1. Anonymous PKINIT

This sub-section defines anonymous PKINIT.

As described earlier in this section, the client can request an anonymous ticket by authenticating to the KDC using the client's identity; alternatively, without revealing the client's identity to the KDC, the Kerberos client can request an anonymous ticket as

follows: the client sets the client name as the anonymous principal in the AS exchange and provides PA_PK_AS_REQ pre-authentication data [RFC4556] where the signerInfos field of the SignedData [RFC5652] of the PA_PK_AS_REQ is empty, and the certificates field is absent. Because the anonymous client does not have an associated asymmetric key pair, the client MUST choose the Diffie-Hellman key agreement method by filling in the Diffie-Hellman domain parameters in the clientPublicValue [RFC4556]. This use of the anonymous client name in conjunction with PKINIT is referred to as anonymous PKINIT. If anonymous PKINIT is used, the realm name in the returned anonymous ticket MUST be the anonymous realm.

Upon receiving the anonymous PKINIT request from the client, the KDC processes the request, according to Section 3.1.2 of [RFC4120]. The KDC skips the checks for the client's signature and the client's public key (such as the verification of the binding between the client's public key and the client name), but performs otherwise applicable checks, and proceeds as normal, according to [RFC4556]. For example, the AS MUST check if the client's Diffie-Hellman domain parameters are acceptable. The Diffie-Hellman key agreement method MUST be used and the reply key is derived according to Section 3.2.3.1 of [RFC4556]. If the clientPublicValue is not present in the request, the KDC MUST return a KRB-ERROR with the code KDC_ERR_PUBLIC_KEY_ENCRYPTION_NOT_SUPPORTED [RFC4556]. If all goes well, an anonymous ticket is generated, according to Section 3.1.3 of [RFC4120], and PA_PK_AS_REP [RFC4556] pre-authentication data is included in the KDC reply, according to [RFC4556]. If the KDC does not have an asymmetric key pair, it MAY reply anonymously or reject the authentication attempt. If the KDC replies anonymously, the signerInfos field of the SignedData [RFC5652] of PA_PK_AS_REP in the reply is empty, and the certificates field is absent. The server name in the anonymous KDC reply contains the name of the TGS.

Upon receipt of the KDC reply that contains an anonymous ticket and PA_PK_AS_REP [RFC4556] pre-authentication data, the client can then authenticate the KDC based on the KDC's signature in the PA_PK_AS_REP. If the KDC's signature is missing in the KDC reply (the reply is anonymous), the client MUST reject the returned ticket if it cannot authenticate the KDC otherwise.

A KDC that supports anonymous PKINIT MUST indicate the support of PKINIT, according to Section 3.4 of [RFC4556]. In addition, such a KDC MUST indicate support for anonymous PKINIT by including a padata element of padata-type PA_PKINIT_KX and empty padata-value when including PA-PK-AS-REQ in an error reply.

When included in a KDC error, PA_PKINIT_KX indicates support for anonymous PKINIT. As discussed in Section 7, when included in an AS-

REP, PA_PKINIT_KX proves that the KDC and client both contributed to the session key for any use of Diffie-Hellman key agreement with PKINIT.

Note that in order to obtain an anonymous ticket with the anonymous realm name, the client MUST set the client name as the anonymous principal in the request when requesting an anonymous ticket in an AS exchange. Anonymous PKINIT is the only way via which an anonymous ticket with the anonymous realm as the client realm can be generated in this specification.

4.2. Anonymity Support in TGS Exchange

The client requests an anonymous ticket by setting the anonymous KDC option in a TGS exchange, and in that request the client can use a normal Ticket Granting Ticket (TGT) with the client’s identity, or an anonymous TGT, or an anonymous cross-realm TGT. If the client uses a normal TGT, the client’s identity is known to the TGS.

Note that the client can completely hide the client’s identity in an AS exchange using anonymous PKINIT, as described in the previous section.

If the ticket in the PA-TGS-REQ of the TGS request is an anonymous one, the anonymous KDC option SHOULD be set in the request.

When policy allows, the KDC issues an anonymous ticket. If the ticket in the TGS request is an anonymous one, the client name and the client realm are copied from that ticket; otherwise, the ticket in the TGS request is a normal ticket, the returned anonymous ticket contains the client name as the anonymous principal and the client realm as the true realm of the client. In all cases, according to [RFC4120] the client name and the client realm in the EncTicketPart of the reply MUST match with the corresponding client name and the client realm of the anonymous ticket in the reply; the client MUST use the client name and the client realm returned in the KDC-REP in subsequent message exchanges when using the obtained anonymous ticket.

The TGS MUST NOT reveal the client’s identity in the authorization data of the returned ticket. When propagating authorization data in the ticket or in the enc-authorization-data field of the request, the TGS MUST ensure that the client confidentiality is not violated in the returned anonymous ticket. The TGS MUST process the authorization data recursively, according to Section 5.2.6 of [RFC4120], beyond the container levels such that all embedded authorization elements are interpreted. The TGS SHOULD NOT populate identity-based authorization data into an anonymous ticket in that

such authorization data typically reveals the client's identity. The specification of a new authorization data type MUST specify the processing rules of the authorization data when an anonymous ticket is returned. If there is no processing rule defined for an authorization data element or the authorization data element is unknown, the TGS MUST process it when an anonymous ticket is returned as follows:

- o If the authorization data element may reveal the client's identity, it MUST be removed unless otherwise specified.

- o If the authorization data element, that could reveal the client's identity, is intended to restrict the use of the ticket or limit the rights otherwise conveyed in the ticket, it cannot be removed in order to hide the client's identity. In this case, the authentication attempt MUST be rejected, and the TGS MUST return an error message with the code KDC_ERR_POLICY. Note this is applicable to both critical and optional authorization data.

- o If the authorization data element is unknown, the TGS MAY remove it, or transfer it into the returned anonymous ticket, or reject the authentication attempt, based on local policy for that authorization data type unless otherwise specified. If there is no policy defined for a given unknown authorization data type, the authentication MUST be rejected. The error code is KDC_ERR_POLICY when the authentication is rejected.

The AD_INITIAL_VERIFIED_CAS authorization data, as defined in [RFC4556], contains the issuer name of the client certificate. If it is undesirable to disclose such information about the client's identity, the AD_INITIAL_VERIFIED_CAS authorization data SHOULD be removed from an anonymous ticket.

The TGS encodes the name of the previous realm into the transited field, according to Section 3.3.3.2 of [RFC4120]. Based on local policy, the TGS MAY omit the previous realm, if the cross realm TGT is an anonymous one, in order to hide the authentication path of the client. The unordered set of realms in the transited field, if present, can reveal which realm may potentially be the realm of the client or the realm that issued the anonymous TGT. The anonymous Kerberos realm name MUST NOT be present in the transited field of a ticket. The true name of the realm that issued the anonymous ticket MAY be present in the transited field of a ticket.

4.3. Subsequent Exchanges and Protocol Actions Common to AS and TGS for Anonymity Support

In both AS and TGS exchanges, the realm field in the KDC request is always the realm of the target KDC, not the anonymous realm when the client requests an anonymous ticket.

Absent other information, the KDC MUST NOT include any identifier in the returned anonymous ticket that could reveal the client’s identity to the server.

Unless anonymous PKINIT is used, if a client requires anonymous communication, then the client MUST check to make sure that the ticket in the reply is actually anonymous by checking the presence of the anonymous ticket flag in the flags field of the EncKDCRepPart. This is because KDCs ignore unknown KDC options. A KDC that does not understand the anonymous KDC option will not return an error, but will instead return a normal ticket.

The subsequent client and server communications then proceed as described in [RFC4120].

Note that the anonymous principal name and realm are only applicable to the client in Kerberos messages, the server cannot be anonymous in any Kerberos message per this specification.

A server accepting an anonymous service ticket may assume that subsequent requests using the same ticket originate from the same client. Requests with different tickets are likely to originate from different clients.

Upon receipt of an anonymous ticket, the transited policy check is performed in the same way as that of a normal ticket if the client’s realm is not the anonymous realm; if the client realm is the anonymous realm, absent other information any realm in the authentication path is allowed by the cross-realm policy check.

5. Interoperability Requirements

Conforming implementations MUST support the anonymous principal with a non-anonymous realm, and they MAY support the anonymous principal with the anonymous realm using anonymous PKINIT.

6. GSS-API Implementation Notes

GSS-API defines the name_type GSS_C_NT_ANONYMOUS [RFC2743] to represent the anonymous identity. In addition, Section 2.1.1 of [RFC1964] defines the single string representation of a Kerberos

principal name with the name_type GSS_KRB5_NT_PRINCIPAL_NAME. The anonymous principal with the anonymous realm corresponds to the GSS-API anonymous principal. A principal with the anonymous principal name and a non-anonymous realm is an authenticated principal; hence, such a principal does not correspond to the anonymous principal in GSS-API with the GSS_C_NT_ANONYMOUS name type. The [RFC1964] name syntax for GSS_KRB5_NT_PRINCIPAL_NAME MUST be used for importing the anonymous principal name with a non-anonymous realm name and for displaying and exporting these names. In addition, this syntax must be used along with the name type GSS_C_NT_ANONYMOUS for displaying and exporting the anonymous principal with the anonymous realm.

At the GSS-API [RFC2743] level, an initiator/client requests the use of an anonymous principal with the anonymous realm by asserting the "anonymous" flag when calling GSS_Init_Sec_Context(). The GSS-API implementation MAY provide implementation-specific means for requesting the use of an anonymous principal with a non-anonymous realm.

GSS-API does not know or define "anonymous credentials", so the (printable) name of the anonymous principal will rarely be used by or relevant for the initiator/client. The printable name is relevant for the acceptor/server when performing an authorization decision based on the initiator name that is returned from the acceptor side upon the successful security context establishment.

A GSS-API initiator MUST carefully check the resulting context attributes from the initial call to GSS_Init_Sec_Context() when requesting anonymity, because (as in the GSS-API tradition and for backwards compatibility) anonymity is just another optional context attribute. It could be that the mechanism doesn’t recognize the attribute at all or that anonymity is not available for some other reasons -- and in that case the initiator MUST NOT send the initial security context token to the acceptor, because it will likely reveal the initiators identity to the acceptor, something that can rarely be "un-done".

Portable initiators are RECOMMENDED to use default credentials whenever possible, and request anonymity only through the input anon_req_flag [RFC2743] to GSS_Init_Sec_Context().

7. PKINIT Client Contribution to the Ticket Session Key

The definition in this section was motivated by protocol analysis of anonymous PKINIT (defined in this document) in building secure channels [RFC6113] and subsequent channel bindings [RFC5056]. In order to enable applications of anonymous PKINIT to form secure channels, all implementations of anonymous PKINIT need to meet the

requirements of this section. There is otherwise no connection to the rest of this document.

PKINIT is useful for constructing secure channels. To ensure that an active attacker cannot create separate channels to the client and KDC with the same known key, it is desirable that neither the KDC nor the client unilaterally determine the ticket session key. The specific reason why the ticket session key is derived jointly is discussed at the end of this section. To achieve that end, a KDC conforming to this definition MUST encrypt a randomly generated key, called the KDC contribution key, in the PA_PKINIT_KX padata (defined next in this section). The KDC contribution key is then combined with the reply key to form the ticket session key of the returned ticket. These two keys are then combined using the KRB-FX-CF2 operation defined in Section 7.1, where K1 is the KDC contribution key, K2 is the reply key, the input pepper1 is American Standard Code for Information Interchange (ASCII) [ASAX34] string "PKINIT", and the input pepper2 is ASCII string "KEYEXCHANGE".

```
PA_PKINIT_KX      147
  -- padata for PKINIT that contains an encrypted
  -- KDC contribution key.

PA-PKINIT-KX  ::= EncryptedData -- EncryptionKey
  -- Contains an encrypted key randomly
  -- generated by the KDC (known as the KDC contribution key).
  -- Both EncryptedData and EncryptionKey are defined in [RFC4120]
```

The PA_PKINIT_KX padata MUST be included in the KDC reply when anonymous PKINIT is used; it SHOULD be included if PKINIT is used with the Diffie-Hellman key exchange but the client is not anonymous; it MUST NOT be included otherwise (e.g., when PKINIT is used with the public key encryption as the key exchange).

The padata-value field of the PA-PKINIT-KX type padata contains the DER [X.680] [X.690] encoding of the Abstract Syntax Notation One (ASN.1) type PA-PKINIT-KX. The PA-PKINIT-KX structure is an EncryptedData. The cleartext data being encrypted is the DER-encoded KDC contribution key randomly generated by the KDC. The encryption key is the reply key and the key usage number is KEY_USAGE_PA_PKINIT_KX (44).

The client then decrypts the KDC contribution key and verifies the ticket session key in the returned ticket is the combined key of the KDC contribution key and the reply key as described above. A conforming client MUST reject anonymous PKINIT authentication if the PA_PKINIT_KX padata is not present in the KDC reply or if the ticket session key of the returned ticket is not the combined key of the KDC

contribution key and the reply key when PA-PKINIT-KX is present in the KDC reply.

This protocol provides a binding between the party which generated the session key and the DH exchange used to generate the reply key. Hypothetically, if the KDC did not use PA-PKINIT-KX, the client and KDC would perform a DH key exchange to determine a shared key, and that key would be used as a reply key. The KDC would then generate a ticket with a session key encrypting the reply with the DH agreement. A MITM attacker would just decrypt the session key and ticket using the DH key from the attacker-KDC DH exchange, and re-encrypt it using the key from the attacker-client DH exchange, while keeping a copy of the session key and ticket. This protocol binds the ticket to the DH exchange and prevents the MITM attack by requiring the session key to be created in a way that can be verified by the client.

7.1. Combining Two Protocol Keys

KRB-FX-CF2() combines two protocol keys based on the pseudo-random() function defined in [RFC3961].

Given two input keys, K1 and K2, where K1 and K2 can be of two different enctypes, the output key of KRB-FX-CF2(), K3, is derived as follows:

```
KRB-FX-CF2(protocol key, protocol key, octet string,
          octet string)  ->  (protocol key)

PRF+(K1, pepper1) -> octet-string-1
PRF+(K2, pepper2) -> octet-string-2
KRB-FX-CF2(K1, K2, pepper1, pepper2) ->
       random-to-key(octet-string-1 ^ octet-string-2)
```

Where ^ denotes the exclusive-OR operation. PRF+() is defined as follows:

```
PRF+(protocol key, octet string) -> (octet string)

PRF+(key, shared-info) -> pseudo-random( key,  1 || shared-info ) ||
             pseudo-random( key, 2 || shared-info ) ||
             pseudo-random( key, 3 || shared-info ) || ...
```

Here the counter value 1, 2, 3, and so on are encoded as a one-octet integer. The pseudo-random() operation is specified by the enctype of the protocol key. PRF+() uses the counter to generate enough bits as needed by the random-to-key() [RFC3961] function for the encryption type specified for the resulting key; unneeded bits are removed from the tail.

8. Security Considerations

Since KDCs ignore unknown options, a client requiring anonymous communication needs to make sure that the returned ticket is actually anonymous. This is because a KDC that does not understand the anonymous option would not return an anonymous ticket.

By using the mechanism defined in this specification, the client does not reveal the client's identity to the server but the client identity may be revealed to the KDC of the server principal (when the server principal is in a different realm than that of the client), and any KDC on the cross-realm authentication path. The Kerberos client MUST verify the ticket being used is indeed anonymous before communicating with the server, otherwise, the client's identity may be revealed unintentionally.

In cases where specific server principals must not have access to the client's identity (for example, an anonymous poll service), the KDC can define server-principal-specific policy that ensures any normal service ticket can NEVER be issued to any of these server principals.

If the KDC that issued an anonymous ticket were to maintain records of the association of identities to an anonymous ticket, then someone obtaining such records could breach the anonymity. Additionally, the implementations of most (for now all) KDC's respond to requests at the time that they are received. Traffic analysis on the connection to the KDC will allow an attacker to match client identities to anonymous tickets issued. Because there are plaintext parts of the tickets that are exposed on the wire, such matching by a third-party observer is relatively straightforward. A service that is authenticated by the anonymous principals may be able to infer the identity of the client by examining and linking quasi-static protocol information such as the IP address from which a request is received, or by linking multiple uses of the same anonymous ticket.

Two mechanisms, the FAST facility with the hide-client-names option in [RFC6113] and the Kerberos5 starttls option [STARTTLS], protect the client identity so that an attacker would never be able to observe the client identity sent to the KDC. Transport or network layer security between the client and the server will help prevent tracking of a particular ticket to link a ticket to a user. In addition, clients can limit how often a ticket is reused to minimize ticket linking.

The client's real identity is not revealed when the client is authenticated as the anonymous principal. Application servers MAY reject the authentication in order to, for example, prevent information disclosure or as part of Denial of Service (DoS)

prevention. Application servers MUST avoid accepting anonymous credentials in situations where they must record the client's identity; for example, when there must be an audit trail.

9. Acknowledgments

JK Jaganathan helped editing early revisions of this document.

Clifford Neuman contributed the core notions of this document.

Ken Raeburn reviewed the document and provided suggestions for improvements.

Martin Rex wrote the text for GSS-API considerations.

Nicolas Williams reviewed the GSS-API considerations section and suggested ideas for improvements.

Sam Hartman and Nicolas Williams were great champions of this work.

Miguel Garcia and Phillip Hallam-Baker reviewed the document and provided helpful suggestions.

In addition, the following individuals made significant contributions: Jeffrey Altman, Tom Yu, Chaskiel M Grundman, Love Hornquist Astrand, Jeffrey Hutzelman, and Olga Kornievskaia.

Greg Hudson and Robert Sparks had provided helpful text in the bis version of the draft.

10. IANA Considerations

This document defines an 'anonymous' Kerberos well-known name and an 'anonymous' Kerberos well-known realm based on [RFC6111]. IANA has added these two values to the Kerberos naming registries that are created in [RFC6111].

Note to IANA: Please update the following Kerberos Parameters registries:

- o Well-Known Kerberos Principal Names
- o Well-Known Kerberos Realm Names
- o Pre-authentication and Typed Data

to reference this document instead of RFC6112.

11. References

11.1. Normative References

[ASAX34] American Standards Institute, "American Standard Code for Information Interchange", ASA X3.4-1963, June 1963.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, DOI 10.17487/RFC1964, June 1996, <http://www.rfc-editor.org/info/rfc1964>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <http://www.rfc-editor.org/info/rfc2743>.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, DOI 10.17487/RFC3961, February 2005, <http://www.rfc-editor.org/info/rfc3961>.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC4556] Zhu, L. and B. Tung, "Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 4556, DOI 10.17487/RFC4556, June 2006, <http://www.rfc-editor.org/info/rfc4556>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[RFC6111] Zhu, L., "Additional Kerberos Naming Constraints", RFC 6111, April 2011.

[RFC6112] Zhu, L., Leach, P., and S. Hartman, "Anonymity Support for Kerberos", RFC 6112, April 2011.

[X.680] "Abstract Syntax Notation One (ASN.1): Specification of Basic Notation", ITU-T Recommendation X.680: ISO/IEC International Standard 8824-1:1998, 1997.

[X.690] "ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690 ISO/IEC International Standard 8825-1:1998, 1997.

11.2. Informative References

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC6113] Hartman, S. and L. Zhu, "A Generalized Framework for Kerberos Pre-Authentication", RFC 6113, April 2011.

[STARTTLS] Josefsson, S., "Using Kerberos V5 over the Transport Layer Security (TLS) protocol", Work in Progress, August 2010.

Authors' Addresses

Larry Zhu
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

EMail: larry.zhu@microsoft.com

Paul Leach
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
US

EMail: paulle@microsoft.com

Sam Hartman
Painless Security

EMail: hartmans-ietf@mit.edu

Shawn Emery (editor)
Oracle

EMail: shawn.emery@oracle.com

KITTEN
Internet-Draft
Intended status: Standards Track
Expires: November 30, 2015

W. Mills
Microsoft
T. Showalter

H. Tschofenig
ARM Ltd.
May 29, 2015

A set of SASL Mechanisms for OAuth
draft-ietf-kitten-sasl-oauth-23.txt

Abstract

OAuth enables a third-party application to obtain limited access to a protected resource, either on behalf of a resource owner by orchestrating an approval interaction, or by allowing the third-party application to obtain access on its own behalf.

This document defines how an application client uses credentials obtained via OAuth over the Simple Authentication and Security Layer (SASL) to access a protected resource at a resource serve. Thereby, it enables schemes defined within the OAuth framework for non-HTTP-based application protocols.

Clients typically store the user’s long-term credential. This does, however, lead to significant security vulnerabilities, for example, when such a credential leaks. A significant benefit of OAuth for usage in those clients is that the password is replaced by a shared secret with higher entropy, i.e., the token. Tokens typically provide limited access rights and can be managed and revoked separately from the user’s long-term password.

Status of This Memo

This Internet-Draft will expire on November 30, 2015.

Copyright Notice

Table of Contents

1. Introduction

OAuth 1.0a [RFC5849] and OAuth 2.0 [RFC6749] are protocol frameworks that enable a third-party application to obtain limited access to a protected resource, either on behalf of a resource owner by orchestrating an approval interaction, or by allowing the third-party application to obtain access on its own behalf.

The core OAuth 2.0 specification [RFC6749] specifies the interaction between the OAuth client and the authorization server; it does not define the interaction between the OAuth client and the resource server for the access to a protected resource using an Access Token. Instead, the OAuth client to resource server interaction is described in separate specifications, such as the bearer token specification [RFC6750]. OAuth 1.0a included the protocol specification for the communication between the OAuth client and the resource server in [RFC5849].

The main use cases for OAuth 2.0 and OAuth 1.0a have so far focused on an HTTP-based [RFC7230] environment only. This document integrates OAuth 1.0a and OAuth 2.0 into non-HTTP-based applications using the integration into SASL. Hence, this document takes advantage of the OAuth protocol and its deployment base to provide a way to use the Simple Authentication and Security Layer (SASL) [RFC4422] to gain access to resources when using non-HTTP-based protocols, such as the Internet Message Access Protocol (IMAP) [RFC3501] and the Simple Mail Transfer Protocol (SMTP) [RFC5321]. This document gives examples of use in IMAP and SMTP.

To illustrate the impact of integrating this specification into an OAuth-enabled application environment, Figure 1 shows the abstract message flow of OAuth 2.0 [RFC6749]. As indicated in the figure, this document impacts the exchange of messages (E) and (F) since SASL is used for interaction between the client and the resource server instead of HTTP.

```
                                                          ----+
   +--------+                                 +---------------+  |
   |        |--(A)-- Authorization Request --->|   Resource    |  |
   |        |                                 |     Owner     |  |Plain
   |        |<-(B)------ Access Grant ---------|               |  |OAuth
   |        |                                 +---------------+  |2.0
   |        |                                                    |
   |        |         Client Credentials &    +---------------+  |
   |        |--(C)------ Access Grant -------->| Authorization |  |
   | Client |                                 |     Server    |  |
   |        |<-(D)------ Access Token ---------|               |  |
   |        |      (w/ Optional Refresh Token) +---------------+  |
   |        |                                                  ----+
   |        |                                                  ----+
   |        |                                 +---------------+  |
   |        |                                 |               |  |OAuth
   |        |--(E)------ Access Token -------->|    Resource   |  |over
   |        |                                 |     Server    |  |SASL
   |        |<-(F)---- Protected Resource -----|               |  |
   |        |                                 |               |  |
   +--------+                                 +---------------+  |
                                                          ----+
```

Figure 1: OAuth 2.0 Protocol Flow

The Simple Authentication and Security Layer (SASL) is a framework for providing authentication and data security services in connection-oriented protocols via replaceable authentication mechanisms. It provides a structured interface between protocols and mechanisms. The resulting framework allows new protocols to reuse existing authentication mechanisms and allows old protocols to make use of new authentication mechanisms. The framework also provides a protocol for securing subsequent exchanges within a data security layer.

When OAuth is integrated into SASL the high-level steps are as follows:

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or indirectly via the authorization server as an intermediary.

(B) The client receives an authorization grant which is a credential representing the resource owner’s authorization, expressed using one of the grant types defined in [RFC6749] or [RFC5849] or using an extension grant type. The authorization

grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

(C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates the authorization grant, and if valid issues an access token.

(E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F) The resource server validates the access token, and if valid, indicates a successful authentication.

Again, steps (E) and (F) are not defined in [RFC6749] (but are described in, for example, [RFC6750] for the OAuth Bearer Token instead) and are the main functionality specified within this document. Consequently, the message exchange shown in Figure 1 is the result of this specification. The client will generally need to determine the authentication endpoints (and perhaps the service endpoints) before the OAuth 2.0 protocol exchange messages in steps (A)-(D) are executed. The discovery of the resource owner, authorization server endpoints, and client registration are outside the scope of this specification. The client must discover the authorization endpoints using a discovery mechanism such as OpenID Connect Discovery [OpenID.Discovery] or Webfinger using host-meta [RFC7033]. Once credentials are obtained the client proceeds to steps (E) and (F) defined in this specification. Authorization endpoints MAY require client registration and generic clients SHOULD support the Dynamic Client Registration protocol [I-D.ietf-oauth-dyn-reg].

OAuth 1.0 follows a similar model but uses a different terminology and does not separate the resource server from the authorization server.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The reader is assumed to be familiar with the terms used in the OAuth 2.0 specification [RFC6749] and SASL [RFC4422].

In examples, "C:" and "S:" indicate lines sent by the client and server respectively. Line breaks have been inserted for readability.

Note that the IMAP SASL specification requires base64 encoding, as specified in Section 4 of [RFC4648].

3. OAuth SASL Mechanism Specifications

SASL is used as an authentication framework in a variety of application layer protocols. This document defines the following SASL mechanisms for usage with OAuth:

OAUTHBEARER: OAuth 2.0 bearer tokens, as described in [RFC6750]. RFC 6750 uses Transport Layer Security (TLS) [RFC5246] to secure the protocol interaction between the client and the resource server.

OAUTH10A: OAuth 1.0a MAC tokens (using the HMAC-SHA1 keyed message digest), as described in Section 3.4.2 of [RFC5849].

New extensions may be defined to add additional OAuth Access Token Types. Such a new SASL OAuth mechanism can be added by registering the new name(s) with IANA in the SASL Mechanisms registry and citing this specification for the further definition.

SASL mechanisms using this document as their definition do not provide a data security layer; that is, they cannot provide integrity or confidentiality protection for application messages after the initial authentication. If such protection is needed, TLS or some similar solution should be used. Additionally, for the two mechanisms specified in this document, TLS MUST be used for OAUTHBEARER to protect the bearer token; for OAUTH10A the use of TLS is RECOMMENDED.

These mechanisms are client initiated and lock-step, the server always replying to a client message. In the case where the client has and correctly uses a valid token the flow is:

1. Client sends a valid and correct initial client response.

2. Server responds with a successful authentication.

In the case where authentication fails the server sends an error result, then client MUST then send an additional message to the server in order to allow the server to finish the exchange. Some protocols and common SASL implementations do not support both sending

a SASL message and finalizing a SASL negotiation. The additional client message in the error case deals with this problem. This exchange is:

1. Client sends an invalid initial client response.

2. Server responds with an error message.

3. Client sends a dummy client response.

4. Server fails the authentication.

3.1. Initial Client Response

Client responses are a GS2 [RFC5801] header followed by zero or more key/value pairs, or may be empty. The gs2-header is defined here for compatibility with GS2 if a GS2 mechanism is formally defined, but this document does not define one. The key/value pairs take the place of the corresponding HTTP headers and values to convey the information necessary to complete an OAuth style HTTP authorization. Unknown key/value pairs MUST be ignored by the server. The ABNF [RFC5234] syntax is:

```
  kvsep          = %x01
  key            = 1*(ALPHA)
  value          = *(VCHAR / SP / HTAB / CR / LF )
  kvpair         = key "=" value kvsep
;;gs2-header     = See RFC 5801
  client_resp    = (gs2-header kvsep *kvpair kvsep) / kvsep
```

The GS2 header MAY include the user name associated with the resource being accessed, the "authzid". It is worth noting that application protocols are allowed to require an authzid, as are specific server implementations.

The client response consisting of only a single kvsep is used only when authentication fails, and is only valid in that context. If sent as the first message from the client the server MAY simply fail the authentication without returning discovery information since there is no user or server name indication.

The following keys and corresponding values are defined in the client response:

auth (REQUIRED): The payload that would be in the HTTP Authorization header if this OAuth exchange was being carried out over HTTP.

host: Contains the host name to which the client connected. In an HTTP context this is the value of the HTTP Host header.

port: Contains the destination port that the client connected to, represented as a decimal positive integer string without leading zeros.

For OAuth token types such as OAuth 1.0a that use keyed message digests the client MUST send host and port number key/values, and the server MUST fail an authorization request requiring keyed message digests that are not accompanied by host and port values. In OAuth 1.0a for example, the so-called "signature base string calculation" includes the reconstructed HTTP URL.

3.1.1. Reserved Key/Values

In these mechanisms values for path, query string and post body are assigned default values. OAuth authorization schemes MAY define usage of these in the SASL context and extend this specification. For OAuth Access Token Types that include a keyed message digest of the request the default values MUST be used unless explicit values are provided in the client response. The following key values are reserved for future use:

mthd (RESERVED): HTTP method, the default value is "POST".

path (RESERVED): HTTP path data, the default value is "/".

post (RESERVED): HTTP post data, the default value is the empty string ("").

qs (RESERVED): The HTTP query string, the default value is the empty string ("").

3.2. Server’s Response

The server validates the response according to the specification for the OAuth Access Token Types used. If the OAuth Access Token Type utilizes a keyed message digest of the request parameters then the client must provide a client response that satisfies the data requirements for the scheme in use.

The server fully validates the client response before generating a server response; this will necessarily include the validation steps listed in the specification for the OAuth Access Token Type used. However, additional validation steps may be needed, depending on the particular application protocol making use of SASL. In particular, values included as kvpairs in the client response (such as host and port) which correspond to values known to the application server by some other mechanism (such as an application protocol data unit or pre-configured values) MUST be validated to match between the initial client response and the the other source(s) of such information. As a concrete example, when SASL is used over IMAP to an IMAP server for a single domain the hostname can be available via configuration; this hostname must be validated to match the value sent in the 'host' kvpair.

The server responds to a successfully verified client message by completing the SASL negotiation. The authenticated identity reported by the SASL mechanism is the identity securely established for the client with the OAuth credential. The application, not the SASL mechanism, based on local access policy determines whether the identity reported by the mechanism is allowed access to the requested resource. Note that the semantics of the authzid is specified by the SASL framework [RFC4422].

3.2.1. OAuth Identifiers in the SASL Context

In the OAuth framework the client may be authenticated by the authorization server and the resource owner is authenticated to the authorization server. OAuth access tokens may contain information about the authentication of the resource owner and about the client and may therefore make this information accessible to the resource server.

If both identifiers are needed by an application the developer will need to provide a way to communicate that from the SASL mechanism back to the application.

3.2.2. Server Response to Failed Authentication

For a failed authentication the server returns a JSON [RFC7159] formatted error result, and fails the authentication. The error result consists of the following values:

status (REQUIRED): The authorization error code. Valid error codes are defined in the IANA "OAuth Extensions Error Registry" specified in the OAuth 2 core specification.

scope (OPTIONAL): An OAuth scope which is valid to access the service. This may be omitted which implies that unscoped tokens are required. If a scope is specified then a single scope is preferred. At the time this document was written there are several implementations that do not properly support space separated lists of scopes, so the use of a space separated list of scopes is NOT RECOMMENDED.

openid-configuration (OPTIONAL): The URL for a document following the OpenID Provider Configuration Information schema as described in OpenID Connect Discovery (OIDCD) [OpenID.Discovery] section 3 that is appropriate for the user. As specified in OIDCD this will have the "https" URL scheme. This document MUST have all OAuth related data elements populated. The server MAY return different URLs for users in different domains and the client SHOULD NOT cache a single returned value and assume it applies for all users/domains that the server suports. The returned discovery document SHOULD have all data elements required by the OpenID Connect Discovery specification populated. In addition, the discovery document SHOULD contain the 'registration_endpoint' element to identify the endpoint to be used with the Dynamic Client Registration protocol [I-D.ietf-oauth-dyn-reg] to obtain the minimum number of parameters necessary for the OAuth protocol exchange to function. Another comparable discovery or client registration mechanism MAY be used if available.

The use of the 'offline_access' scope, as defined in [OpenID.Core] is RECOMMENDED to give clients the capability to explicitly request a refresh token.

If the resource server provides a scope then the client MUST always request scoped tokens from the token endpoint. If the resource server does not return a scope the client SHOULD presume an unscoped token is required to access the resource.

Since clients may interact with a number of application servers, such as email servers and XMPP [RFC6120] servers, they need to have a way to determine whether dynamic client registration has been performed already and whether an already available refresh token can be re-used to obtain an access token for the desired resource server. This specification RECOMMENDs that a client uses the information in the 'iss' element defined in OpenID Connect Core [OpenID.Core] to make this determination.

3.2.3. Completing an Error Message Sequence

Section 3.6 of SASL [RFC4422] explicitly prohibits additional information in an unsuccessful authentication outcome. Therefore, the error message is sent in a normal message. The client MUST then send either an additional client response consisting of a single %x01 (control A) character to the server in order to allow the server to finish the exchange or send a SASL cancellation token as generally defined in section 3.5 of SASL [RFC4422]. A specific example of a cancellation token can be found in IMAP [RFC3501] section 6.2.2.

3.3. OAuth Access Token Types using Keyed Message Digests

OAuth Access Token Types may use keyed message digests and the client and the resource server may need to perform a cryptographic computation for integrity protection and data origin authentication.

OAuth is designed for access to resources identified by URIs. SASL is designed for user authentication, and has no facility for more fine-grained access control. In this specification we require or define default values for the data elements from an HTTP request which allow the signature base string to be constructed properly. The default HTTP path is "/" and the default post body is empty. These atoms are defined as extension points so that no changes are needed if there is a revision of SASL which supports more specific resource authorization, e.g., IMAP access to a specific folder or FTP access limited to a specific directory.

Using the example in the OAuth 1.0a specification as a starting point, on an IMAP server running on port 143 and given the OAuth 1.0a style authorization request (with %x01 shown as ^A and line breaks added for readability) below:

```
n,a=user@example.com,^A
host=example.com^A
port=143^A
auth=OAuth realm="Example",
           oauth_consumer_key="9djdj82h48djs9d2",
           oauth_token="kkk9d7dh3k39sjv7",
           oauth_signature_method="HMAC-SHA1",
           oauth_timestamp="137131201",
           oauth_nonce="7d8f3e4a",
           oauth_signature="Tm90IGEgcmVhbCBzaWduYXR1cmU"^A^A
```

The signature base string would be constructed per the OAuth 1.0 specification [RFC5849] with the following things noted:

o The method value is defaulted to POST.

- o The scheme defaults to be "http", and any port number other than 80 is included.

- o The path defaults to "/".

- o The query string defaults to "".

In this example the signature base string with line breaks added for readability would be:

```
POST&http%3A%2F%2Fexample.com:143%2F&oauth_consumer_key%3D9djdj82h4
8djs9d2%26oauth_nonce%3D7d8f3e4a%26oauth_signature_method%3DHMAC-SH
A1%26oauth_timestamp%3D137131201%26oauth_token%3Dkkk9d7dh3k39sjv7
```

4. Examples

These examples illustrate exchanges between IMAP and SMTP clients and servers. All IMAP examples use SASL-IR [RFC4959] and send payload in the initial client response. The Bearer Token examples assume encrypted transport; if the underlying connection is not already TLS then STARTTLS MUST be used as TLS is required in the Bearer Token specification.

Note to implementers: The SASL OAuth method names are case insensitive. One example uses "Bearer" but that could as easily be "bearer", "BEARER", or "BeArEr".

4.1. Successful Bearer Token Exchange

This example shows a successful OAuth 2.0 bearer token exchange in IMAP. Note that line breaks are inserted for readability.

```
[Initial connection and TLS establishment...]
S: * OK IMAP4rev1 Server Ready
C: t0 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=OAUTHBEARER SASL-IR
S: t0 OK Completed
C: t1 AUTH OAUTHBEARER bixhPXVzZXJAZXhhbXBsZS5jb20sAWhvc3Q9c2Vy
      dmVyLmV4YW1wbGUuY29tAXBvcnQ9MTQzAWF1dGg9QmVhcmVyIHZGOWRmd
      DRxbVRjMk52YjNSbGNrQmhiSFJoZG1semRHRXVZMjl0Q2c9PQEB
S: t1 OK SASL authentication succeeded
```

As required by IMAP [RFC3501], the payloads are base64-encoded. The decoded initial client response (with %x01 represented as ^A and long lines wrapped for readability) is:

```
n,a=user@example.com,^Ahost=server.example.com^Aport=143^A
auth=Bearer vF9dft4qmTc2Nvb3RlckBhbHRhdmlzdGEuY29tCg==^A^A
```

The same credential used in an SMTP exchange is shown below. Again this example assumes that TLS is already established per the Bearer Token specification requirements.

```
[connection begins]
S: 220 mx.example.com ESMTP 12sm2095603fks.9
C: EHLO sender.example.com
S: 250-mx.example.com at your service,[172.31.135.47]
S: 250-SIZE 35651584
S: 250-8BITMIME
S: 250-AUTH LOGIN PLAIN OAUTHBEARER
S: 250-ENHANCEDSTATUSCODES
S: 250-STARTTLS
S: 250 PIPELINING
[Negotiate TLS...]
C: t1 AUTH OAUTHBEARER bixhPXVzZXJAZXhhbXBsZS5jb20sAWhvc3Q9c2Vy
      dmVyLmV4YW1wbGUuY29tAXBvcnQ9NTg3AWF1dGg9QmVhcmVyIHZGOWRmd
      DRxbVRjMk52YjNSbGNrQmhiSFJoZG1semRHRXVZMjl0Q2c9PQEB
S: 235 Authentication successful.
[connection continues...]
```

The decoded initial client response is:

```
n,a=user@example.com,^Ahost=server.example.com^Aport=587^A
auth=Bearer vF9dft4qmTc2Nvb3RlckBhbHRhdmlzdGEuY29tCg==^A^A
```

4.2. Successful OAuth 1.0a Token Exchange

This IMAP example shows a successful OAuth 1.0a token exchange. Note that line breaks are inserted for readability. This example assumes that TLS is already established. Signature computation is discussed in Section 3.3.

```
S: * OK IMAP4rev1 Server Ready
C: t0 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=OAUTHBEARER OAUTH10A SASL-IR
S: t0 OK Completed
C: t1 AUTH OAUTH10A bixhPXVzZXJAZXhhbXBsZS5jb20sAWhvc3Q9ZXhhb
      XBsZS5jb20BcG9ydD0xNDMBYXV0aD1PQXV0aCByZWFsbT0iRXhhbXBsZSIsb2F1
      dGhfY29uc3VtZXJfa2V5PSI5ZGpkajgyaDQ4ZGpzOWQyIixvYXV0aF90b2tlbj0
      ia2trOWQ3ZGgzazM5c2p2NyIsb2F1dGhfc2lnbmF0dXJlX21ldGhvZD0iSE1BQy
      1TSEExIixvYXV0aF90aW1lc3RhbXA9IjEzNzEzMTIwMSIsb2F1dGhfbm9uY2U9I
      jdkOGYzZTRhIixvYXV0aF9zaWduYXR1cmU9IlRtOTBJR0VnY21WaGJDQnphV2R1
      WVhSMWNtVSUzRCIBAQ==
S: t1 OK SASL authentication succeeded
```

As required by IMAP [RFC3501], the payloads are base64-encoded. The decoded initial client response (with %x01 represented as ^A and lines wrapped for readability) is:

```
n,a=user@example.com,^A
host=example.com^A
port=143^A
auth=OAuth realm="Example",
           oauth_consumer_key="9djdj82h48djs9d2",
           oauth_token="kkk9d7dh3k39sjv7",
           oauth_signature_method="HMAC-SHA1",
           oauth_timestamp="137131201",
           oauth_nonce="7d8f3e4a",
           oauth_signature="SSdtIGEgbGl0dGxlIHRlYSBwb3Qu"^A^A
```

4.3. Failed Exchange

This IMAP example shows a failed exchange because of the empty Authorization header, which is how a client can query for the needed scope. Note that line breaks are inserted for readability.

```
S: * OK IMAP4rev1 Server Ready
C: t0 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=OAUTHBEARER SASL-IR
S: t0 OK Completed
C: t1 AUTH OAUTHBEARER bixhPXVzZXJAZXhhbXBsZS5jb20sAW
      hvc3Q9c2VydmVyLmV4YW1wbGUuY29tAXBvcnQ9MTQzAWF1dGg9AQE=
S: + eyJzdGF0dXMiOiJpbnZhbGlkX3Rva2VuIiwic2NvcGUiOiJleGFtcGxl
     X3Njb3BlIiwib3BlbmlkLWNvbmZpZ3VyYXRpb24iOiJodHRwczovL2V4
     YW1wbGUuY29tLy53ZWxsLWtub3duL29wZW5pZC1jb25maWcifQ==
C: AQ==
S: t1 NO SASL authentication failed
```

The decoded initial client response is:

```
n,a=user@example.com,^Ahost=server.example.com^A
port=143^Aauth=^A^A
```

The decoded server error response is:

```
{
"status":"invalid_token",
"scope":"example_scope",
"openid-configuration":"https://example.com/.well-known/openid-config"
}
```

The client responds with the required dummy response, "AQ==" is the base64 encoding of the ASCII value 0x01. The same exchange using the

IMAP specific method of cancelling an AUTHENTICATE command sends "*" and is shown below.

```
S: * OK IMAP4rev1 Server Ready
C: t0 CAPABILITY
S: * CAPABILITY IMAP4rev1 AUTH=OAUTHBEARER SASL-IR IMAP4rev1
S: t0 OK Completed
C: t1 AUTH OAUTHBEARER bixhPXVzZXJAZXhhbXBsZS5jb20sAW
      hvc3Q9c2VydmVyLmV4YW1wbGUuY29tAXBvcnQ9MTQzAWF1dGg9AQE=
S: + eyJzdGF0dXMiOiJpbnZhbGlkX3Rva2VuIiwic2NvcGUiOiJleGFtcGxl
     X3Njb3BlIiwib3BlbmlkLWNvbmZpZ3VyYXRpb24iOiJodHRwczovL2V4
     YW1wbGUuY29tLy53ZWxsLWtub3duL29wZW5pZC1jb25maWd1cmF0aW9u
     In0=
C: *
S: t1 NO SASL authentication failed
```

4.4. SMTP Example of a Failed Negotiation

This example shows an authorization failure in an SMTP exchange. TLS negotiation is not shown but as noted above it is required for the use of Bearer Tokens.

```
[connection begins]
S: 220 mx.example.com ESMTP 12sm2095603fks.9
C: EHLO sender.example.com
S: 250-mx.example.com at your service,[172.31.135.47]
S: 250-SIZE 35651584
S: 250-8BITMIME
S: 250-AUTH LOGIN PLAIN OAUTHBEARER
S: 250-ENHANCEDSTATUSCODES
S: 250 PIPELINING
[Negotiate TLS...]
C: AUTH OAUTHBEARER bix1c2VyPXNvbWV1c2VyQGV4YW1wbGUuY29tLAFhdXRoPUJlYXJl
       ciB2RjlkZnQ0cW1UYzJOdmIzUmxja0JoZEhSaGRtbHpkR0V1WTI5dENnPT0BAQ==
S: 334 eyJzdGF0dXMiOiJpbnZhbGlkX3Rva2VuIiwic2NoZW1lcyI6ImJlYXJlciBtYWMiL
       CJzY29wZSI6Imh0dHBzOi8vbWFpbC5leGFtcGxlLmNvbS8ifQ==
C: AQ==
S: 535-5.7.1 Username and Password not accepted. Learn more at
S: 535 5.7.1 http://support.example.com/mail/oauth
[connection continues...]
```

The initial client response is:

```
n,user=someuser@example.com,^A
auth=Bearer vF9dft4qmTc2Nvb3RlckBhdHRhdmlzdGEuY29tCg==^A^A
```

The server returned an error message in the 334 SASL message, the client responds with the required dummy response, and the server finalizes the negotiation.

```
{
    "status":"invalid_token",
    "schemes":"bearer mac",
    "scope":"https://mail.example.com/"
}
```

5. Security Considerations

OAuth 1.0a and OAuth 2 allow for a variety of deployment scenarios, and the security properties of these profiles vary. As shown in Figure 1 this specification is aimed to be integrated into a larger OAuth deployment. Application developers therefore need to understand their security requirements based on a threat assessment before selecting a specific SASL OAuth mechanism. For OAuth 2.0 a detailed security document [RFC6819] provides guidance to select those OAuth 2.0 components that help to mitigate threats for a given deployment. For OAuth 1.0a Section 4 of RFC 5849 [RFC5849] provides guidance specific to OAuth 1.0.

This document specifies two SASL Mechanisms for OAuth and each comes with different security properties.

OAUTHBEARER: This mechanism borrows from OAuth 2.0 bearer tokens [RFC6750]. It relies on the application using TLS to protect the OAuth 2.0 Bearer Token exchange; without TLS usage at the application layer this method is completely insecure. Consequently, TLS MUST be provided by the application when choosing this authentication mechanism.

OAUTH10A: This mechanism re-uses OAuth 1.0a MAC tokens (using the HMAC-SHA1 keyed message digest), as described in Section 3.4.2 of [RFC5849]. To compute the keyed message digest in the same way as in RFC 5839 this specification conveys additional parameters between the client and the server. This SASL mechanism only supports client authentication. If server-side authentication is desireable then it must be provided by the application underneath the SASL layer. The use of TLS is strongly RECOMMENDED.

Additionally, the following aspects are worth pointing out:

An access token is not equivalent to the user’s long term password.

Care has to be taken when these OAuth credentials are used for actions like changing passwords (as it is possible with some

protocols, e.g., XMPP [RFC6120]). The resource server should ensure that actions taken in the authenticated channel are appropriate to the strength of the presented credential.

Lifetime of the application sessions.

It is possible that SASL will be used to authenticate a connection and the life of that connection may outlast the life of the access token used to establish it. This is a common problem in application protocols where connections are long-lived, and not a problem with this mechanism per se. Resource servers may unilaterally disconnect clients in accordance with the application protocol.

Access tokens have a lifetime.

Reducing the lifetime of an access token provides security benefits and OAuth 2.0 introduces refresh tokens to obtain new access token on the fly without any need for a human interaction. Additionally, a previously obtained access token might be revoked or rendered invalid at any time. The client MAY request a new access token for each connection to a resource server, but it SHOULD cache and re-use valid credentials.

6. Internationalization Considerations

The identifer asserted by the OAuth authorization server about the resource owner inside the access token may be displayed to a human. For example, when SASL is used in the context of IMAP the client may assert the resource owner’s email address to the IMAP server for usage in an email-based application. The identifier may therefore contain internationalized characters and an application needs to ensure that the mapping between the identifier provided by OAuth is suitable for use with the application layer protocol SASL is incorporated into.

At the time of writing the standardization of the various claims in the access token (in JSON format) is still ongoing, see [I-D.ietf-oauth-json-web-token]. Once completed it will provide a standardized format for exchanging identity information between the authorization server and the resource server.

7. IANA Considerations

7.1. SASL Registration

The IANA is requested to register the following entry in the SASL Mechanisms registry:

SASL mechanism name: OAUTHBEARER

Security Considerations: See this document

Published Specification: See this document

For further information: Contact the authors of this document.

Intended usage: common

Owner/Change controller: the IESG

Note: None

The IANA is requested to register the following entry in the SASL Mechanisms registry:

SASL mechanism name: OAUTH10A

Security Considerations: See this document

Published Specification: See this document

For further information: Contact the authors of this document.

Intended usage: common

Owner/Change controller: the IESG

Note: None

8. References

8.1. Normative References

[I-D.ietf-oauth-dyn-reg]
Richer, J., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", draft-ietf-oauth-dyn-reg-27 (work in progress), March 2015.

[OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", February 2014.

[OpenID.Discovery]
Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", July 2011.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5801] Josefsson, S. and N. Williams, "Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family", RFC 5801, July 2010.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

8.2. Informative References

[I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-32 (work in progress), December 2014.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, March 2003.

[RFC4959] Siemborski, R. and A. Gulbrandsen, "IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response", RFC 4959, September 2007.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

[RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, September 2013.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

Appendix A. Acknowlegements

The authors would like to thank the members of the Kitten working group, and in addition and specifically: Simon Josefson, Torsten Lodderstadt, Ryan Troll, Alexey Melnikov, Jeffrey Hutzelman, Nico Williams, Matt Miller, and Benjamin Kaduk.

This document was produced under the chairmanship of Alexey Melnikov, Tom Yu, Shawn Emery, Josh Howlett, Sam Hartman. The supervising area director was Stephen Farrell.

Appendix B. Document History

[[to be removed by RFC editor before publication as an RFC]]

-23

- o AD feedback from IESG review and comments.
- o Fixed port number in SMTP examples.
- o Minor editorial changes.
- o Dyn-Reg draft becomes normative.

- o Added explicit TLS start indicator in all examples, removed text that said we assume that.

-19

- o Last call feedback agaiun.
- o Clarified usage of TLS in examples and fixed them some more. Adding reference to RFC4422 and cancellation token and an example for that.

-18

- o Last call feedback round #5. Fixed -17 change log.
- o Corrected "issue" to "iss", other minor changes.

-17

- o Last call feedback again (WGLC #4). eradicated comma splicing. Removed extra server message in example 4.3.
- o Added recommendations for discovery and dynamic client registration support.

-16

- o Last call feedback again. Primarily editorial changes. Corrected examples.

-15

- o Last call feedack on the GS2 stuff being ripped out completely.
- o Removed the "user" parameter and put stuff back into the gs2-header. Call out that the authzid goes in the gs2-header with some prose about when it might be required. Very comparable to -10.
- o Added an OAuth 1.0A example explicitly.

-14

- o Last call feedack on RFC citations needed, small editorial.
- o Added the "user" parameter back, which was pulled when we started down the GS2 path. Same language as -03.

- o Defined a stub GS2 header to make sure that when the GS2 bride is defined for this that nothing will break when it actually starts to get populated.

-13

- o Changed affiliation.

-12

- o Removed -PLUS components from the specification.

-11

- o Removed GSS-API components from the specification.
- o Updated security consideration section.

-10

- o Clarifications throughout the document in response to the feedback from Jeffrey Hutzelman.

-09

- o Incorporated review by Alexey and Hannes.
- o Clarified the three OAuth SASL mechanisms.
- o Updated references
- o Extended acknowledgements

-08

- o Fixed the channel binding examples for p=$cbtype
- o More tuning of the authcid language and edited and renamed 3.2.1.

-07

- o Struck the MUST langiage from authzid.
- o

-06

- o Removed the user field. Fixed the examples again.

- Added canonicalization language.

-

-05

- Fixed the GS2 header language again.

- Separated out different OAuth schemes into different SASL mechanisms. Took out the scheme in the error return. Tuned up the IANA registrations.

- Added the user field back into the SASL message.

- Fixed the examples (again).

-

-04

- Changed user field to be carried in the gs2-header, and made gs2 header explicit in all cases.

- Converted MAC examples to OAuth 1.0a. Moved MAC to an informative reference.

- Changed to sending an empty client response (single control-A) as the second message of a failed sequence.

- Fixed channel binding prose to refer to the normative specs and removed the hashing of large channel binding data, which brought mroe problems than it solved.

- Added a SMTP examples for Bearer use case.

-03

- Added user field into examples and fixed egregious errors there as well.

- Added text reminding developers that Authorization scheme names are case insensitive.

-02

- Added the user data element back in.

- Minor editorial changes.

-01

- o Ripping out discovery. Changed to refer to I-D.jones-appsawg-webfinger instead of WF and SWD older drafts.

- o Replacing HTTP as the message format and adjusted all examples.

-00

- o Renamed draft into proper IETF naming format now that it's adopted.

- o Minor fixes.

Authors' Addresses

William Mills
Microsoft

Email: wimills@microsoft.com

Tim Showalter

Email: tjs@psaux.com

Hannes Tschofenig
ARM Ltd.
110 Fulbourn Rd
Cambridge CB1 9NJ
Great Britain

Email: Hannes.tschofenig@gmx.net
URI: http://www.tschofenig.priv.at

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 11, 2021

S. Cantor
Shibboleth Consortium
M. Cullen
Painless Security
S. Josefsson
SJD AB

May 10, 2021

SAML Enhanced Client SASL and GSS-API Mechanisms
draft-ietf-kitten-sasl-saml-ec-20

Abstract

Security Assertion Markup Language (SAML) 2.0 is a generalized framework for the exchange of security-related information between asserting and relying parties. Simple Authentication and Security Layer (SASL) and the Generic Security Service Application Program Interface (GSS-API) are application frameworks that facilitate an extensible authentication model, among other things. This document specifies a SASL and GSS-API mechanism for SAML 2.0 that leverages the capabilities of a SAML-aware "enhanced client" to address significant barriers to federated authentication in a manner that encourages reuse of existing SAML bindings and profiles designed for non-browser scenarios.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] is a modular specification that provides various means for a user to be identified to a relying party (RP) through the exchange of (typically signed) assertions issued by an identity provider (IdP).

Simple Authentication and Security Layer (SASL) [RFC4422] is a generalized mechanism for identifying and authenticating a user and for optionally negotiating a security layer for subsequent protocol interactions. SASL is used by application protocols like IMAP [RFC3501], the Post Office Protocol(POP [RFC1939]) and XMPP [RFC6120]. The effect of SASL is to make authentication modular, so that newer authentication mechanisms can be added as needed.

There are related protocols, protocol bindings [OASIS.saml-bindings-2.0-os], and interoperability profiles [OASIS.saml-profiles-2.0-os] designed for different use cases. Additional profiles and extensions are also routinely developed and published.

The Generic Security Service Application Program Interface (GSS-API) [RFC2743] provides a framework for applications to support multiple authentication mechanisms through a unified programming interface, as well as additional optional cryptographic functionality. This document defines a pure SASL mechanism for SAML, but it conforms to the bridge between SASL and GSS-API called GS2 [RFC5801]. This means that this document defines both a SASL mechanism and a GSS-API mechanism. The GSS-API interface is optional for SASL implementers, and the GSS-API considerations can be avoided in environments that use SASL directly without GSS-API.

The mechanisms specified in this document allow a SASL- or GSS-API-enabled server to act as a SAML relying party, or service provider (SP), by advertising this mechanism as an option for SASL or GSS-API clients that support the use of SAML to communicate identity and attribute information. Clients supporting this mechanism are termed "enhanced clients" in SAML terminology because they understand the federated authentication model and have specific knowledge of the IdP(s) associated with the user. This knowledge, and the ability to act on it, addresses a significant problem with browser-based SAML profiles known as the "discovery", or "where are you from?" (WAYF) problem. In a "dumb" client such as a web browser, various intrusive user interface techniques are used to determine the appropriate IdP to use because the request to the IdP is generated as an HTTP

redirect by the RP, which does not generally have prior knowledge of the IdP to use. Obviating the need for the RP to interact with the client to determine the right IdP (and its network location) is both a user interface and security improvement.

The SAML mechanism described in this document is an adaptation of an existing SAML profile, the Enhanced Client or Proxy (ECP) Profile (V2.0) [SAMLECP20].

Figure 1 describes the interworking between SAML and SASL: this document requires enhancements to the RP and to the client (as the two SASL communication endpoints) but no changes to the SAML IdP are assumed apart from its support for the applicable SAML profile. To accomplish this, a SAML protocol exchange between the RP and the IdP, brokered by the client, is tunneled within SASL. There is no assumed communication between the RP and the IdP, but such communication may occur in conjunction with additional SAML-related profiles not in scope for this document.

```
                                      +-----------+
                                      |  SAML     |
                                      |  Relying  |
                                      |  Party    |
                                      |           |
                                      +-----------+
                                            ^
                                         +--|--+
                                         | S|  |
                                       S | A|  |
                                       A | M|  |
                                       S | L|  |
                                       L |  |  |
                                         |  |  |
                                         +--|--+
          +------------+                     v
          |            |                 +----------+
          |  SAML      |     SAML SOAP   |          |
          |  Identity  |<--------------->|  Client  |
          |  Provider  |      Binding    |          |
          +------------+                 +----------+
```

Figure 1: Interworking Architecture

```
         +-------+               +-------+               +-------+
         | SAML  |               |Client |               | SAML  |
         | IDP   |               |       |               | RP    |
         +---+---+               +---+---+               +---+---+
             |                       |                       |
             |                       |---------------------->|
             |                       |   Resource Request    |
             |                       |                       |
             |                       |                       |
             |<----------------------+-----------------------|
             |                       |   SAML Auth Request   |
             |                       |                       |
             |                       |                       |
             |<--------------------->|                       |
             |  User Authentication  |                       |
             |                       |                       |
             |                       |                       |
             |-----------------------+---------------------->|
             |  SAML Auth Response   |                       |
             |                       |                       |
             |                       |                       |
             |                       |<----------------------|
             |                       |  Requested Resource   |
             |                       |                       |
```

Figure 2: Communication Flow

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here.

The reader is also assumed to be familiar with the terms used in the SAML 2.0 specification, and an understanding of the Enhanced Client or Proxy (ECP) Profile (V2.0) [SAMLECP20] is necessary, as part of this mechanism explicitly reuses and references it.

This document can be implemented without knowledge of GSS-API since the normative aspects of the GS2 protocol syntax have been duplicated in this document. The document may also be implemented to provide a GSS-API mechanism, and then knowledge of GSS-API is essential.

3. Applicability for Non-HTTP Use Cases

While SAML is designed to support a variety of application scenarios, the profiles for authentication defined in the original standard are designed around HTTP [RFC7230] applications. They are not, however, limited to browsers, because browsers do not always meet the needs of more security-sensitive applications. Specifically, the notion of an "Enhanced Client" (or a proxy acting as one on behalf of a browser, thus the term "ECP") was specified for a software component that acts somewhat like a browser from an application perspective, but includes limited, but sufficient, awareness of SAML to play a more conscious role in the authentication exchange between the RP and the IdP. What follows is an outline of the Enhanced Client or Proxy (ECP) Profile (V2.0) [SAMLECP20], as applied to the web/HTTP service use case:

1. The Enhanced Client requests a resource of a Relying Party (RP) (via an HTTP request). In doing so, it advertises its "enhanced" capability using HTTP headers.

2. The RP, desiring SAML authentication and noting the client's capabilities, responds not with an HTTP redirect or form, but with a SOAP [W3C.soap11] envelope containing a SAML <AuthnRequest> along with some supporting headers. This request identifies the RP (and may be signed), and may provide hints to the client as to what IdPs the RP finds acceptable, but the choice of IdP is generally left to the client.

3. The client is then responsible for delivering the body of the SOAP message to the IdP it is instructed to use (often via out-of-band configuration). The user authenticates to the IdP ahead of, during, or after the delivery of this message, and perhaps explicitly authorizes the response to the RP.

4. Whether authentication succeeds or fails, the IdP responds with its own SOAP envelope, generally containing a SAML <Response> message for delivery to the RP. In a successful case, the message will include one or more SAML <Assertion> elements containing authentication, and possibly attribute, statements about the subject. Either the response or each assertion is signed, and the assertion(s) may be encrypted to a key negotiated with or known to belong to the RP.

5. The client then delivers the SOAP envelope containing the <Response> to the RP at a location the IdP directs (which acts as an additional, though limited, defense against MITM attacks). This completes the SAML exchange.

6. The RP now has sufficient identity information to approve the original HTTP request or not, and acts accordingly. Everything between the original request and this response can be thought of as an "interruption" of the original HTTP exchange.

When considering this flow in the context of an arbitrary application protocol and SASL, the RP and the client both must change their code to implement this SASL mechanism, but the IdP can remain unmodified. The existing RP/client exchange that is tunneled through HTTP maps well to the tunneling of that same exchange in SASL. In the parlance of SASL [RFC4422], this mechanism is "client-first" for consistency with GS2. The steps are shown below:

1. The server MAY advertise the SAML20EC and/or SAML20EC-PLUS mechanisms.

2. The client initiates a SASL authentication with SAML20EC or SAML20EC-PLUS.

3. The server sends the client a challenge consisting of a SOAP envelope containing its SAML <AuthnRequest>.

4. The SASL client unpacks the SOAP message and communicates with its chosen IdP to relay the SAML <AuthnRequest> to it. This communication, and the authentication with the IdP, proceeds separately from the SASL process.

5. Upon completion of the exchange with the IdP, the client responds to the SASL server with a SOAP envelope containing the SAML <Response> it obtained, or a SOAP fault, as warranted.

6. The SASL Server indicates success or failure.

Note: The details of the SAML processing, which are consistent with the Enhanced Client or Proxy (ECP) Profile (V2.0) [SAMLECP20], are such that the client MUST interact with the IdP in order to complete any SASL exchange with the RP. The assertions issued by the IdP for the purposes of the profile, and by extension this SASL mechanism, are short lived, and therefore cannot be cached by the client for later use.

Encompassed in step four is the client-driven selection of the IdP, authentication to it, and the acquisition of a response to provide to the SASL server. These processes are all external to SASL.

Note also that unlike an HTTP-based profile, the IdP cannot participate in the selection of, or evaluation of, the location to which the SASL Client Response will be delivered by the client. The

use of GSS-API Channel Binding is an important mitigation of the risk of a "Man in the Middle" attack between the client and RP, as is the use of a negotiated or derived session key in whatever protocol is secured by this mechanism.

With all of this in mind, the typical flow appears as follows:

```
   SASL Serv.       Client          IdP
      |------(1)----->|              | Advertisement
      |               |              |
      |<-----(2)------|              | Initiation
      |               |              |
      |------(3)----->|              | SASL Server Response
      |               |              |
      |               |<- - -(4)- - >| SOAP AuthnRequest + user authn
      |               |              |
      |<-----(5)------|              | SASL Client Response
      |               |              |
      |------(6)----->|              | Server sends Outcome
      |               |              |

    ----- = SASL
    - - - = SOAP over HTTPS (external to SASL)
```

Figure 3: Authentication flow

4. SAML Enhanced Client SASL Mechanism Specification

Based on the previous figures, the following operations are defined by the SAML SASL mechanism:

4.1. Advertisement

To advertise that a server supports this mechanism, during application session initiation, it displays the name "SAML20EC" and/or "SAML20EC-PLUS" in the list of supported SASL mechanisms.

In accordance with [RFC5801] the "-PLUS" variant indicates that the server supports channel binding and would be selected by a client with that capability.

4.2. Initiation

A client initiates "SAML20EC" or "SAML20EC-PLUS" authentication. If supported by the application protocol, the client MAY include an initial response, otherwise it waits until the server has issued an empty challenge (because the mechanism is client-first).

The format of the initial client response ("initresp") is as follows:

```
hok = "urn:oasis:names:tc:SAML:2.0:cm:holder-of-key"

mut = "urn:oasis:names:tc:SAML:2.0:profiles:SSO:ecp:2.0:" \
    "WantAuthnRequestsSigned"

del = "urn:oasis:names:tc:SAML:2.0:conditions:delegation"

initresp = gs2-cb-flag "," [gs2-authzid] "," [hok] "," [mut] "," [del]
```

The gs2-cb-flag flag MUST be set as defined in [RFC5801] to indicate whether the client supports channel binding. This takes the place of the PAOS HTTP header extension used in [SAMLECP20] to indicate channel binding support.

The optional "gs2-authzid" field holds the authorization identity, as requested by the client.

The optional "hok" field is a constant that signals the client’s support for stronger security by means of a locally held key. This takes the place of the PAOS HTTP header extension used in [SAMLECP20] to indicate "holder of key" support.

The optional "mut" field is a constant that signals the client’s desire for mutual authentication. If set, the SASL server MUST digitally sign its SAML <AuthnRequest> message. The URN constant above is a single string; the linefeed is shown for RFC formatting reasons.

The optional "del" field is a constant that signals the client’s desire for the acceptor to request an assertion usable for delegation of the client’s identity to the acceptor.

4.3. Server Response

The SASL server responds with a SOAP envelope constructed in accordance with section 2.3.2 of [SAMLECP20]. This includes adhering to the SOAP header requirements of the SAML PAOS Binding [OASIS.saml-bindings-2.0-os], for compatibility with the existing

profile. Various SOAP headers are also consumed by the client in exactly the same manner prescribed by that section.

4.4. User Authentication with Identity Provider

Upon receipt of the Server Response (Section 4.3), the steps described in sections 2.3.3 through 2.3.6 of [SAMLECP20] are performed between the client and the chosen IdP. The means by which the client determines the IdP to use, and where it is located, are out of scope of this mechanism.

The exact means of authentication to the IdP are also out of scope, but clients supporting this mechanism MUST support HTTP Basic Authentication as defined in [RFC7617] and TLS 1.3 client authentication as defined in [RFC8446].

4.5. Client Response

Assuming a response is obtained from the IdP, the client responds to the SASL server with a SOAP envelope constructed in accordance with section 2.3.7 of [SAMLECP20]. This includes adhering to the SOAP header requirements of the SAML PAOS Binding [OASIS.saml-bindings-2.0-os], for compatibility with the existing profile. If the client is unable to obtain a response from the IdP, or must otherwise signal failure, it responds to the SASL server with a SOAP envelope containing a SOAP fault.

4.6. Outcome

The SAML protocol exchange having completed, the SASL server will transmit the outcome to the client depending on local validation of the client responses (including the assertion conveyed from the chosen IDP). This outcome is transmitted in accordance with the application protocol in use.

4.7. Additional Notes

Because this mechanism is an adaptation of an HTTP-based profile, there are a few requirements outlined in [SAMLECP20] that make reference to a response URL that is normally used to regulate where the client returns information to the RP. There are also security-related checks built into the profile that involve this location.

For compatibility with existing IdP and profile behavior, and to provide for mutual authentication, the SASL server MUST populate the responseConsumerURL and AssertionConsumerServiceURL attributes with its service name. As discussed in Section 5.6.2, most SASL profiles rely on a service name format of "service@host", but regardless of

the form, the service name is used directly rather than transformed into an absolute URI if it is not already one, and MUST be percent-encoded per [RFC3986].

The IdP MUST securely associate the service name with the SAML entityID claimed by the SASL server, such as through the use of SAML metadata [OASIS.saml-metadata-2.0-os]. If metadata is used, a SASL service's <SPSSODescriptor> role MUST contain a corresponding <AssertionConsumerService> whose Location attribute contains the appropriate service name, as described above. The Binding attribute MUST be one of "urn:ietf:params:xml:ns:samlec" (RECOMMENDED) or "urn:oasis:names:tc:SAML:2.0:bindings:PAOS" (for compatibility with older implementations of the ECP profile in existing IdP software).

Finally, note that the use of HTTP status signaling between the RP and client mandated by [SAMLECP20] may not be applicable.

5. SAML EC GSS-API Mechanism Specification

This section and its sub-sections and all normative references of it not referenced elsewhere in this document are INFORMATIONAL for SASL implementors, but they are NORMATIVE for GSS-API implementors.

The SAML Enhanced Client SASL mechanism is also a GSS-API mechanism. The messages are the same, but a) the GS2 [RFC5801] header on the client's first authentication message is excluded when SAML EC is used as a GSS-API mechanism, and b) the [RFC2743] section 3.1 initial context token header is used for the client's first authentication message (context token) instead, with the body of the message being the same as for the SASL mechanism case.

The GSS-API mechanism OID for SAML EC is OID-TBD (IANA to assign: see IANA considerations). The DER encoding of the OID is TBD.

The mutual_state request flag (GSS_C_MUTUAL_FLAG) MAY be set to TRUE, resulting in the "mut" option set in the initial client response. The security context mutual_state flag is set to TRUE only if the server digitally signs its SAML <AuthnRequest> message and the signature and signing credential are appropriately verified by the IdP. The IdP signals this to the client in an <ecp:RequestAuthenticated> SOAP header block.

The lifetime of a security context established with this mechanism SHOULD be limited by the value of a SessionNotOnOrAfter attribute, if any, in the <AuthnStatement> element(s) of the SAML assertion(s) received by the RP. By convention, in the rare case that multiple valid/confirmed assertions containing <AuthnStatement> elements are

received, the most restrictive SessionNotOnOrAfter is generally applied.

5.1. GSS-API Credential Delegation

This mechanism can support credential delegation through the issuance of SAML assertions that an IdP will accept as proof of authentication by a service on behalf of a subject. An initiator may request delegation of its credentials by setting the "del" option field in the initial client response to "urn:oasis:names:tc:SAML:2.0:conditions:delegation".

An acceptor, upon receipt of this constant, requests a delegated assertion by including in its <AuthnRequest> message a <Conditions> element containing an <AudienceRestriction> identifying the IdP as a desired audience for the assertion(s) to be issued. In the event that the specific IdP to be used is unknown, the constant "urn:oasis:names:tc:SAML:2.0:conditions:delegation" may be used as a stand-in, per Section 2.3.2 of [SAMLECP20].

Upon receipt of an assertion satisfying this property, and containing a <SubjectConfirmation> element that the acceptor can satisfy, the security context will have its deleg_state flag (GSS_C_DELEG_FLAG) set to TRUE.

The IdP, if it issues a delegated assertion to the acceptor, MUST include in the SOAP response to the initiator a <samlec:Delegated> SOAP header block, indicating that delegation was enabled. It has no content, other than mandatory SOAP attributes (an example follows):

```
<samlec:Delegated xmlns:samlec="urn:ietf:params:xml:ns:samlec"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    S:mustUnderstand="1"
    S:actor="http://schemas.xmlsoap.org/soap/actor/next" />
```

Upon receipt of such a header block, the initiator MUST fail the establishment of the security context if it did not request delegation in its initial client response to the acceptor. It SHOULD signal this failure to the acceptor with a SOAP fault message in its final client response.

As noted previously, the exact means of client authentication to the IdP is formally out of scope of this mechanism. This extends to the use of a delegation assertion as a means of authentication by an acceptor acting as an initiator. In practice, some profile of

[WSS-SAML] is used to attach the assertion and a confirmation proof to the SOAP message from the client to the IdP.

5.2. GSS-API Channel Binding

GSS-API channel binding [RFC5554] is a protected facility for exchanging a cryptographic identifier for an enclosing channel between the initiator and acceptor. The initiator sends channel binding data and the acceptor confirms that channel binding data has been checked.

The acceptor SHOULD accept any channel binding provided by the initiator if null channel bindings are passed into gss_accept_sec_context. Protocols such as HTTP Negotiate [RFC4559] depend on this behavior of some Kerberos implementations.

The exchange and verification of channel binding information is described by [SAMLECP20].

5.3. Session Key Derivation

Some GSS-API features (discussed in the following sections) require a session key be established as a result of security context establishment. In the common case of a "bearer" assertion in SAML, a mechanism is defined to communicate a key to both parties via the IdP. In other cases such as assertions based on "holder of key" confirmation bound to a client-controlled key, there may be additional methods defined in the future, and extension points are provided for this purpose.

Information defining or describing the session key, or a process for deriving one, is communicated between the initiator and acceptor using a <samlec:SessionKey> element, defined by the XML schema in Appendix A. This element is a SOAP header block. The content of the element further depends on the specific use in the mechanism. The Algorithm XML attribute identifies a mechanism for key derivation. It is omitted to identify the use of an IdP-generated key (see following section) or will contain a URI value identifying a derivation mechanism defined outside this specification. Each header block’s mustUnderstand and actor attributes MUST be set to "1" and "http://schemas.xmlsoap.org/soap/actor/next" respectively.

In the acceptor’s first response message containing its SAML request, one or more <samlec:SessionKey> SOAP header blocks MUST be included. The element MUST contain one or more <EncType> elements containing the number of a supported encryption type defined in accordance with [RFC3961]. Encryption types should be provided in order of preference by the acceptor.

In the final client response message, a single <samlec:SessionKey> SOAP header block MUST be included. A single <EncType> element MUST be included to identify the chosen encryption type used by the initiator.

All parties MUST support the "aes128-cts-hmac-sha1-96" encryption type, number 17, defined by [RFC3962].

Further details depend on the mechanism used, one of which is described in the following section.

5.3.1. Generated by Identity Provider

The IdP, if issuing a bearer assertion for use with this mechanism, SHOULD provide a generated key for use by the initiator and acceptor. This key is used as pseudorandom input to the "random-to-key" function for a specific encryption type defined in accordance with [RFC3961]. The key is base64-encoded and placed inside a <samlec:GeneratedKey> element. The IdP does not participate in the selection of the encryption type and simply generates enough pseudorandom bits to supply key material to the other parties.

The resulting <samlec:GeneratedKey> element is placed within the <saml:Advice> element of the assertion issued. The identity provider MUST encrypt the assertion (implying that it MUST have the means to do so, typically knowledge of a key associated with the RP). If multiple assertions are issued (allowed, but not typical), the element need only be included in one of the assertions issued for use by the relying party.

A copy of the element is also added as a SOAP header block in the response from the IdP to the client (and then removed when constructing the response to the acceptor).

If this mechanism is used by the initiator, then the <samlec:SessionKey> SOAP header block attached to the final client response message will identify this via the omission of the Algorithm attribute and will identify the chosen encryption type using the <samlec:EncType> element:

```
<samlec:SessionKey xmlns:samlec="urn:ietf:params:xml:ns:samlec"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    S:mustUnderstand="1"
    S:actor="http://schemas.xmlsoap.org/soap/actor/next">
  <samlec:EncType>17</samlec:EncType>
<samlec:SessionKey>
```

Both the initiator and acceptor MUST execute the chosen encryption type's random-to-key function over the pseudorandom value provided by the <samlec:GeneratedKey> element. The result of that function is used as the protocol and session key. Support for subkeys from the initiator or acceptor is not specified.

5.3.2. Alternate Key Derivation Mechanisms

In the event that a client is proving possession of a secret or private key, a formal key agreement algorithm might be supported. This specification does not define such a mechanism, but the <samlec:SessionKey> element is extensible to allow for future work in this space by means of the Algorithm attribute and an optional <ds:KeyInfo> child element to carry extensible content related to key establishment.

However a key is derived, the <samlec:EncType> element will identify the chosen encrytion type, and both the initiator and acceptor MUST execute the encryption type's random-to-key function over the result of the key agreement or derivation process. The result of that function is used as the protocol key.

5.4. Per-Message Tokens

The per-message tokens SHALL be the same as those for the Kerberos V5 GSS-API mechanism [RFC4121] (see Section 4.2 and sub-sections).

The replay_det_state (GSS_C_REPLAY_FLAG), sequence_state (GSS_C_SEQUENCE_FLAG), conf_avail (GSS_C_CONF_FLAG) and integ_avail (GSS_C_INTEG_FLAG) security context flags are always set to TRUE.

The "protocol key" SHALL be a key established in a manner described in the previous section. "Specific keys" are then derived as usual as described in Section 2 of [RFC4121], [RFC3961], and [RFC3962].

The terms "protocol key" and "specific key" are Kerberos V5 terms [RFC3961].

SAML20EC is PROT_READY as soon as the SAML response message has been seen.

5.5. Pseudo-Random Function (PRF)

The GSS-API has been extended with a Pseudo-Random Function (PRF) interface in [RFC4401]. The purpose is to enable applications to derive a cryptographic key from an established GSS-API security context. This section defines a GSS_Pseudo_random that is applicable for the SAML20EC GSS-API mechanism.

The GSS_Pseudo_random() [RFC4401] SHALL be the same as for the Kerberos V5 GSS-API mechanism [RFC7802]. There is no acceptor-asserted sub-session key, thus GSS_C_PRF_KEY_FULL and GSS_C_PRF_KEY_PARTIAL are equivalent. The protocol key to be used for the GSS_Pseudo_random() SHALL be the same as the key defined in the previous section.

5.6. GSS-API Principal Name Types for SAML EC

Services that act as SAML relying parties are typically identified by means of a URI called an "entityID". Clients that are named in the <Subject> element of a SAML assertion are typically identified by means of a <NameID> element, which is an extensible XML structure containing, at minimum, an element value that names the subject and a Format attribute.

In practice, a GSS-API client and server are unlikely to know in advance the name of the initiator as it will be expressed by the SAML IdP upon completion of authentication. It is also generally incorrect to assume that a particular acceptor name will directly map into a particular RP entityID, because there is often a layer of naming indirection between particular services on hosts and the identity of a relying party in SAML terms.

To avoid complexity, and avoid unnecessary use of XML within the naming layer, the SAML EC mechanism relies on the common/expected name types used for acceptors and initiators, GSS_C_NT_HOSTBASED_SERVICE and GSS_C_NT_USER_NAME. The mechanism provides for validation of the host-based service name in conjunction with the SAML exchange. It does not attempt to solve the problem of mapping between an initiator "username", the user’s identity while authenticating to the IdP, and the information supplied by the IdP to the acceptor. These relationships must be managed through local policy at the initiator and acceptor.

SAML-based information associated with the initiator SHOULD be expressed to the acceptor using GSS-API naming extensions [RFC6680], in a similar manner to [RFC7056].

5.6.1. User Naming Considerations

The GSS_C_NT_USER_NAME form represents the name of an individual user. Clients often rely on this value to determine the appropriate credentials to use in authenticating to the IdP, and supply it to the server for use by the acceptor.

Upon successful completion of this mechanism, the server MUST construct the authenticated initiator name based on the <saml:NameID>

element in the assertion it successfully validated. The name is constructed as a UTF-8 string in the following form:

```
name = element-value "!" Format  "!" NameQualifier
    "!" SPNameQualifier "!" SPProvidedID
```

The "element-value" token refers to the content of the <saml:NameID> element. The other tokens refer to the identically named XML attributes defined for use with the element. If an attribute is not present, which is common, it is omitted (i.e., replaced with the empty string). The Format value is never omitted; if not present, the SAML-equivalent value of "urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified" is used.

Not all SAML assertions contain a <saml:NameID> element. In the event that no such element is present, including the exceptional cases of a <saml:BaseID> element or a <saml:EncryptedID> element that cannot be decrypted, the GSS_C_NT_ANONYMOUS name type MUST be used for the initiator name.

As noted in the previous section, it is expected that most applications able to rely on SAML authentication would make use of naming extensions to obtain additional information about the user based on the assertion. This is particularly true in the anonymous case, or in cases in which the SAML name is pseudonymous or transient in nature. The ability to express the SAML name in GSS_C_NT_USER_NAME form is intended for compatibility with applications that cannot make use of additional information.

5.6.2. Service Naming Considerations

The GSS_C_NT_HOSTBASED_SERVICE name form represents a service running on a host; it is textually represented as "service@host". This name form is required by most SASL profiles and is used by many existing applications that use the Kerberos GSS-API mechanism. As described in in the SASL mechanism’s Section 4.7, such a name is used directly by this mechanism as the effective AssertionConsumerService "location" associated with the service and applied in IdP verification of the request against the claimed SAML entityID.

6. Example

Suppose the user has an identity at the SAML IdP saml.example.org and a Jabber Identifier (jid) "somenode@example.com", and wishes to authenticate his XMPP connection to xmpp.example.com (and example.com and example.org have established a SAML-capable trust relationship). The authentication on the wire would then look something like the following:

Step 1: Client initiates stream to server:

```
<stream:stream xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
to='example.com' version='1.0'>
```

Step 2: Server responds with a stream tag sent to client:

```
<stream:stream
xmlns='jabber:client' xmlns:stream='http://etherx.jabber.org/streams'
id='some_id' from='example.com' version='1.0'>
```

Step 3: Server informs client of available authentication mechanisms:

```
<stream:features>
 <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
  <mechanism>DIGEST-MD5</mechanism>
  <mechanism>PLAIN</mechanism>
  <mechanism>SAML20EC</mechanism>
 </mechanisms>
</stream:features>
```

Step 4: Client selects an authentication mechanism and sends the initial client response (it is base64 encoded as specified by the XMPP SASL protocol profile):

```
<auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl' mechanism='SAML20EC'>
biwsLCw=
</auth>
```

The initial response is "n,,,," which signals that channel binding is not used, there is no authorization identity, and the client does not support key-based confirmation, or want mutual authentication or delegation.

Step 5: Server sends a challenge to client in the form of a SOAP envelope containing its SAML <AuthnRequest>:

```
<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
PFM6RW52ZWxvcGUKICAgIHhtbG5zOnNhbWw9InVybjpvYXNpczpuYW1lczp0YzpT
QU1MOjIuMDphc3NlcnRpb24iCiAgICB4bWxuczpzYW1scD0idXJuOm9hc2lzOm5h
bWVzOnRjOlNBTUw6Mi4wOnByb3RvY29sIgogICAgeG1sbnM6Uz0iaHR0cDovL3Nj
aGVtYXMueG1sc29hcC5vcmcvc29hcC9lbnZlbG9wZS8iPgogIDxTOkhlYWRlcj4K
ICAgIDxwYW9zOlJlcXVlc3QgeG1sbnM6cGFvcz0idXJuOmxpYmVydHk6cGFvczoy
MDAzLTA4IgogICAgICBtZXNzYWdlSUQ9ImMzYTRmOGI5YzJkIiBTOm11c3RVbmRl
cnN0YW5kPSIxIgogICAgICBTOmFjdG9yPSJodHRwOi8vc2NoZW1hcy54bWxzb2Fw
Lm9yZy9zb2FwL2FjdG9yL25leHQiCiAgICAgIHJlc3BvbnNlQ29uc3VtZXJVUkw9
InhtcHBAeG1wcC5leGFtcGxlLmNvbSIKICAgICAgc2Vydmljj0idXJuOm9hc2lz
Om5hbWVzOnRjOlNBTUw6Mi4wOnByb2ZpbGVzOlNTTzplY3AiLz4KICAgIDxlY3A6
UmVxdWVzdAogICAgICB4bWxuczplY3A9InVybjpvYXNpczpuYW1lczp0YzpTQU1M
OjIuMDpwcm9maWxlczpTU086ZWNwIgogICAgICBTOmFjdG9yPSJodHRwOi8vc2No
ZW1hcy54bWxzb2FwLm9yZy9zb2FwL2FjdG9yL25leHQiCiAgICAgIFM6bXVzdFVu
ZGVyc3RhbmQ9IjEiIFByb3ZpZGVyTmFtZT0iSmFiYmVyIGF0IGV4YW1wbGUuY29t
Ij4KICAgICAgPHNhbWw6SXNzdWVyPmh0dHBzOi8veG1wcC5leGFtcGxlLmNvbTwv
c2FtbDpJc3N1ZXI+CiAgICA8L2VjcDpSZXF1ZXN0PgogICAgPHNhbWxlYzpTZXNz
aW9uS2V5IHhtbG5zOnNhbWxlYz0idXJuOmlldGY6cGFyYW1zOnhtbDpuczpzYW1s
ZWMiCiAgICAgIHhtbG5zOlM9Imh0dHA6Ly9zY2hlbWFzLnhtbHNvYXAub3JnL3Nv
YXAvZW52ZWxvcGUvIgogICAgICBTOm11c3RVbmRlcnN0YW5kPSIxIgogICAgICBT
OmFjdG9yPSJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy9zb2FwL2FjdG9yL25l
eHQiPgogICAgICA8c2FtbGVjOkVuY1R5cGU+MTc8L3NhbWxlYzpFbmNUeXBlPgog
ICAgICA8c2FtbGVjOkVuY1R5cGU+MTg8L3NhbWxlYzpFbmNUeXBlPgogICAgPHNh
bWxlYzpTZXNzaW9uS2V5PgogIDwvUzpIZWFkZXI+CiAgPFM6Qm9keT4KICAgIDxz
YW1scDpBdXRoblJlcXVlc3QKICAgICAgSUQ9ImMzYTRmOGI5YzJkIiBWZXJzaW9u
PSIyLjAiIElzc3VlSW5zdGFudD0iMjAwNy0xMi0xMFQxMTozOTozNFoiCiAgICAg
IEFzc2VydGlvbkNvbnN1bWVyU2VydmljZVVSTD0ieG1wcEB4bXBwLmV4YW1wbGUu
Y29tIj4KICAgICAgPHNhbWw6SXNzdWVyIHhtbG5zOnNhbWw9InVybjpvYXNpczpu
YW1lczp0YzpTQU1MOjIuMDphc3NlcnRpb24iPgogICAgICAgaHR0cHM6Ly94bXBw
LmV4YW1wbGUuY29tCiAgICAgIDwvc2FtbDpJc3N1ZXI+CiAgICAgIDxzYW1scDpO
YW1lSURQb2xpY3kgQWxsb3dDcmVhdGU9InRydWUiCiAgICAgICAgRm9ybWF0PSJ1
cm46b2FzaXM6bmFtZXM6dGM6U0FNTDoyLjA6bmFtZWlkLWZvcm1hdDpwZXJzaXN0
ZW50Ii8+CiAgICAgIDxzYW1scDpSZXF1ZXN0ZWRBdXRobkNvbnRleHQgQ29tcGFy
aXNvbj0iZXhhY3QiPgogICAgICAgPHNhbWw6QXV0aG5Db250ZXh0Q2xhc3NSZWY+
CiAgICAgICB1cm46b2FzaXM6bmFtZXM6dGM6U0FNTDoyLjA6YWM6Y2xhc3NlczpQ
YXNzd29yZFByb3RlY3RlZFRyYW5zcG9ydAogICAgICAgPC9zYW1sOkF1dGhuQ29u
dGV4dENsYXNzUmVmPgogICAgICA8L3NhbWxwOlJlcXVlc3RlZEF1dGhuQ29udGV4
dD4gCiAgICA8L3NhbWxwOkF1dGhuUmVxdWVzdD4KICA8L1M6Qm9keT4KPC9TOkVu
dmVsb3BlPgo=
</challenge>
```


The Base64 [RFC4648] decoded envelope:

```
<S:Envelope
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <paos:Request xmlns:paos="urn:liberty:paos:2003-08"
      messageID="c3a4f8b9c2d" S:mustUnderstand="1"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next"
      responseConsumerURL="xmpp@xmpp.example.com"
      service="urn:oasis:names:tc:SAML:2.0:profiles:SSO:ecp"/>
    <ecp:Request
      xmlns:ecp="urn:oasis:names:tc:SAML:2.0:profiles:SSO:ecp"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next"
      S:mustUnderstand="1" ProviderName="Jabber at example.com">
      <saml:Issuer>https://xmpp.example.com</saml:Issuer>
    </ecp:Request>
    <samlec:SessionKey xmlns:samlec="urn:ietf:params:xml:ns:samlec"
      xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
      S:mustUnderstand="1"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next">
      <samlec:EncType>17</samlec:EncType>
      <samlec:EncType>18</samlec:EncType>
    <samlec:SessionKey>
  </S:Header>
  <S:Body>
    <samlp:AuthnRequest
      ID="c3a4f8b9c2d" Version="2.0" IssueInstant="2020-12-10T11:39:34Z"
      AssertionConsumerServiceURL="xmpp@xmpp.example.com">
      <saml:Issuer xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">
       https://xmpp.example.com
      </saml:Issuer>
      <samlp:NameIDPolicy AllowCreate="true"
        Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent"/>
      <samlp:RequestedAuthnContext Comparison="exact">
       <saml:AuthnContextClassRef>
       urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
       </saml:AuthnContextClassRef>
      </samlp:RequestedAuthnContext>
    </samlp:AuthnRequest>
  </S:Body>
</S:Envelope>
```

Step 5 (alt): Server returns error to client:

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
 <incorrect-encoding/>
</failure>
</stream:stream>
```

Step 6: Client relays the request to IdP in a SOAP message transmitted over HTTP (over TLS). The HTTP portion is not shown, so the use of Basic Authentication is assumed. The body of the SOAP envelope is exactly the same as received in the previous step.

```
<S:Envelope
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <samlp:AuthnRequest>
        <!-- same as above -->
        </samlp:AuthnRequest>
    </S:Body>
</S:Envelope>
```

Step 7: IdP responds to client with a SOAP response containing a SAML <Response> containing a short-lived SSO assertion (shown as an encrypted variant in the example). A generated key is included in the assertion and in a header for the client.

```
<S:Envelope
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <ecp:Response S:mustUnderstand="1"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next"
      AssertionConsumerServiceURL="xmpp@xmpp.example.com"/>
    <samlec:GeneratedKey xmlns:samlec="urn:ietf:params:xml:ns:samlec">
      3w1wSBKUosRLsU69xGK7dg==
    </samlec:GeneratedKey>
  </S:Header>
  <S:Body>
    <samlp:Response ID="d43h94r389309r" Version="2.0"
        IssueInstant="2020-12-10T11:42:34Z" InResponseTo="c3a4f8b9c2d"
        Destination="xmpp@xmpp.example.com">
      <saml:Issuer>https://saml.example.org</saml:Issuer>
      <samlp:Status>
        <samlp:StatusCode
            Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
      </samlp:Status>
      <saml:EncryptedAssertion>
        <!-- contents elided, copy of samlec:GeneratedKey in Advice -->
      </saml:EncryptedAssertion>
    </samlp:Response>
  </S:Body>
</S:Envelope>
```

Step 8: Client sends SOAP envelope containing the SAML <Response> as a response to the SASL server’s challenge:

```
<response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
PFM6RW52ZWxvcGUKICAgIHhtbG5zOnNhbWw9InVybjpvYXNpczpuYW1lczp0YzpT
QU1MOjIuMDphc3NlcnRpb24iCiAgICB4bWxuczpzYW1scD0idXJuOm9hc2lzOm5h
bWVzOnRjOlNBTUw6Mi4wOnByb3RvY29sIgogICAgeG1sbnM6Uz0iaHR0cDovL3Nj
aGVtYXMueG1sc29hcC5vcmcvc29hcC9lbnZlbG9wZS8iPgogIDxTOkhlYWRlcj4K
ICAgIDxwYW9zOlJlc3BvbnNlIHhtbG5zOnBhb3M9InVybjpsaWJlcnR5OnBhb3M6
MjAwMy0wOCIKICAgICAgUzphY3Rvcj0iaHR0cDovL3NjaGVtYXMueG1sc29hcC5v
cmcvc29hcC9hY3Rvci9uZXh0IgogICAgICBTOm11c3RVbmRlcnN0YW5kPSIxIiBy
ZWZUb01lc3NhZ2VJRD0iNmMzYTRmOGI5YzJkIi8+CiAgICA8c2FtbGVjOlNlc3Np
b25LZXkgeG1sbnM6c2FtbGVjPSJ1cm46aWV0ZjpwYXJhbXM6eG1sOm5zOnNhbWxl
YyIKICAgICAgeG1sbnM6Uz0iaHR0cDovL3NjaGVtYXMueG1sc29hcC5vcmcvc29h
cC9lbnZlbG9wZS8iCiAgICAgIFM6bXVzdFVuZGVyc3RhbmQ9IjEiCiAgICAgIFM6
YWN0b3I9Imh0dHA6Ly9zY2hlbWFzLnhtbHNvYXAub3JnL3NvYXAvYWN0b3IvbmV4
dCI+CiAgICAgIDxzYW1sZWM6RW5jVHlwZT5hZXMxMjgtY3RzLWhtYWMtc2hhMS05
Njwvc2FtbGVjOkVuY1R5cGU+CiAgICA8c2FtbGVjOlNlc3Npb25LZXk+CiAgPC9T
OkhlYWRlcj4KICA8UzpCb2R5PgogICAgPHNhbWxwOlJlc3BvbnNlIElEPSJkNDNo
OTRyMzg5MzA5ciIgVmVyc2lvbj0iMi4wIgogICAgICAgIElzc3VlSW5zdGFudD0i
MjAwNy0xMi0xMFQxMTo0MjozNFoiIEluUmVzcG9uc2VUbz0iYzNhNGY4YjljMmQi
CiAgICAgICAgRGVzdGluYXRpb249InhtcHBAeG1wcC5leGFtcGxlLmNvbSI+CiAg
ICAgIDxzYW1sOklzc3Vlcj5odHRwczovL3NhbWwuZXhhbXBsZS5vcmc8L3NhbWw6
SXNzdWVyPgogICAgICA8c2FtbHA6U3RhdHVzPgogICAgICAgIDxzYW1scDpTdGF0
dXNDb2RlCiAgICAgICAgICAgIFZhbHVlPSJ1cm46b2FzaXM6bmFtZXM6dGM6U0FN
TDoyLjA6c3RhdHVzOlN1Y2Nlc3MiLz4KICAgICAgPC9zYW1scDpTdGF0dXM+CiAg
ICAgIDxzYW1sOkVuY3J5cHRlZEFzc2VydGlvbj4KICAgICAgICA8IS0tIGNvbnRl
bnRzIGVsaWRlZCwgY29weSBvZiBzYW1sZWM6R2VuZXJhdGVkS2V5IGluIEFkdmlj
ZSAtLT4KICAgICAgPC9zYW1sOkVuY3J5cHRlZEFzc2VydGlvbj4KICAgIDwvc2Ft
bHA6UmVzcG9uc2U+CiAgPC9TOkJvZHk+CjwvUzpFbnZlbG9wZT4K
</response>
```


The Base64 [RFC4648] decoded envelope:

```
<S:Envelope
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <paos:Response xmlns:paos="urn:liberty:paos:2003-08"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next"
      S:mustUnderstand="1" refToMessageID="6c3a4f8b9c2d"/>
    <samlec:SessionKey xmlns:samlec="urn:ietf:params:xml:ns:samlec"
      xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
      S:mustUnderstand="1"
      S:actor="http://schemas.xmlsoap.org/soap/actor/next">
      <samlec:EncType>17</samlec:EncType>
    <samlec:SessionKey>
  </S:Header>
  <S:Body>
    <samlp:Response ID="d43h94r389309r" Version="2.0"
        IssueInstant="2020-12-10T11:42:34Z" InResponseTo="c3a4f8b9c2d"
        Destination="xmpp@xmpp.example.com">
      <saml:Issuer>https://saml.example.org</saml:Issuer>
      <samlp:Status>
        <samlp:StatusCode
            Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
      </samlp:Status>
      <saml:EncryptedAssertion>
        <!-- contents elided, copy of samlec:GeneratedKey in Advice -->
      </saml:EncryptedAssertion>
    </samlp:Response>
  </S:Body>
</S:Envelope>
```

Step 9: Server informs client of successful authentication:

```
<success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

Step 9 (alt): Server informs client of failed authentication:

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
 <temporary-auth-failure/>
</failure>
</stream:stream>
```

Step 10: Client initiates a new stream to server:

```
<stream:stream xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
to='example.com' version='1.0'>
```

Step 11: Server responds by sending a stream header to client along with any additional features (or an empty features element):

```
<stream:stream xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
id='c2s_345' from='example.com' version='1.0'>
<stream:features>
 <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
 <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
</stream:features>
```

Step 12: Client binds a resource:

```
<iq type='set' id='bind_1'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>someresource</resource>
  </bind>
</iq>
```

Step 13: Server informs client of successful resource binding:

```
<iq type='result' id='bind_1'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <jid>somenode@example.com/someresource</jid>
  </bind>
</iq>
```

Please note: line breaks were added to the base64 for clarity.

7. Security Considerations

This section will address only security considerations associated with the use of SAML with SASL applications. For considerations relating to SAML in general, the reader is referred to the SAML specification and to other literature. Similarly, for general SASL Security Considerations, the reader is referred to that specification.

Version 2.0 of the Enhanced Client or Proxy Profile [SAMLECP20] adds optional support for channel binding and use of "Holder of Key" subject confirmation. The former is strongly recommended for use with this mechanism to detect "Man in the Middle" attacks between the client and the RP without relying on the commercial TLS infrastructure that does not provide the level of assurance desired by sensitive SAML applications. The latter may be impractical in many cases, but is a valuable way of strengthening client authentication, protecting against phishing, and improving the overall mechanism.

7.1. Risks Left Unaddressed

The adaptation of a web-based profile that is largely designed around security-oblivious clients and a bearer model for security token validation results in a number of basic security exposures that should be weighed against the compatibility and client simplification benefits of this mechanism.

When channel binding is not used, protection against "Man in the Middle" attacks is left solely to lower layer protocols such as TLS, and the development of user interfaces able to implement that has not been effectively demonstrated. Failure to detect a MITM can result in phishing of the user’s credentials if the attacker is between the client and IdP, or the theft and misuse of a short-lived credential (the SAML assertion) if the attacker is able to impersonate a RP. SAML allows for source address checking as a minor mitigation to the latter threat, but this is often impractical. IdPs can mitigate to some extent the exposure of personal information to RP attackers by encrypting assertions with authenticated keys.

7.2. User Privacy

The IdP is aware of each RP that a user logs into. There is nothing in the protocol to hide this information from the IdP. It is not a requirement to track the activity, but there is nothing technically that prohibits the collection of this information. Servers should be aware that SAML IdPs will track - to some extent - user access to their services. This exposure extends to the use of session keys generated by the IdP to secure messages between the parties, but note that when bearer assertions are involved, the IdP can freely impersonate the user to any relying party in any case.

It is also out of scope of the mechanism to determine under what conditions an IdP will release particular information to a relying party, and it is generally unclear in what fashion user consent could be established in real time for the release of particular

information. The SOAP exchange with the IdP does not preclude such interaction, but neither does it define that interoperably.

7.3. Collusion between RPs

Depending on the information supplied by the IdP, it may be possible for RPs to correlate data that they have collected. By using the same identifier to log into every RP, collusion between RPs is possible. SAML supports the notion of pairwise, or targeted/directed, identity. This allows the IdP to manage opaque, pairwise identifiers for each user that are specific to each RP. However, correlation is often possible based on other attributes supplied, and is generally a topic that is beyond the scope of this mechanism. It is sufficient to say that this mechanism does not introduce new correlation opportunities over and above the use of SAML in web-based use cases.

8. IANA Considerations

8.1. GSS-API and SASL Mechanism Registration

The IANA is requested to assign a new entry for this GSS mechanism in the sub-registry for SMI Security for Mechanism Codes, whose prefix is iso.org.dod.internet.security.mechanisms (1.3.6.1.5.5) and to reference this specification in the registry.

The IANA is requested to register the following SASL profile:

SASL mechanism profiles: SAML20EC and SAML20EC-PLUS

Security Considerations: See this document

Published Specification: See this document

For further information: Contact the authors of this document.

Owner/Change controller: the IETF

Note: None

8.2. XML Namespace Name for SAML-EC

A URN sub-namespace for XML constructs introduced by this mechanism is defined as follows:

URI: urn:ietf:params:xml:ns:samlec

Specification: See Appendix A of this document.

Description: This is the XML namespace name for XML constructs introduced by the SAML Enhanced Client SASL and GSS-API Mechanisms.

Registrant Contact: the IESG

9. References

9.1. Normative References

[OASIS.saml-bindings-2.0-os]
Cantor, S., Hirsch, F., Kemp, J., Philpott, R., and E. Maler, "Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-bindings-2.0-os, March 2005.

[OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.

[OASIS.saml-profiles-2.0-os]
Hughes, J., Cantor, S., Hodges, J., Hirsch, F., Mishra, P., Philpott, R., and E. Maler, "Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard OASIS.saml-profiles-2.0-os, March 2005.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <https://www.rfc-editor.org/info/rfc2743>.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, DOI 10.17487/RFC3961, February 2005, <https://www.rfc-editor.org/info/rfc3961>.

[RFC3962] Raeburn, K., "Advanced Encryption Standard (AES) Encryption for Kerberos 5", RFC 3962, DOI 10.17487/RFC3962, February 2005, <https://www.rfc-editor.org/info/rfc3962>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, DOI 10.17487/RFC4121, July 2005, <https://www.rfc-editor.org/info/rfc4121>.

[RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API Extension for the Generic Security Service Application Program Interface (GSS-API)", RFC 4401, DOI 10.17487/RFC4401, February 2006, <https://www.rfc-editor.org/info/rfc4401>.

[RFC4422] Melnikov, A., Ed. and K. Zeilenga, Ed., "Simple Authentication and Security Layer (SASL)", RFC 4422, DOI 10.17487/RFC4422, June 2006, <https://www.rfc-editor.org/info/rfc4422>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5554] Williams, N., "Clarifications and Extensions to the Generic Security Service Application Program Interface (GSS-API) for the Use of Channel Bindings", RFC 5554, DOI 10.17487/RFC5554, May 2009, <https://www.rfc-editor.org/info/rfc5554>.

[RFC5801] Josefsson, S. and N. Williams, "Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family", RFC 5801, DOI 10.17487/RFC5801, July 2010, <https://www.rfc-editor.org/info/rfc5801>.

[RFC6680] Williams, N., Johansson, L., Hartman, S., and S. Josefsson, "Generic Security Service Application Programming Interface (GSS-API) Naming Extensions", RFC 6680, DOI 10.17487/RFC6680, August 2012, <https://www.rfc-editor.org/info/rfc6680>.

[RFC7056] Hartman, S. and J. Howlett, "Name Attributes for the GSS-API Extensible Authentication Protocol (EAP) Mechanism", RFC 7056, DOI 10.17487/RFC7056, December 2013, <https://www.rfc-editor.org/info/rfc7056>.

[RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <https://www.rfc-editor.org/info/rfc7617>.

[RFC7802] Emery, S. and N. Williams, "A Pseudo-Random Function (PRF) for the Kerberos V Generic Security Service Application Program Interface (GSS-API) Mechanism", RFC 7802, DOI 10.17487/RFC7802, March 2016, <https://www.rfc-editor.org/info/rfc7802>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[SAMLECP20] Cantor, S., "SAML V2.0 Enhanced Client or Proxy Profile Version 2.0", OASIS Committee Specification OASIS.sstc-saml-ecp-v2.0-cs01, August 2013.

[W3C.soap11] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and D. Winer, "Simple Object Access Protocol (SOAP) 1.1", W3C Note soap11, May 2000, <http://www.w3.org/TR/SOAP/>.

9.2. Informative References

[OASIS.saml-metadata-2.0-os] Cantor, S., Moreh, J., Philpott, R., and E. Maler, "Metadata for the Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-metadata-2.0-os, March 2005.

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, DOI 10.17487/RFC1939, May 1996, <https://www.rfc-editor.org/info/rfc1939>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, DOI 10.17487/RFC4559, June 2006, <https://www.rfc-editor.org/info/rfc4559>.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011, <https://www.rfc-editor.org/info/rfc6120>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[W3C.REC-xmlschema-1] Thompson, H., Beech, D., Maloney, M., and N. Mendelsohn, "XML Schema Part 1: Structures", W3C REC-xmlschema-1, May 2001, <http://www.w3.org/TR/xmlschema-1/>.

[WSS-SAML] Monzillo, R., "Web Services Security SAML Token Profile Version 1.1.1", OASIS Standard OASIS.wss-SAMLTokenProfile, May 2012.

Appendix A. XML Schema

The following schema formally defines the "urn:ietf:params:xml:ns:samlec" namespace used in this document, in conformance with [W3C.REC-xmlschema-1] While XML validation is optional, the schema that follows is the normative definition of the constructs it defines. Where the schema differs from any prose in this specification, the schema takes precedence.

```
<schema
  targetNamespace="urn:ietf:params:xml:ns:samlec"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:samlec="urn:ietf:params:xml:ns:samlec"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified"
  blockDefault="substitution"
  version="1.0">

  <import namespace="http://www.w3.org/2000/09/xmldsig#"/>
  <import namespace="http://schemas.xmlsoap.org/soap/envelope/"/>

  <element name="SessionKey" type="samlec:SessionKeyType"/>
  <complexType name="SessionKeyType">
    <sequence>
      <element ref="samlec:EncType" maxOccurs="unbounded"/>
      <element ref="ds:KeyInfo" minOccurs="0"/>
    </sequence>
    <attribute ref="S:mustUnderstand" use="required"/>
    <attribute ref="S:actor" use="required"/>
    <attribute name="Algorithm"/>
  </complexType>

  <element name="EncType" type="integer"/>

  <element name="GeneratedKey" type="samlec:GeneratedKeyType"/>
  <complexType name="GeneratedKeyType">
    <simpleContent>
      <extension base="base64Binary">
        <attribute ref="S:mustUnderstand"/>
        <attribute ref="S:actor"/>
      </extension>
    </simpleContent>
  </complexType>

  <element name="Delegated" type="samlec:DelegatedType"/>
  <complexType name="DelegatedType">
    <sequence/>
    <attribute ref="S:mustUnderstand" use="required"/>
    <attribute ref="S:actor" use="required"/>
  </complexType>

</schema>
```

Appendix B. Acknowledgments

The authors would also like to thank Klaas Wierenga, Sam Hartman, Nico Williams, Jim Basney, Venkat Yekkirala, and Ben Kaduk for their contributions.

Appendix C. Changes

This section to be removed prior to publication.

- o 20, address nits and easy fixes from Ben Kaduk's AD review
- o 19, update obsoleted references
- o 15,16,17,18 avoid expiration
- o 14, address some minor comments
- o 13, clarify SAML metadata usage, adding a recommended Binding value alongside the backward-compatibility usage of PAOS
- o 12, clarifying comments based on WG feedback, with a normative change to use enctype numbers instead of names
- o 11, update EAP Naming reference to RFC
- o 10, update SAML ECP reference to final CS
- o 09, align delegation signaling to updated ECP draft
- o 08, more corrections, added a delegation signaling header
- o 07, corrections, revised section on delegation
- o 06, simplified session key schema, moved responsibility for random-to-key to the endpoints, and defined advertisement of session key algorithm and enctypes by acceptor
- o 05, revised session key material, added requirement for random-to-key, revised XML schema to capture enctype name, updated GSS naming reference
- o 04, stripped down the session key material to simplify it, and define an IdP-brokered keying approach, moved session key XML constructs from OASIS draft into this one
- o 03, added TLS key export as a session key option, revised GSS naming material based on list discussion

- o 02, major revision of GSS-API material and updated references

- o 01, SSH language added, noted non-assumption of HTTP error handling, added guidance on life of security context.

- o 00, Initial Revision, first WG-adopted draft. Removed support for unsolicited SAML responses.

Authors' Addresses

Scott Cantor
Shibboleth Consortium
1050 Carmack Rd
Columbus, Ohio 43210
United States

Phone: +1 614 247 6147
Email: cantor.2@osu.edu

Margaret Cullen
Painless Security
4 High St, Suite 134
North Andover, Massachusets 01845
United States

Phone: +1 781 405 7464
Email: mrcullen42@painless-security.com

Simon Josefsson
SJD AB
Hagagatan 24
Stockholm 113 47
SE

Email: simon@josefsson.org
URI: http://josefsson.org/

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 4, 2014

S. Josefsson
SJD AB

March 3, 2014

Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family

draft-josefsson-kitten-gs2bis-00

Abstract

This document describes how to use a Generic Security Service Application Program Interface (GSS-API) mechanism in the Simple Authentication and Security Layer (SASL) framework. This is done by defining a new SASL mechanism family, called GS2. This mechanism family offers a number of improvements over the previous "SASL/GSSAPI" mechanism: it is more general, uses fewer messages for the authentication phase in some cases, and supports negotiable use of channel binding. This is an update of RFC 5801 that relaxes the requirement for channel binding support and mutual authentication in the underlying GSS-API mechanism.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Generic Security Service Application Program Interface (GSS-API) [RFC2743] is a framework that provides security services to applications using a variety of authentication mechanisms. Simple Authentication and Security Layer (SASL) [RFC4422] is a framework to provide authentication and security layers for connection-based protocols, also using a variety of mechanisms. This document describes how to use a GSS-API mechanism as though it were a SASL mechanism. This facility is called GS2 -- a moniker that indicates that this is the second GSS-API->SASL mechanism bridge. The original GSS-API->SASL mechanism bridge was specified by [RFC2222], now [RFC4752]; we shall sometimes refer to the original bridge as GS1 in this document.

All GSS-API mechanisms are implicitly registered for use within SASL by this specification. The SASL mechanisms defined in this document are known as the GS2 family of mechanisms.

The GS1 bridge failed to gain wide deployment for any GSS-API mechanism other than "The Kerberos Version 5 GSS-API Mechanism" [RFC1964] [RFC4121], and has a number of problems that led us to desire a new bridge. Specifically, a) GS1 was not round-trip optimized and b) GS1 did not support channel binding [RFC5056]. These problems and the opportunity to create the next SASL password-based mechanism, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms" [RFC5802], as a GSS-API mechanism used by SASL applications via GS2, provide the motivation for GS2.

In particular, the current consensus of the SASL community appears to be that SASL "security layers" (i.e., confidentiality and integrity protection of application data after authentication) are too complex and redundant because SASL applications tend to have an option to run over a Transport Layer Security (TLS) [RFC5246] channel. Use of SASL security layers is best replaced with channel binding to a TLS channel.

GS2 is designed to be as simple as possible. It adds to GSS-API security context token exchanges only the bare minimum to support SASL semantics and negotiation of use of channel binding. Specifically, GS2 adds a small header (a few bytes plus the length of the client-requested SASL authorization identity) to the initial GSS-API context token and to the application channel binding data. GS2 uses SASL mechanism negotiation to implement channel binding negotiation. Security-relevant GS2 plaintext is protected via the use of GSS-API channel binding. Additionally, to simplify the implementation of GS2 mechanisms for implementors who will not

implement a GSS-API framework, we compress the initial security context token header required by [RFC2743], Section 3.1.

GS2 does not protect any plaintext exchanged outside GS2, such as SASL mechanism negotiation plaintext, or application messages following authentication. But using channel binding to a secure channel over which all SASL and application plaintext is sent will cause all that plaintext to be authenticated.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The document uses many terms and function names defined in [RFC2743], as updated by [RFC5554].

3. Mechanism Name

There are two SASL mechanism names for any GSS-API mechanism used through this facility. One denotes that the server supports channel binding. The other denotes that it does not.

The SASL mechanism name for a GSS-API mechanism is that which is provided by that mechanism when it was specified, if one was specified. This name denotes that the server does not support channel binding. Add the suffix "-PLUS" and the resulting name denotes that the server does support channel binding. SASL implementations can use the GSS_Inquire_SASLname_for_mech call (see below) to query for the SASL mechanism name of a GSS-API mechanism.

If the GSS_Inquire_SASLname_for_mech interface is not used, the GS2 implementation needs some other mechanism to map mechanism Object Identifiers (OIDs) to SASL names internally. In this case, the implementation can only support the mechanisms for which it knows the SASL name. If GSS_Inquire_SASLname_for_mech() fails and the GS2 implementation cannot map the OID to a SASL mechanism name via some other means, then the GS2 implementation MUST NOT use the given GSS-API mechanism.

3.1. Generating SASL Mechanism Names from GSS-API OIDs

For GSS-API mechanisms whose SASL names are not defined together with the GSS-API mechanism or in this document, the SASL mechanism name is concatenation of the string "GS2-" and the Base32 encoding [RFC4648]

(with an uppercase alphabet) of the first 55 bits of the binary SHA-1 hash [FIPS.180-1.1995] string computed over the ASN.1 DER encoding [CCITT.X690.2002], including the tag and length octets, of the GSS-API mechanism’s Object Identifier. The Base32 rules on padding characters and characters outside of the Base32 alphabet are not relevant to this use of Base32. If any padding or non-alphabet characters are encountered, the name is not a GS2 family mechanism name. This name denotes that the server does not support channel binding. Add the suffix "-PLUS" and the resulting name denotes that the server does support channel binding.

A GS2 mechanism that has a non-OID-derived SASL mechanism name is said to have a "user-friendly SASL mechanism name".

3.2. Computing Mechanism Names Manually

The hash-derived GS2 SASL mechanism name may be computed manually. This is useful when the set of supported GSS-API mechanisms is known in advance. This eliminates the need to implement Base32, SHA-1, and DER in the SASL mechanism. The computed mechanism name can be used directly in the implementation, and the implementation need not be concerned if the mechanism is part of a mechanism family.

3.3. Examples

The OID for the Simple Public-Key GSS-API Mechanism (SPKM-1) [RFC2025] is 1.3.6.1.5.5.1.1. The ASN.1 DER encoding of the OID, including the tag and length, is (in hex) 06 07 2b 06 01 05 05 01 01. The SHA-1 hash of the ASN.1 DER encoding is (in hex) 1c f8 f4 2b 5a 9f 80 fa e9 f8 31 22 6d 5d 9d 56 27 86 61 ad. Convert the first 7 octets to binary, drop the last bit, and re-group them in groups of 5, and convert them back to decimal, which results in these computations:

```
hex:
1c f8 f4 2b 5a 9f 80

binary:
00011100 11111000 11110100 00101011 01011010
10011111 1000000

binary in groups of 5:
00011 10011 11100 01111 01000 01010 11010 11010
10011 11110 00000

decimal of each group:
3 19 28 15 8 10 26 26 19 30 0

base32 encoding:
D T 4 P I K 2 2 T 6 A
```

The last step translates each decimal value using table 3 in Base32 [RFC4648]. Thus, the SASL mechanism name for the SPKM-1 GSSAPI mechanism is "GS2-DT4PIK22T6A".

The OID for the Kerberos V5 GSS-API mechanism [RFC1964] is 1.2.840.113554.1.2.2 and its DER encoding is (in hex) 06 09 2A 86 48 86 F7 12 01 02 02. The SHA-1 hash is 82 d2 73 25 76 6b d6 c8 45 aa 93 25 51 6a fc ff 04 b0 43 60. Convert the 7 octets to binary, drop the last bit, and re-group them in groups of 5, and convert them back to decimal, which results in these computations:

```
hex:
82 d2 73 25 76 6b d6

binary:
10000010 11010010 01110011 00100101 01110110
01101011 1101011

binary in groups of 5:
10000 01011 01001 00111 00110 01001 01011 10110
01101 01111 01011

decimal of each group:
16 11 9 7 6 9 11 22 13 15 11

base32 encoding:
Q L J H G J L W N P L
```

The last step translates each decimal value using table 3 in Base32 [RFC4648]. Thus, the SASL mechanism name for the Kerberos V5 GSS-API mechanism would be "GS2-QLJHGJLWNPL" and (because this mechanism

supports channel binding) "GS2-QLJHGJLWNPL-PLUS". Instead, the next section assigns the Kerberos V5 mechanism a non-hash-derived mechanism name.

3.4. Grandfathered Mechanism Names

Some older GSS-API mechanisms were not specified with a SASL GS2 mechanism name. Using a shorter name can be useful, nonetheless. We specify the names "GS2-KRB5" and "GS2-KRB5-PLUS" for the Kerberos V5 mechanism, to be used as if the original specification documented it, see Section 16.

4. SASL Authentication Exchange Message Format

During the SASL authentication exchange for GS2, a number of messages following the following format are sent between the client and server. On success, this number is the same as the number of context tokens that the GSS-API mechanism would normally require in order to establish a security context. On failures, the exchange can be terminated early by any party.

When using a GS2 mechanism the SASL client is always a GSS-API initiator and the SASL server is always a GSS-API acceptor. The client calls GSS_Init_sec_context and the server calls GSS_Accept_sec_context.

All the SASL authentication messages exchanged are exactly the same as the security context tokens of the GSS-API mechanism, except for the initial security context token.

The client and server MAY send GSS-API error tokens (tokens output by GSS_Init_sec_context() or GSS_Accept_sec_context() when the major status code is other than GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED). As this indicates an error condition, after sending the token, the sending side should fail the authentication.

The initial security context token is modified as follows:

- o The initial context token header (see Section 3.1 of [RFC2743]) MUST be removed if present. If the header is not present, the client MUST send a "gs2-nonstd-flag" flag (see below). On the server side, this header MUST be recomputed and restored prior to passing the token to GSS_Accept_sec_context, except when the "gs2-nonstd-flag" is sent.
- o A GS2 header MUST be prefixed to the resulting initial context token. This header has the form "gs2-header" given below in ABNF [RFC5234].

The figure below describes the permissible attributes, their use, and the format of their values. All attribute names are single US-ASCII letters and are case sensitive.

```
 UTF8-1-safe    = %x01-2B / %x2D-3C / %x3E-7F
                  ;; As UTF8-1 in RFC 3629 except
                  ;; NUL, "=", and ",".
 UTF8-2         = <as defined in RFC 3629 (STD 63)>
 UTF8-3         = <as defined in RFC 3629 (STD 63)>
 UTF8-4         = <as defined in RFC 3629 (STD 63)>
 UTF8-char-safe = UTF8-1-safe / UTF8-2 / UTF8-3 / UTF8-4

 saslname       = 1*(UTF8-char-safe / "=2C" / "=3D")
 gs2-authzid    = "a=" saslname
                    ;; GS2 has to transport an authzid since
                    ;; the GSS-API has no equivalent
 gs2-nonstd-flag = "F"
                    ;; "F" means the mechanism is not a
                    ;; standard GSS-API mechanism in that the
                    ;; RFC 2743, Section 3.1 header was missing
 cb-name         = 1*(ALPHA / DIGIT / "." / "-")
                    ;; See RFC 5056, Section 7.
 gs2-cb-flag     = ("p=" cb-name) / "n" / "y"
                    ;; GS2 channel binding (CB) flag
                    ;; "p" -> client supports and used CB
                    ;; "n" -> client does not support CB
                    ;; "y" -> client supports CB, thinks the server
                    ;;           does not
 gs2-header = [gs2-nonstd-flag ","] gs2-cb-flag "," [gs2-authzid] ","
                      ;; The GS2 header is gs2-header.
```

When the "gs2-nonstd-flag" flag is present, the client did not find/remove a token header ([RFC2743], Section 3.1) from the initial token returned by GSS_Init_sec_context. This signals to the server that it MUST NOT re-add the data that is normally removed by the client.

The "gs2-cb-flag" signals the channel binding mode. One of "p", "n", or "y" is used. A "p" means the client supports and used a channel binding, and the name of the channel binding type is indicated. An "n" means that the client does not support channel binding. A "y" means the client supports channel binding, but believes the server does not support it, so it did not use a channel binding. See the next section for more details.

The "gs2-authzid" holds the SASL authorization identity. It is encoded using UTF-8 [RFC3629] with three exceptions:

- o The NUL character is forbidden as required by section 3.4.1 of [RFC4422].
- o The server MUST replace any "," (comma) in the string with "=2C".
- o The server MUST replace any "=" (equals) in the string with "=3D".

Upon receipt of this value, the server verifies its correctness according to the used SASL protocol profile. Failed verification results in a failed authentication exchange.

5. Channel Bindings

GS2 supports channel binding to external secure channels, such as TLS. Clients and servers may or may not support channel binding; therefore, the use of channel binding is negotiable. However, GS2 does not provide security layers; therefore, it is imperative that GS2 provide integrity protection for the negotiation of channel binding.

Use of channel binding is negotiated as follows:

- o Servers that support the use of channel binding SHOULD advertise both the non-PLUS and PLUS-variant of each GS2 mechanism name. If the server cannot support channel binding, it SHOULD advertise only the non-PLUS-variant. If the server would never succeed in the authentication of the non-PLUS-variant due to policy reasons, it MUST advertise only the PLUS-variant.
- o If the client supports channel binding and the server does not appear to (i.e., the client did not see the -PLUS name advertised by the server), then the client MUST NOT use an "n" gs2-cb-flag.
- o Clients that support mechanism negotiation and channel binding MUST use a "p" gs2-cb-flag when the server offers the PLUS-variant of the desired GS2 mechanism.
- o If the client does not support channel binding, then it MUST use an "n" gs2-cb-flag. Conversely, if the client requires the use of channel binding then it MUST use a "p" gs2-cb-flag. Clients that do not support mechanism negotiation never use a "y" gs2-cb-flag, they use either "p" or "n" according to whether they require and support the use of channel binding or whether they do not, respectively.
- o The client generates the chan_bindings input parameter for GSS_Init_sec_context as described below.
- o Upon receipt of the initial authentication message, the server checks the gs2-cb-flag in the GS2 header and constructs a chan_bindings parameter for GSS_Accept_sec_context as described below. If the client channel binding flag was "y" and the server did advertise support for channel bindings (by advertising the PLUS-variant of the mechanism chosen by the client), then the server MUST fail authentication. If the client channel binding flag was "p" and the server does not support the indicated channel

binding type, then the server MUST fail authentication.

- o If the client used an "n" gs2-cb-flag and the server requires the use of channel binding, then the server MUST fail authentication.

FLAG	CLIENT CB SUPPORT	SERVER CB SUPPORT	DISPOSITION
n	no support	N/A	If server disallows non-channel-bound authentication, then fail
y	Yes, not required	No	Authentication may succeed; CB not used
y	Yes, not required	Yes	Authentication must fail
p	Yes	Yes	Authentication may succeed, with CB used
p	Yes	No	Authentication will fail
N/A	Yes, required	No	Client does not even try

For more discussion of channel bindings, and the syntax of the channel binding data for various security protocols, see [RFC5056].

5.1. Content of GSS-CHANNEL-BINDINGS Structure

The calls to GSS_Init_sec_context and GSS_Accept_sec_context take a chan_bindings parameter. The value is a GSS-CHANNEL-BINDINGS structure [RFC5554].

The initiator-address-type and acceptor-address-type fields of the GSS-CHANNEL-BINDINGS structure MUST be set to 0. The initiator-address and acceptor-address fields MUST be the empty string.

The application-data field MUST be set to the gs2-header, excluding the initial [gs2-nonstd-flag ","] part, concatenated with, when a gs2-cb-flag of "p" is used, the application’s channel binding data.

5.2. Default Channel Binding

A default channel binding type agreement process for all SASL application protocols that do not provide their own channel binding type agreement is provided as follows.

’tls-unique’ is the default channel binding type for any application

that doesn’t specify one.

Servers MUST implement the "tls-unique" [RFC5929] channel binding type, if they implement any channel binding. Clients SHOULD implement the "tls-unique" channel binding type, if they implement any channel binding. Clients and servers SHOULD choose the highest-layer/innermost end-to-end TLS channel as the channel to which to bind.

Servers MUST choose the channel binding type indicated by the client, or fail authentication if they don’t support it.

6. When the mechanism does not support channel binding and/or mutual authentication

Some authentication mechanisms does not offer mutual authentication or is unable to provide channel bindings. This is unfortunate, and usually suggests that the authentication mechanism offers limited authentication functionality. However there are situations when the lack of this functionality can be mitigated with other protection mechanisms, leading to acceptable overall security. Being able to define and use an authentication mechanism as a GSS-API mechanism and then use that GSS-API mechanism in the SASL environment using GS2 has advantages; for example, being able to re-use existing generic GS2 implementations. Further, being able to express all mechanisms that can be expressed as a GSS-API mechanisms as a SASL mechanism (and vice versa) provides design elegance and framework replacability. Therefor, this document relaxes the requirement that the GSS-API mechanism support channel bindings and/or mutual authentication. Implementing and deploying applications that supports those mechanism require some consideration, and this section discuss the relevant areas.

For the discussion it helps to understand what happens with the GS2 bridge when a GSS-API mechanism does not offer channel bindings or mutual authentication. When channel bindings is not supported by the underlying mechanism, GS2 cannot protect its data (essentially: the channel binding flag and the SASL authorization identity). This means that the security of the channel binding mode breaks down and that the other side cannot trust the SASL authorization identity. When mutual authentication is not happening, the client cannot know that it sends its data to the intended server.

It is acceptable to use these mechanisms with GS2 in some situations. For example, if the client uses TLS against a server, and the client verify the server’s certificate properly so that server authentication has occured, then authenticating the client to the

server using a "weak" GSS-API mechanism will technically work. The security properties will not be as good as they would have been if the underlying mechanism supported channel binding or mutual authentication, however they become as good as possible.

This document relaxes the requirements on GSS-API mechanism so that all GSS-API mechanism can be expressed in GS2. For these mechanisms, the "gs2-cb-flag" value MUST always be "n", and the PLUS-variant of the GS2 mechanism name MUST NOT be advertised or negotiated.

The SAML SASL bridge [RFC6595] and the SAML OpenID bridge [RFC6616] are two examples of documents that describe such bridges. These documents did not meet the requirements of the original GS2 bridge, but with the update in this document they are conformant. Note that both documents had discussions describing this aspect and sufficient requirements for safe implementation and deployment.

7. Examples

Example #1: a one round-trip GSS-API context token exchange, no channel binding, optional authzid given.

```
C: Request authentication exchange
S: Empty Challenge
C: n,a=someuser,<initial context token with standard
                 header removed>
S: Send reply context token as is
C: Empty message
S: Outcome of authentication exchange
```

Example #2: a one and one half round-trip GSS-API context token exchange, no channel binding.

```
C: Request authentication exchange
S: Empty Challenge
C: n,,<initial context token with standard
                 header removed>
S: Send reply context token as is
C: Send reply context token as is
S: Outcome of authentication exchange
```

Example #3: a two round-trip GSS-API context token exchange, no channel binding, no standard token header.

```
C: Request authentication exchange
S: Empty Challenge
C: F,n,,<initial context token without
                    standard header>
S: Send reply context token as is
C: Send reply context token as is
S: Send reply context token as is
C: Empty message
S: Outcome of authentication exchange
```

Example #4: using channel binding, optional authzid given.

```
C: Request authentication exchange
S: Empty Challenge
C: p=tls-unique,a=someuser,<initial context token with standard
                        header removed>
S: Send reply context token as is
...
```

Example #5: using channel binding.

```
C: Request authentication exchange
S: Empty Challenge
C: p=tls-unique,,<initial context token with standard
                        header removed>
S: Send reply context token as is
...
```

Example #6: using non-standard channel binding (requires out-of-band negotiation).

```
C: Request authentication exchange
S: Empty Challenge
C: p=tls-server-end-point,,<initial context token with standard
                        header removed>
S: Send reply context token as is
...
```

Example #7: client supports channel bindings but server does not, optional authzid given.

```
C: Request authentication exchange
S: Empty Challenge
C: y,a=someuser,<initial
                 context token with standard header removed>
S: Send reply context token as is
...
```

GSS-API authentication is always initiated by the client. The SASL framework allows either the client or the server to initiate authentication. In GS2, the server will send an initial empty challenge (zero-byte string) if it has not yet received a token from the client. See Section 3 of [RFC4422].

8. Authentication Conditions

Authentication MUST NOT succeed if any one of the following conditions are true:

- If GSS_Init/Accept_sec_context returns anything other than GSS_S_CONTINUE_NEEDED or GSS_S_COMPLETE.
- If the client’s initial GS2 header does not match the ABNF.
- In particular, if the initial character of the client message is anything except "F", "p", "n", or "y".
- If the client’s GS2 channel binding flag was "y" and the server supports channel bindings.
- If the client’s GS2 channel binding flag was "p" and the server does not support the indicated channel binding.
- If the client requires use of channel binding and the server did not advertise support for channel binding.
- If authorization of client principal (i.e., src_name in GSS_Accept_sec_context) to requested authzid failed.
- If the client is not authorized to the requested authzid or an authzid could not be derived from the client’s initiator principal name.

9. GSS-API Parameters

GS2 does not use any GSS-API per-message tokens. Therefore, the per-message token ret_flags from GSS_Init_sec_context() and GSS_Accept_sec_context() are irrelevant; implementations SHOULD NOT set the per-message req_flags.

The mutual_req_flag MUST be set. Clients MUST check that the

corresponding ret_flag is set when the context is fully established, else authentication MUST fail.

Use or non-use of deleg_req_flag and anon_req_flag is an implementation-specific detail. SASL and GS2 implementors are encouraged to provide programming interfaces by which clients may choose to delegate credentials and by which servers may receive them. SASL and GS2 implementors are encouraged to provide programming interfaces that provide a good mapping of GSS-API naming options.

10. Naming

There is no requirement that any particular GSS-API name-types be used. However, typically, SASL servers will have host-based acceptor principal names (see [RFC2743], Section 4.1) and clients will typically have username initiator principal names (see [RFC2743], Section 4.2). When a host-based acceptor principal name is used ("service@hostname"), "service" is the service name specified in the protocol’s profile and "hostname" is the fully qualified host name of the server.

11. GSS_Inquire_SASLname_for_mech Call

We specify a new GSS-API utility function to allow SASL implementations to more efficiently identify the GSS-API mechanism to which a particular SASL mechanism name refers.

Inputs:

o desired_mech OBJECT IDENTIFIER

Outputs:

o major_status INTEGER

o minor_status INTEGER

o sasl_mech_name UTF-8 STRING -- SASL name for this mechanism; caller must release with GSS_Release_buffer()

o mech_name UTF-8 STRING -- name of this mechanism, possibly localized; caller must release with GSS_Release_buffer()

o mech_description UTF-8 STRING -- possibly localized description of this mechanism; caller must release with GSS_Release_buffer()

Return major_status codes:

o GSS_S_COMPLETE indicates successful completion, and that output parameters holds correct information.

o GSS_S_BAD_MECH indicates that a desired_mech was unsupported by the GSS-API implementation.

o GSS_S_FAILURE indicates that the operation failed for reasons unspecified at the GSS-API level.

The GSS_Inquire_SASLname_for_mech call is used to get the SASL mechanism name for a GSS-API mechanism. It also returns a name and description of the mechanism in user-friendly form.

The output variable sasl_mech_name will hold the IANA registered mechanism name for the GSS-API mechanism, or if none is registered, a mechanism name computed from the OID as described in Section 3.1 of this document.

11.1. gss_inquire_saslname_for_mech

The C binding for the GSS_Inquire_SASLname_for_mech call is as follows.

As mentioned in [RFC2744], routines may return GSS_S_FAILURE, indicating an implementation-specific or mechanism-specific error condition, further details of which are reported via the minor_status parameter.

```
OM_uint32 gss_inquire_saslname_for_mech(
  OM_uint32     *minor_status,
  const gss_OID  desired_mech,
  gss_buffer_t   sasl_mech_name,
  gss_buffer_t   mech_name,
  gss_buffer_t   mech_description
);
```

Purpose:

Output the SASL mechanism name of a GSS-API mechanism. It also returns a name and description of the mechanism in a user-friendly form.

Parameters:

```
minor_status      Integer, modify
                  Mechanism-specific status code.

desired_mech      OID, read
                  Identifies the GSS-API mechanism to query.

sasl_mech_name    buffer, character-string, modify, optional
                  Buffer to receive SASL mechanism name.
                  The application must free storage associated
                  with this name after use with a call to
                  gss_release_buffer().

mech_name         buffer, character-string, modify, optional
                  Buffer to receive human-readable mechanism name.
                  The application must free storage associated
                  with this name after use with a call to
                  gss_release_buffer().

mech_description  buffer, character-string, modify, optional
                  Buffer to receive description of mechanism.
                  The application must free storage associated
                  with this name after use with a call to
                  gss_release_buffer().

Function value:   GSS status code:

GSS_S_COMPLETE    Successful completion.

GSS_S_BAD_MECH    The desired_mech OID is unsupported.
```

12. GSS_Inquire_mech_for_SASLname Call

To allow SASL clients to more efficiently identify to which GSS-API mechanism a particular SASL mechanism name refers, we specify a new GSS-API utility function for this purpose.

Inputs:

- sasl_mech_name UTF-8 STRING -- SASL name of mechanism.

Outputs:

- major_status INTEGER
- minor_status INTEGER
- mech_type OBJECT IDENTIFIER -- must be explicit mechanism, and not "default" specifier. Caller should treat as read-only and should not attempt to release.

Return major_status codes:

- GSS_S_COMPLETE indicates successful completion, and that output parameters holds correct information.
- GSS_S_BAD_MECH indicates that no supported GSS-API mechanism had the indicated sasl_mech_name.
- GSS_S_FAILURE indicates that the operation failed for reasons unspecified at the GSS-API level.

The GSS_Inquire_mech_for_SASLname call is used to get the GSS-API mechanism OID associated with a SASL mechanism name.

12.1. gss_inquire_mech_for_saslname

The C binding for the GSS_Inquire_mech_for_SASLname call is as follows.

As mentioned in [RFC2744], routines may return GSS_S_FAILURE, indicating an implementation-specific or mechanism-specific error condition, further details of which are reported via the minor_status parameter.

```
OM_uint32 gss_inquire_mech_for_saslname(
  OM_uint32           *minor_status,
  const gss_buffer_t   sasl_mech_name,
  gss_OID             *mech_type
);
```

Purpose:

Output GSS-API mechanism OID of mechanism associated with given sasl_mech_name.

Parameters:

```
minor_status      Integer, modify
                  Mechanism-specific status code.

sasl_mech_name    buffer, character-string, read
                  Buffer with SASL mechanism name.

mech_type         OID, modify, optional
                  Actual mechanism used.  The OID returned via
                  this parameter will be a pointer to static
                  storage that should be treated as read-only.
                  In particular, the application should not attempt
                  to free it.  Specify NULL if not required.

Function value:   GSS status code:

GSS_S_COMPLETE    Successful completion.

GSS_S_BAD_MECH    There is no GSS-API mechanism known
                  as sasl_mech_name.
```

13. Security Layers

GS2 does not support SASL security layers. Applications that need integrity or confidentiality protection can use either channel

binding to a secure external channel or another SASL mechanism that does provide security layers.

14. Interoperability with the SASL GSSAPI Mechanism

The Kerberos V5 GSS-API [RFC1964] mechanism is currently used in SASL under the name GSSAPI, see [RFC4752]. The Kerberos V5 mechanism may also be used with the GS2 family. This causes an interoperability problem, which is discussed and resolved below.

14.1. The Interoperability Problem

The SASL "GSSAPI" mechanism is not wire compatible with the Kerberos V GSS-API mechanism used as a SASL GS2 mechanism.

If a client (or server) only support Kerberos V5 under the "GSSAPI" name, and the server (or client) only support Kerberos V5 under the GS2 family, the mechanism negotiation will fail.

14.2. Resolving the Problem

If the Kerberos V5 mechanism is supported under GS2 in a server, the server SHOULD also support Kerberos V5 through the "GSSAPI" mechanism, to avoid interoperability problems with older clients.

Reasons for violating this recommendation may include security considerations regarding the absent features in the GS2 mechanism. The SASL "GSSAPI" mechanism lacks support for channel bindings, which means that using an external secure channel may not be sufficient protection against active attackers (see [RFC5056] and [MITM]).

14.3. Additional Recommendations

If the application requires SASL security layers, then it MUST use the SASL "GSSAPI" mechanism [RFC4752] instead of "GS2-KRB5" or "GS2-KRB5-PLUS".

If the application can use channel binding to an external channel, then it is RECOMMENDED that it select Kerberos V5 through the GS2 mechanism rather than the "GSSAPI" mechanism.

15. GSS-API Mechanisms That Negotiate Other Mechanisms

A GSS-API mechanism that negotiates other mechanisms will interact badly with the SASL mechanism negotiation. There are two problems. The first is an interoperability problem and the second is a security

concern. The problems are described and resolved below.

15.1. The Interoperability Problem

If a client implements GSS-API mechanism X, potentially negotiated through a GSS-API mechanism Y, and the server also implements GSS-API mechanism X negotiated through a GSS-API mechanism Z, the authentication negotiation will fail.

15.2. Security Problem

If a client’s policy is to first prefer GSSAPI mechanism X, then non-GSSAPI mechanism Y, then GSSAPI mechanism Z, and if a server supports mechanisms Y and Z but not X, then if the client attempts to negotiate mechanism X by using a GSS-API mechanism that negotiates other mechanisms (such as Simple and Protected GSS-API Negotiation (SPNEGO) [RFC4178]), it may end up using mechanism Z when it ideally should have used mechanism Y. For this reason, the use of GSS-API mechanisms that negotiate other mechanisms is disallowed under GS2.

15.3. Resolving the Problems

GSS-API mechanisms that negotiate other mechanisms MUST NOT be used with the GS2 SASL mechanism. Specifically, SPNEGO [RFC4178] MUST NOT be used as a GS2 mechanism. To make this easier for SASL implementations, we assign a symbolic SASL mechanism name to the SPNEGO GSS-API mechanism, "SPNEGO". SASL client implementations MUST NOT choose the SPNEGO mechanism under any circumstances.

The GSS_C_MA_MECH_NEGO attribute of GSS_Inquire_attrs_for_mech [RFC5587] can be used to identify such mechanisms.

16. IANA Considerations

The IANA has registered a SASL mechanism family as per [RFC4422] using the following information.

```
Subject: Registration of SASL mechanism family GS2-*
SASL mechanism prefix: GS2-
Security considerations: RFC 5801
Published specification: RFC 5801
Person & email address to contact for further information:
  Simon Josefsson <simon@josefsson.org>
Intended usage: COMMON
Owner/Change controller: iesg@ietf.org
Note: Compare with the GSSAPI and GSS-SPNEGO mechanisms.
```

The IANA is advised that SASL mechanism names starting with "GS2-" are reserved for SASL mechanisms that conform to this document. The IANA has placed a statement to that effect in the SASL Mechanisms registry.

The IANA is further advised that GS2 SASL mechanism names MUST NOT end in "-PLUS" except as a version of another mechanism name simply suffixed with "-PLUS".

The SASL names for the Kerberos V5 GSS-API mechanism [RFC4121] [RFC1964] used via GS2 SHALL be "GS2-KRB5" and "GS2-KRB5-PLUS".

The SASL names for the SPNEGO GSS-API mechanism used via GS2 SHALL be "SPNEGO" and "SPNEGO-PLUS". As described in Section 15, the SASL "SPNEGO" and "SPNEGO-PLUS" MUST NOT be used. These names are provided as a convenience for SASL library implementors.

17. Security Considerations

Security issues are also discussed throughout this memo.

The security provided by a GS2 mechanism depends on the security of the GSS-API mechanism. The GS2 mechanism family depends on channel binding support, so GSS-API mechanisms that do not support channel binding cannot be successfully used as SASL mechanisms via the GS2 bridge.

Because GS2 does not support security layers, it is strongly RECOMMENDED that channel binding to a secure external channel be used. Successful channel binding eliminates the possibility of man-in-the-middle (MITM) attacks, provided that the external channel and its channel binding data are secure and that the GSS-API mechanism used is secure. Authentication failure because of channel binding failure may indicate that an MITM attack was attempted, but note that a real MITM attacker would likely attempt to close the connection to the client or simulate network partition; thus, MITM attack detection is heuristic.

Use of channel binding will also protect the SASL mechanism negotiation -- if there is no MITM, then the external secure channel will have protected the SASL mechanism negotiation.

The channel binding data MAY be sent (by the actual GSS-API mechanism used) without confidentiality protection and knowledge of it is assumed to provide no advantage to an MITM (who can, in any case, compute the channel binding data independently). If the external channel does not provide confidentiality protection and the GSS-API

mechanism does not provide confidentiality protection for the channel binding data, then passive attackers (eavesdroppers) can recover the channel binding data, see [RFC5056].

When constructing the input_name_string for GSS_Import_name with the GSS_C_NT_HOSTBASED_SERVICE name type, the client SHOULD NOT canonicalize the server’s fully qualified domain name using an insecure or untrusted directory service, such as the Domain Name System [RFC1034] without DNS Security (DNSSEC) [RFC4033].

SHA-1 is used to derive SASL mechanism names, but no traditional cryptographic properties are required -- the required property is that the truncated output for distinct inputs are different for practical input values. GS2 does not use any other cryptographic algorithm. Therefore, GS2 is "algorithm agile", or, as agile as the GSS-API mechanisms that are available for use in SASL applications via GS2.

GS2 does not protect against downgrade attacks of channel binding types. Negotiation of channel binding type was intentionally left out of scope for this document.

The security considerations of SASL [RFC4422], the GSS-API [RFC2743], channel binding [RFC5056], any external channels (such as TLS, [RFC5246], channel binding types (see the IANA channel binding type registry), and GSS-API mechanisms (such as the Kerberos V5 mechanism [RFC4121] [RFC1964]), also apply.

18. Acknowledgements

The history of GS2 can be traced to the "GSSAPI" mechanism originally specified by RFC 2222. This document was derived from draft-ietf-sasl-gssapi-02 which was prepared by Alexey Melnikov with significant contributions from John G. Myers, although the majority of this document has been rewritten by the current authors.

Contributions of many members of the SASL mailing list are gratefully acknowledged. In particular, ideas and feedback from Pasi Eronen, Sam Hartman, Jeffrey Hutzelman, Alexey Melnikov, and Tom Yu improved the document and the protocol. Other suggestions to the documents were made by Spencer Dawkins, Ralph Droms, Adrian Farrel, Robert Sparks, and Glen Zorn.

19. References

19.1. Normative References

[FIPS.180-1.1995]
National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-1, April 1995, <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5554] Williams, N., "Clarifications and Extensions to the Generic Security Service Application Program Interface (GSS-API) for the Use of Channel Bindings", RFC 5554, May 2009.

[CCITT.X690.2002]
International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

19.2. Informative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

[RFC2025] Adams, C., "The Simple Public-Key GSS-API Mechanism (SPKM)", RFC 2025, October 1996.

[RFC2222] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, January 2000.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

[RFC4178] Zhu, L., Leach, P., Jaganathan, K., and W. Ingersoll, "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism", RFC 4178, October 2005.

[RFC4752] Melnikov, A., "The Kerberos V5 ("GSSAPI") Simple Authentication and Security Layer (SASL) Mechanism", RFC 4752, November 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5587] Williams, N., "Extended Generic Security Service Mechanism Inquiry APIs", RFC 5587, July 2009.

[RFC5802] Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, July 2010.

[RFC6595] Wierenga, K., Lear, E., and S. Josefsson, "A Simple Authentication and Security Layer (SASL) and GSS-API Mechanism for the Security Assertion Markup Language (SAML)", RFC 6595, April 2012.

[RFC6616] Lear, E., Tschofenig, H., Mauldin, H., and S. Josefsson, "A Simple Authentication and Security Layer (SASL) and Generic Security Service Application Program Interface

(GSS-API) Mechanism for OpenID", RFC 6616, May 2012.

[MITM] Asokan, N., Niemi, V., and K. Nyberg, "Man-in-the-Middle in Tunnelled Authentication", WWW http://www.saunalahti.fi/~asokan/research/mitm.html.

Author's Address

Simon Josefsson
SJD AB
Hagagatan 24
Stockholm 113 47
SE

Email: simon@josefsson.org
URI: http://josefsson.org/

Network Working Group
Internet-Draft
Obsoletes: 4757 (if approved)
Updates: 3961 (if approved)
Intended status: Informational
Expires: October 1, 2017

B. Kaduk
Akamai
M. Short
Microsoft Corporation
March 30, 2017

Deprecate 3DES and RC4 in Kerberos
draft-kaduk-kitten-des-des-des-die-die-die-01

Abstract

The 3DES and RC4 encryption types are steadily weakening in cryptographic strength, and the deprecation process should be begun for their use in Kerberos.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The 3DES and RC4 encryption types are steadily weakening in cryptographic strength, and the deprecation process should be begun for their use in Kerberos.

2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Affected Specifications

The RC4 Kerberos encryption types are specified in [RFC4757], which is moved to historic.

The des3-cbc-sha1-kd encryption type is specified in [RFC3961]. Additional 3DES encryption types are in use with no formal specification, in particular des3-cbc-md5 and des3-cbc-sha1. These unspecified encryption types are also deprecated by this document.

4. Affected Encryption Types

The following encryption types are deprecated. The numbers are the official identifiers; the names are only for convenience.

enctype number	enctype convenience name
5	des3-cbc-md5
7	des3-cbc-sha1
16	des3-cbc-sha1-kd
23	rc4-hmac

5. RC4 Weakness

RC4's weakness as a TLS cipher due to statistical biases in the keystream has been well-publicized [RFC7465], and these statistical biases cause concern for any consumer of the RC4 cipher. However, the RC4 Kerberos enctypes have additional flaws which reduce the security of applications using them, including the weakness of the password hashing algorithm, the reuse of key material across protocols, and the lack of a salt when hashing the password.

5.1. Statistical Biases

The RC4 stream cipher is known to have statistical biases in its output, which have led to practical attacks against protocols using RC4, such as TLS ([RFC7465]). These attacks seem to rely on repeated encryptions of thousands of copies of the same plaintext; whereas it is easy for malicious javascript in a website to cause such traffic, it is unclear that there is an easy way to induce a kerberized application to generate such repeated encryptions. The statistical biases are most pronounced for earlier bits in the output stream, which is somewhat mitigated by the use of a confounder in kerberos messages -- the first 64 bits of plaintext are a random confounder, and are thus of no use to an attacker who can retrieve them.

Nonetheless, the statistical biases in the RC4 keystream extend well past 64 bits, and provide potential attack surface to an attacker. Continuing to use a known weak algorithm is inviting further development of attacks.

5.2. Password Hash

Kerberos long-term keys can either be random (as might be used in a service’s keytab) or derived from a password (usable for individual users to authenticate to a system). The specification for a Kerberos encryption type must include a "string2key" algorithm for generating a raw crypto key from a string (i.e., password). Modern encryption types such as those using the AES and Camellia block ciphers use a string2key function based on the PBKDF2 algorithm, which involves many iterations of a cryptographic hash function, designed to increase the computational effort required to perform a brute-force password-guessing attack. There is an additional option to specify an increased iteration count for a given principal, providing some modicum of adaptability for increases in computing power.

It is also best practice when deriving cryptographic secrets from user passwords, to include a value which is unique to both the user and the realm of authentication as input to the has function; this user-specific input is known as a "salt". The default salt for Kerberos principals includes both the name of the principal and the name of the realm, in accordance with these best practices. However, the RC4 encryption types ignore the salt input to the string2key function, which is a single iteration of the MD4 HMAC function applied to the UTF-16 encoded password, with no salt at all. The MD4 hash function is very old, and is considered to be weak and unsuitable for new cryptographic applications at this time. [RFC6150]

The omission of a salt input to the hash is contrary to cryptographic best practices, and allows an attacker to construct a "rainbow table" of password hashes, which are applicable to all principals in all Kerberos realms. Given the prevalance of poor-quality user-selected password, it is likely that a rainbow table derived from a database of common passwords would be able to compromise a sizable number of Kerberos principals in any realm using RC4 encryption types for password-derived keys.

5.3. Cross-Protocol Key Reuse

The selection of unsalted MD4 as the Kerberos string2key function was deliberate, since it allowed systems to be converted in-place from the old NTLM logon protocol [MS-NLMP] to use Kerberos.

Unfortunately, there still exist systems using NTLM for authentication to applications, which can result in application servers possessing the NT password hash of user passwords. Because the RC4 string2key was chosen to be compatible with the NTLM scheme, this means that these application servers also possess the long-term

Kerberos key for those users (even though the password is unknown). The cross-protocol use of the long-term key/password hash was convenient for migrating to Kerberos, but now provides a vulnerability in Kerberos as NTLM continues to be used.

5.4. Interoperability Concerns

The RC4 Kerberos encryption type remains in use in many environments because of interoperability requirements -- in those sites, RC4 is the strongest enctype which allows two parties to use Kerberos to communicate. In particular, the Kerberos implementions included with Windows XP and Windows Server 2003 support only single-DES and RC4. Since single-DES is deprecated ([RFC6649]), machines running those operating systems must use RC4.

Similarly, there are cross-realm situations where the cross-realm key was initially established when one peer only supported RC4, or where machines only supporting RC4 will need to obtain a cross-realm TGT. It can be difficult to inventory all clients in a Kerberos realm and know what implementations will be used by those client principals; this leads to concerns that disabling RC4 will cause breakage on machines that are unknown to the realm administrators.

However, both Windows XP and Windows Server 2003 are already out of their official support periods. It is now believed that all machines that might be broken by disabling RC4 are unsupported, and concerns about breaking them will be reduced. That should facilitate the removal of RC4 from common use.

6. 3DES Weakness

The flaws in triple-DES as used for Kerberos are not quite as damning as those in RC4, but there is still ample justification for deprecating their use. As is the case for the RC4 enctypes, the string2key algorithm is weak. Additionally, the 3DES encryption types were never implemented in all Kerberos implementations, and the 64-bit blocksize may be problematic in some environments.

6.1. Password-based Keys

The string2key function used by the des-cbc-sha1-kd encryption type is essentially just the same n-fold algorithm used by the single-DES family of enctypes. It is known to not provide effective mixing of the input bits, and is computationally easy to evaluate. As such, it does not slow down brute-force attacks in the way that the computationally demanding PBKDF2 algorithm used by more modern encryption types does. The salt is used by des-cbc-sha1-kd’s

string2key, in contrast to RC4, but a brute-force dictionary attack on common passwords may still be feasible.

6.2. Interoperability

The triple-DES encryption types were implemented by MIT Kerberos early in its development, but encryption types 17 and 18 (AES) quickly followed, so there are only a small number of such deployments which support 3DES but not AES. Similarly, the Heimdal Kerberos implementation provided 3DES shortly followed by AES, and has provided AES for nearly ten years.

The Kerberos implementation in Microsoft Windows does not currently and has never implemented the 3DES encryption type. Support for AES was introduced with Windows Vista and Windows Server 2008; older versions such as Windows XP and Windows Server 2003 only supported the RC4 encryption types.

The 3DES encryption type offers very slow encryption, especially compared to the performance of AES using the hardware accelleration available in modern CPUs. There are no areas where it offers advantages over other encryption types except in the rare case where AES is not available.

6.3. Block Size

Because triple-DES is based on the single-DES primitive, just using additional key material and nested encryption, it inherits the 64-bit cipher block size from single-DES. As a result, an attacker who can collect approximately 2**32 blocks of ciphertext has a good chance of finding a cipher block collision (the "birthday attack"), which would potentially reveal a couple blocks of plaintext.

A cipher block collision would not necessarily cause the key itself to be leaked, so the plaintext revealed by such a collision would be limited. For some sites, that may be an acceptable risk, but it is still considered a weakness in the encryption type.

7. Recommendations

This document hereby removes the following RECOMMENDED types from [RFC4120]:

Encryption: DES3-CBC-SHA1-KD

Checksum: HMAC-SHA1-DES3-KD

Kerberos implementations and deployments SHOULD NOT implement or deploy the following triple-DES encryption types: DES3-CBC-MD5(5), DES3-CBC-SHA1(7), and DES3-CBC-SHA1-KD(16) (updates [RFC4120]).

Kerberos implementations and deployments SHOULD NOT implement or deploy the RC4 encryption type RC4-HMAC(23).

Kerberos implementations and deployments SHOULD NOT implement or deploy the following checksum types: RSA-MD5(7), RSA-MD5-DES3(9), HMAC-SHA1-DES3-KD(12), and HMAC-SHA1-DES3(13) (updates [RFC4120]).

Kerberos GSS mechanism implementations and deployments SHOULD NOT implement or deploy the following SGN_ALGs: HMAC MD5(1100) and HMAC SHA1 DES3 KD (updates [RFC4757]).

Kerberos GSS mechanism implementations and deployments SHOULD NOT implement or deploy the following SEAL_ALGs: RC4(1000) and DES3KD(0400).

This document recommends the reclassification of [RFC4757] as Historic.

8. Security Considerations

This document is entirely about security considerations, namely that the use of the 3DES and RC4 Kerberos encryption types is not secure, and they should not be used.

9. IANA Considerations

IANA is requested to update the registry of Kerberos Encryption Type Numbers to note that encryption types 1, 2, 3, and 24 are deprecated, with RFC 6649 ([RFC6649]) as the reference, and that encryption types 5, 7, 16, and 23 are deprecated, with this document as the reference.

Similarly, IANA Is requested to update the registry of Kerberos Checksum Type Numbers to note that checksum types 1, 2, 3, 4, 5, 6, and 8 are deprecated, with RFC 6649 as the reference, and that checksum types 7, 12, and 13 are deprecated, with this document as the reference.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, DOI 10.17487/RFC3961, February 2005, <http://www.rfc-editor.org/info/rfc3961>.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC6150] Turner, S. and L. Chen, "MD4 to Historic Status", RFC 6150, DOI 10.17487/RFC6150, March 2011, <http://www.rfc-editor.org/info/rfc6150>.

10.2. Informative References

[RFC4757] Jaganathan, K., Zhu, L., and J. Brezak, "The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows", RFC 4757, DOI 10.17487/RFC4757, December 2006, <http://www.rfc-editor.org/info/rfc4757>.

[RFC6649] Hornquist Astrand, L. and T. Yu, "Deprecate DES, RC4-HMAC-EXP, and Other Weak Cryptographic Algorithms in Kerberos", BCP 179, RFC 6649, DOI 10.17487/RFC6649, July 2012, <http://www.rfc-editor.org/info/rfc6649>.

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <http://www.rfc-editor.org/info/rfc7465>.

[MS-NLMP] Microsoft Corporation, "[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol", May 2014.

Appendix A. Acknowledgements

Many people have contributed to the understanding of the weaknesses of these encryption types over the years, and they cannot all be named here.

Authors' Addresses

Benjamin Kaduk
Akamai Technologies

Email: kaduk@mit.edu

Michiko Short
Microsoft Corporation

Email: michikos@microsoft.com

Network Working Group
Internet-Draft
Updates: 1964, 2743, 2744
(if approved)
Intended status: Standards Track
Expires: April 30, 2015

N. Williams
Cryptonector

October 27, 2014

Generic Naming Attributes for the Generic Security Services Application Programming Interface (GSS-API)
draft-williams-kitten-generic-naming-attributes-02

Abstract

This document specifies several useful generic naming attributes for use with the Generic Security Services Application Programming Interface (GSS-API) Naming Extensions specified in RFC6680.

These attributes allow applications to extract discrete components of a GSS-API "mechanism name" (MN) object: issuer (e.g., realm name, domain name, certification authority name), service and host names (for host-based service names), user names, and others.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction and Motivation

The Generic Security Services Application Programming Interface (GSS-API) [RFC2743] allows applications -and application protocol specifications- to use various security mechanisms in a generic way. There are some shortcomings of this API that preclude a fully-generic treatment of security mechanisms. This document builds on the naming extensions to the GSS-API [RFC6680] to correct some of those shortcomings.

In RFC6680 we introduced an interface by which to access "attributes" of names, but we did not specify any attributes. This document specifies some such attributes. Some of the new attributes are specifically intended to make it possible to use the GSS-API in a mechanism-generic way in common use cases where it is otherwise not possible to do so.

For example, some applications need to be able to observe the discrete elements of a peer principal’s host-based service name, but they generally could only do so by parsing mechanism-specific display syntaxes or exported name token formats. Such applications are inherently not generic: they can only function correctly when used with security mechanism whose principal naming conventions/formats the applications understand.

More generally, we use the the extended naming interface to introduce an attribute model of principal naming.

1.1. Naming Constraints

This document also introduces a notion of naming constraints, not unlike PKIX’s [RFC5280]. Naming constraints apply to "issuers" of principal names and/or their attributes. For example, to Kerberos [RFC4120] realms, to PKIX certification authorities, to identity providers (IdPs), and so on. The goal is allow specification of policies which constrain the set of principal names that a given issuer can issue credentials for.

For example, the Kerberos realm FOO.EXAMPLE would generally not be expected to issue credentials to host-based principals in domains other than "foo.example".

For each concrete attribute specified below there are several ways to inquire a NAME’s value for that attribute:

1. with naming constraint checking, providing no output if naming constraints are violated;

2. with naming constraint checking, providing an output indicator of naming constraint violations;

3. without naming constraint checking;

4. any of the above with "fast" (no slow I/O involved) naming constraint checking.

(1) is the default behavior. The others are obtained by adding an appropriate prefix to the attribute name.

Existing security mechanisms may not have any formal notion of naming constraints, but it is common to have some naming constraint conventions nonetheless. For example, Kerberos realm naming conventions are that realm names should mirror Domain Name System (DNS) [RFC1035] domain names, and that hostnames embedded in Kerberos principal names should a) be fully-qualified, b) within the domain corresponding to the DNS domain name derived from the realm’s name. Or a Kerberos implementation might lookup a host’s realm and check that it matches the principal’s realm. Naming constraints should be formalized for all GSS-API security mechanisms.

1.2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Generic Attributes

We add a number of generic name attributes, to be used via the GSS-API extended naming facility [RFC6680]. Some of these attributes can be used as prefixes of other attributes, that is, they can be used to modify the semantics of other attributes (see section 6 of RFC6680).

We also provide C bindings for these attributes, namely, the same symbolic names that we provide for the generic attributes.

Note: in all cases the display form of each attribute SHALL consist of text using the character set, codeset, and encoding from the caller’s locale.

2.1. Concrete Attributes

These attributes generally have a single value each. Only one of these attributes can also be used a prefix: the issuer name attribute.

2.1.1. Issuer Name

We add an attribute by which to obtain a name of an issuer of a mechanism name (MN) or of an attribute of an MN. The API name for this attribute is GSS_C_ATTR_GENERIC_ISSUERNAME, and it’s actual attribute name is "urn:ietf:id:ietf-kitten-name-attrs-00-issuername".

The display form of issuer names is mechanism-specific.

The non-display form of issuer names SHALL be the exported name token form of the issuer’s name. Not all mechanisms will support issuer names as MNs, therefore implementations MAY output a null non-display value.

For example, for the Kerberos mechanism [RFC4121] an issuer name would generally (but not always!) be a Kerberos realm name, probably displayed as just the realm name. (But note that there is not yet a Kerberos realm name as MN specification. We will specify one separately.)

This attribute can be used as prefix of other attributes. When used as a prefix, this attribute indicates that the application wishes to know the name of the issuer of the prefixed attribute of the given MN.

2.1.2. Trust Validation Path

We add an attribute by which to obtain the trust validation path for a given authenticated MN. The API for this attribute is GSS_C_ATTR_GENERIC_TRUST_PATH, and its actual attribute name is urn:ietf:id:ietf-kitten-name-attrs-01-trust-path".

This attribute has zero or more ordered values. The interpretation of the trust validation path will vary somewhat by mechanism. For PKIX-based mechanisms this is the list of issuers in the trust validation path for the given MN’s cert. For Kerberos this is the list of realms traversed from the MN to the local name of a security context. The MN’s immediate issuer is not included. In the case of Kerberos, the issuer of the local MN is also not included. For Kerberos the trust validation path is the realm transit path of the Ticket used to establish a security context, but may also include PKIX trust validation paths (e.g., if PKINIT is used).

The display and non-display forms of trust validation path values is as for issuer names; see Section 2.1.1.

2.1.3. User Name

We add an attribute by which to obtain the component of an MN naming a user. The API name for this attribute is GSS_C_ATTR_GENERIC_USERNAME, and it’s actual attribute name is "urn:ietf:id:ietf-kitten-name-attrs-00-username".

The display form of user names is mechanism-specific.

The non-display form of user names is mechanism-specific.

2.1.4. Service Name

We add an attribute by which to obtain the component of an MN naming a service as part of a host- or domain-based service name. The API name for this attribute is GSS_C_ATTR_GENERIC_SERVICENAME, and it’s actual attribute name is "urn:ietf:id:ietf-kitten-name-attrs-00-servicename".

The non-display form of the service name SHALL be the UTF-8 encoding of the service name.

2.1.5. Host Name

We add an attribute by which to obtain the component of an MN naming a host as part of a host- or domain-based service name. The API name for this attribute is GSS_C_ATTR_GENERIC_HOSTNAME, and it’s actual

attribute name is "urn:ietf:id:ietf-kitten-name-attrs-00-hostname".

The display form of a host name MAY be stylized and SHOULD NOT be A-labels. [RFC5890].

The non-display form of host names SHOULD be a character string as described in [RFC1123], and SHOULD NOT be U-labels [RFC5890].

2.1.6. Domain Name

We add an attribute by which to obtain the component of an MN naming a domain as part of a domain-based service name. The API name for this attribute is GSS_C_ATTR_GENERIC_DOMAINNAME, and it’s actual attribute name is "urn:ietf:id:ietf-kitten-name-attrs-00-domainname".

The display form of a domain name MAY be stylized and SHOULD NOT be A-labels. [RFC5890].

The non-display form of domain names SHOULD be a character string as described in [RFC1123], and SHOULD NOT be U-labels [RFC5890].

2.2. Prefix Attributes

GSS_Get_name_attribute() using attributes described in the preceding section SHALL fail if there are any name constraints that can be applied to the issuers of those names and, in applying those constraints, it is discovered that the issuer was not permitted to issue credentials for the MN.

For example, a Kerberos realm named "FOO.EXAMPLE" might not be expected to issue credentials (tickets, keys) to host-based service names for hosts not ending in ".foo.example" or which are not "foo.example".

Several generic attribute prefixes are described below for overriding this behavior.

2.2.1. GSS_C_ATTR_GENERIC_UNCONSTRAINED

This attribute prefix, named GSS_C_ATTR_GENERIC_UNCONSTRAINED in the API, and with an actual name of "urn:ietf:id:ietf-kitten-name-attrs-00-gen-unconstrained", indicates that the application wants the value of the prefixed attribute without any name constraint checking.

2.2.2. GSS_C_ATTR_GENERIC_UNCONSTRAINED_OK

This attribute prefix, named GSS_C_ATTR_GENERIC_UNCONSTRAINED_OK in the API, and with an actual name of "urn:ietf:id:ietf-kitten-name-attrs-00-gen-unconstrained-ok", indicates that the application wants the value of the prefixed attribute regardless of any applicable naming constraints, but to indicate the name constraint status via the 'authenticated' output parameter of the GSS_Get_name_attribute() interface.

2.2.3. GSS_C_ATTR_GENERIC_FAST

This attribute prefix, named GSS_C_ATTR_GENERIC_FAST in the API, and with an actual name of "urn:ietf:id:ietf-kitten-name-attrs-00-gen-fast", indicates that the application requires that the mechanism not perform any slow operations (e.g., connecting to a directory for the purposes of name constraint validation) in obtaining the prefixed attribute of the given MN.

3. Local Name Attributes

Normally an Internet specification would not be expected to specify any local name attributes of GSS names. However, there is one common and very useful local name attribute, which we specify below. Implementations are free to use different names for this attribute or exclude it altogether -- it is a local name attribute, after all.

3.1. GSS_C_ATTR_LOCAL_LOGIN_USER

This attribute, with suggested API symbolic name GSS_C_ATTR_LOCAL_LOGIN_USER, and suggested actual name "local-login-user", requests a local user name corresponding to the given MN, if any.

Obtaining the local user name corresponding to an MN may require complex name mapping or lookup operations that are completely implementation-defined.

4. Suggested Mechanism-Specific Name Attributes (INFORMATIONAL)

[[anchor1: This section should really be split out into separate Internet-Drafts. It is here only because the author lacks the time at the moment of writing to create such separate I-Ds.]]

[[anchor2: Actually, we should probably make this section normative. It's easier than publishing a larger number of RFCs...]]

4.1. Suggested Kerberos-Specific Name Attributes

- o realm (corresponding to issuer name)
- o component 0 (first component of a principal name)
- o component 1 (second component of a principal name)
- o ..
- o component 9 (tenth component of a principal name; ten is enough)
- o components (ordered set of all components of a principal name)
- o specific authorization data elements
- o PKINIT client certificate
- o session key enctype
- o enctypes involved in transit path (this would only be available to initiators)

4.1.1. Kerberos Transit Path Constraint Semantics

For initiator MNs obtained by acceptors from established security contexts, the trust path SHALL be the uncompressed domain- and X.500-style realm names from the initiator's Ticket's 'transited' field, plus the issuer names from the AD-INITIAL-VERIFIED-CAS authorization-data element (if it's in an AD-KDC-ISSUED or similar) if PKINIT [RFC4556] was used.

For acceptor MNs obtained by initiators from established security contexts, the trust path SHALL be the realms traversed -including realms issuing referrals- to obtain a service ticket for the target acceptor.

For MNs for the local end of a security context, the trust path SHALL be empty. This means that GSS_Get_name_attribute() will return empty

value sets; for the C bindings the gss_get_name_attribute() function will return zero in the ’more’ output parameter and empty values.

For initiator MNs as seen by acceptors, if the initiator’s Ticket has the TRANSIT-POLICY-CHECKED flag set, and if local transit path policy is missing, then the GSS_C_ATTR_GENERIC_TRUST_PATH attribute will be considered authenticated -- the trust path will be considered to meet constraints.

Otherwise, if the acceptor has local transit path policy then the GSS_C_ATTR_GENERIC_TRUST_PATH attribute will be considered authenticated -- the trust path will be considered to meet constraints.

In all other cases the GSS_C_ATTR_GENERIC_TRUST_PATH attribute will be considered not authenticated.

4.2. Suggested PKU2U-Specific Name Attributes

[[anchor3: Add reference to PKU2U.]]

- o issuer CA name
- o certificate trust validation path to a trust anchor
- o certificate
- o certificate subject public key
- o certificate subject public key algorithm
- o certificate subject name
- o certificate subject alternate names
- o specific certificate extensions
- o certificate algorithm names
- o session key enctype

5. Generic Issuer Name Type

We add a GSS name-type for use in representing issuer names, designated symbolically as GSS_C_NT_ISSUER. Its query syntax is unspecified and mechanism-specific.

At least initially the common use of this name-type will be for representation of issuer names using the GSS_C_ATTR_GENERIC_ISSUERNAME GSS name attribute (see Section 2.1.1).

5.1. Kerberos Realm Name Type

No name-type is needed in the Kerberos protocol for realm names. Because all three forms of Kerberos realm-names (DOMAIN, X.500, and OTHER) and unambiguously distinguishable from each other, we also do not add a Kerberos-specific GSS name-type.

The query and display syntax of GSS_C_NT_ISSUER names for Kerberos is just a realm name prefixed with an ’@’. We prefix the realm name with ’@’ to take advantage of an otherwise useless ambiguity in the query and display form of Kerberos mechanism principal names [RFC1964], namely that zero-component, and one-zero-length component principal names display identically, therefore those are useless name forms in Kerberos (they would be useless anyways); we appropriate this otherwise useless name form as the query and display syntax of Kerberos realm names. For example, "@FOO.EXAMPLE".

In the unlikely event that a name of GSS_C_NT_ISSUER type is used as a GSS initiator or acceptor principal, the actual Kerberos principal name should be an appropriate TGS principal name. More specific information for such use-cases will be provided by any future application protocol specifications that use them.

5.2. PKIX Issuer Name Type

[[anchor4: A name type for PKIX issuers is needed, even when dealing with Kerberos, since X.500-style realm names may be involved, as well as real PKIX CA names from PKINIT/PKCROSS. We’ll need a mechanism OID for a generic PKIX mechanism (even if it isn’t specified!) for the exported name tokens! One that Kerberos, PKU2U and other PKIX mechanisms can share.]]

6. Security Considerations

[Add text regarding name constraint checking and explaining the default-to-safe design of the generic name attributes defined in section 2.]

7. IANA Considerations

[Add text regarding the registration and assignment of the name attributes described in the preceding sections. In particular we should want these attributes' names to not reflect an Internet-Draft name, but an RFC number.]

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

[RFC1123] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC4556] Zhu, L. and B. Tung, "Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 4556, June 2006.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.

[RFC6680] Williams, N., Johansson, L., Hartman, S., and S. Josefsson, "Generic Security Service Application Programming Interface (GSS-API) Naming Extensions", RFC 6680, August 2012.

8.2. Informative References

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

Author’s Address

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com

Network Working Group
Internet-Draft
Updates: 4121 (if approved)
Intended status: Standards Track
Expires: May 25, 2015

N. Williams
Cryptonector
R. Dowdeswell
Dowdeswell Security Architecture

November 21, 2014

Negotiation of Extra Security Context Tokens for Kerberos V5 Generic Security Services Mechanism
draft-williams-kitten-krb5-extra-rt-04

Abstract

This Internet-Draft proposes an extension to the Kerberos V5 security mechanism for the Generic Security Services Application Programming Interface (GSS-API) for using extra security context tokens in order to recover from certain errors. Other benefits include: user-to-user authentication, authenticated errors, replay cache avoidance, and others.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The Kerberos V5 [RFC4120] AP protocol, and therefore the Kerberos V5 GSS-API [RFC2743] mechanism [RFC4121] security context token exchange, is a one-round trip protocol. Occasionally there are errors that the protocol could recover from by using an additional round trip, but until now there was no way to execute such an additional round trip. For many application protocols the failure of the Kerberos AP protocol is fatal, requiring closing TCP connections and starting over; often there is no automatic recovery.

This document proposes a negotiation of additional security context tokens for automatic recovery from certain errors. This is done in a backwards-compatible way, thus retaining the existing mechanism OID for the Kerberos V5 GSS mechanism. This also enables other new features.

New features enabled by this extension include:

- error recovery (see Section 5)
- user-to-user authentication (see Section 7)
- some authenticated errors (see Section 5.1)
- replay cache avoidance (see Section 6)
- acceptor clock skew correction (see Section 8)
- symmetric authorization data flows

No new interfaces are needed for GSS-API applications to use the features added in this document.

1.1. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. New Protocol Elements

We introduce the following new protocol elements. A partial ASN.1 [CCITT.X680.2002] module (for inclusion in the base Kerberos ASN.1 module) is given in Section 4, and references to its contents are made below.

- o a new ap-options flag for use in the clear-text part of AP-REQs to indicate the desire for an extra round trip if need be;

- o a new authorization data (AD) element for integrity protection of ap-options;

- o a new AD element for use in Authenticators for quoting back a challenge from the acceptor;

- o a new PDU: KRB-ERROR2, also known as AP-REP2, with additional fields and support for integrity- (and confidentiality-)protected errors and optional _key confirmation_ :

  - * a flag is used to indicate which key is used to encrypt the KRB-ERROR2's private part, as in some cases there can be two keys to choose from;

  - * when no key available for encrypting the private part of a KRB-ERROR2, the null enctype is used.

These elements are used to construct security context token exchanges with potentially more than two context tokens.

All context tokens are to be prefixed with the InitialContextToken pseudo-ASN.1/DER header from RFC2743, section 3.1, just as RFCs 1964 and 4121 require of the first two context tokens.

2.1. Fields of KRB-ERROR2

The new KRB-ERROR2 PDU is defined in Section 4. The fields of the KRB-ERROR2 encrypted part have the following purpose/semantics:

continue-challenge A challenge to be quoted back in any subsequent context tokens.

stime The acceptor's current time.

susec Microsecond portion of the acceptor's current time.

subkey The acceptor’s sub-session key. This MUST be absent when the KRB-ERROR2 enc-part is "encrypted" in the null enctype and key or when the acceptor failed to decrypt the initiator’s Authenticator (but, obviously, succeeded at decrypting the Ticket); otherwise it MUST be present.

seq-number The acceptor’s initial per-message token sequence number. This MUST be absent when the subkey is absent; otherwise it MUST be present.

error-code When zero-valued, the KRB-ERROR2 is not an error token, but a key-confirmation that requires continuation with an additional AP-REQ.

e-flags Indicates whether the KRB-ERROR2 is final (error token) or not.

e-text A human-readable string (in any language and script) description of the error, if any.

e-data Currently unused but specified for extensibility reasons. SHOULD be absent and MUST be ignored.

e-typed-data TYPED-DATA; see [RFC4120]. Currently unused but specified for extensibility reasons. SHOULD be absent and MUST be ignored.

your-addresses The initiator’s network address(es) as seen on the acceptor side. Currently unused due to insufficient GSS-API interfaces, but specified for extensibility reasons. SHOULD be absent, MUST be ignored.

ad-data Authorization-data. This is intended for symmetry, so that acceptors can assert authorization data to the initiator just as the initiator can assert authorization data to the acceptor. (For example, this might be useful in user-to-user authentication.) When present this has the same semantics as in the AP-REQ’s Authenticator, but in the opposite direction.

tgt A TGT for use in user-to-user authentication.

2.2. Distinction between KRB-ERROR2 and AP-REP2 PDUs

The ASN.1 does not distinguish between KRB-ERROR2 and AP-REP2 PDUs. A KRB-ERROR2 can serve either or both, the purpose of conveying error information, as well as the purpose of completing the acceptor’s side of the context token exchange and providing key confirmation. We could have used three distinct PDUs instead of one.

It is true that a KRB-ERROR2 that only serves the purpose of final key confirmation without continuation could have a different ASN.1 type for its encrypted part, and a different application tag, however, there seems to be little value in this. Distinguishing between errors with and without key confirmation is even less valuable. Therefore we do not distinguish these three possible PDUs.

3. Negotiation and Use of Extra Context Tokens

In the following text "initiator" refers to the mechanism's initiator functionality (invoked via GSS_Init_sec_context()), and "acceptor" refers to the mechanism's acceptor functionality (invoked via GSS_Accept_sec_context()).

To use this feature, the Kerberos GSS mechanism MUST act as follows:

- o To request this feature, initiators SHALL add the new ap-options flag to their AP-REQs.
  - * And the initiators SHALL repeat the ap-options in the new AD-AP-OPTIONS AD type in the Authenticator.
- o Acceptors that wish to request an additional security context token can only do so when initiators indicate support for it, and MUST do so by returning a KRB-ERROR2. The encrypted part of the KRB-ERROR2 SHALL be encrypted in a key derived (with key usage <TBD>) from one of the following keys: the sub-session key from the AP-REQ's Authenticator (use-initiator-subkey) if it could be decrypted, else the session key from the Ticket (use-ticket-session-key), if it could be decrypted, else the null enc-type/key (use-null-enctype).
- o Any KRB-ERROR2 emitted by the acceptor SHALL have the continue-needed e-flag set when the GSS_Accept_sec_context() returns GSS_S_CONTINUE_NEEDED to the application, and in this case the token ID SHALL be 02 00 (KRB_AP_REP, even though the token isn't actually an AP-REP) (see [RFC4121] section 4.1).
- o When it consumes a KRB-ERROR2, GSS_Init_sec_context() can return an error (GSS_S_FAILURE) and optionally output an error token, or it can attempt recovery (see Section 5) and output a new AP-REQ security context token.
  - * Any error token output by GSS_Init_sec_context() MUST be a KRB-ERROR2, and GSS_Init_sec_context() MUST return GSS_S_FAILURE.
  - * The initiator MUST quote the challenge from the KRB-ERROR2 using an AD-CONTINUE-CHALLENGE (see below) authorization data element in any AP-REQ or KRB-ERROR2 response to the acceptor's KRB-ERROR2.
  - * When GSS_Init_sec_context() outputs a new AP-REQ security context token, it SHALL return GSS_S_CONTINUE_NEEDED if the application requested mutual authentication and the previous acceptor security context token was a recoverable error (rather

than a request for one more AP-REQ), else it SHALL return GSS_S_COMPLETE.

* When GSS_Init_sec_context() returns an error and the acceptor is awaiting a security context token, GSS_Init_sec_context() MAY generate a KRB-ERROR2 or KRB-ERROR to send to the acceptor.

- o Acceptors MUST reject additional AP-REQs which do not have a challenge response nonce matching the one sent by the acceptor in the previous KRB-ERROR2.

- o Acceptors MUST reject initial security context tokens that contain a challenge response nonce.

- o When GSS_Accept_sec_context() returns an error and outputs an error token, the token MUST be either a KRB-ERROR or a KRB-ERROR2, with the latter having the continue-needed flag cleared.

All non-recoverable KRB-ERROR2 tokens SHALL use the token ID 03 00.

Additional AP-REQs produced by the authenticator MUST have the mutual-required ap-options flag set when a) the application requested mutual authentication, and b) the acceptor's KRB-ERROR2 did not supply the required key confirmation. The acceptor MUST respond to the client's last AP-REQ with an AP-REP when the mutual-required ap-options flag is set or when the GSS_C_MUTUAL_FLAG is set in the "checksum 0x8003", otherwise GSS_Accept_sec_context() MUST NOT produce a response token when it returns GSS_S_COMPLETE.

3.1. Number of Security Context Tokens

The first AP-REQ may well result in an error; the second generally should not. Therefore acceptors SHOULD return a fatal error when a second error results in one security context establishment attempt, except when the first error is that the initiator should use user-to-user authentication. This limits the maximum number of round trips to two (not user-to-user) or three (user-to-user).

The mechanism SHOULD impose some limit on the maximum number of security context tokens. For the time being that limit is six.

Note that in the user-to-user cases (see Section 7) it's possible to have up to three round trips under normal conditions if, for example, the acceptor wishes to avoid the use of replay caches (see Section 6), or if the initiator's clock is too skewed, for example.

3.2. Possible Context Token Sequences

The following successful security context token exchange sequences are possible:

- One token (per-RFC4121; mutual authentication not requested): AP-REQ.
  - In principle this can yield an error token in the case of errors, per-RFC2743.
- Two tokens (per-RFC4121; mutual authentication requested): AP-REQ and AP-REP.
- Two tokens (per-RFC4121; mutual authentication requested): AP-REQ and KRB-ERROR.
- Two tokens (per-RFC4121; mutual authentication requested): AP-REQ and KRB-ERROR2 (non-recoverable error, or recoverable error but the acceptor mechanism is configured to not continue).
- Two tokens (per-RFC4121; mutual authentication requested): AP-REQ and KRB-ERROR2 (recoverable error for the acceptor, but not for the initiator, or the initiator application abandons the partially-established security context).
- Three tokens: AP-REQ, KRB-ERROR2 (recoverable error), AP-REQ.
  - The initiator indicates it supports multiple round trips, and a recoverable error results on the acceptor side.
  - Either the initiator did not request mutual authentication, or the KRB-ERROR2 supplied the necessary key confirmation.
- Three tokens: AP-REQ, KRB-ERROR2 (no error, continue needed), AP-REQ.
  - The initiator indicates it supports multiple round trips, and its Authenticator and Ticket decrypt correctly on the acceptor side, but the acceptor wants to continue, e.g., to avoid the need for a replay cache (see Section 6).
  - This can happen in any recoverable error case where the initiator’s Authenticator (and Ticket) decrypt successfully on the acceptor side.

- o Four tokens: AP-REQ, KRB-ERROR2 (recoverable error), AP-REQ, AP-REP.
  - * The initiator wanted mutual authentication and a recoverable error occurred where the KRB-ERROR2 could not provide key confirmation, leading to the second round trip.
  - * This can happen in any recoverable error case where the initiator's Authenticator did not decrypt successfully.
  - * This can also happen in the user-to-user case.
  - * This case provides replay cache avoidance without a fifth token because the acceptor provides a challenge in its first (KRB-ERROR2) token and the initiator completes the challenges in its second token.
- o Five tokens: AP-REQ, KRB-ERROR2 (with user-to-user TGT), AP-REQ, KRB-ERROR2 (recoverable error), AP-REQ.
  - * The initiator does not want mutual authentication, the acceptor wants user-to-user authentication, and the initiator's second AP-REQ elicits a recoverable error.
- o Six tokens: AP-REQ, KRB-ERROR2 (with user-to-user TGT), AP-REQ, KRB-ERROR2 (recoverable error), AP-REQ, AP-REP.
  - * The initiator wants mutual authentication, the acceptor wants user-to-user authentication, and the initiator's second AP-REQ elicits a recoverable error; none of the KRB-ERROR2 tokens was a key-confirmation token.

Other context token sequences might be possible in the future.

In the above sequences the AP-REP tokens can be AP-REP2 tokens as well.

3.3. Per-Message Token Sequence Numbers

It is REQUIRED that each real AP-REQ in a single security token exchange specify the same start sequence number as preceding AP-REQs in the same security context token exchange.

3.4. Early PROT_READY State

The GSS-API allows security mechanisms to support the use of per-message tokens prior to full security context establishment. In this section we'll call this "early PROT_READY". Early PROT_READY is

optional for the GSS-API and for implementations of mechanisms that support it.

The Kerberos V GSS mechanism supports this in the two-token exchange, with the initiator being PROT_READY before consuming the AP-REP. This extension also supports early PROT_READY, which works as follows:

1. The initiator asserts a sub-session key in each AP-REQ that does not follow a key-confirmation KRB-ERROR2, and GSS_Init_sec_context() sets the prot_ready_state return flag on the first call.

    1. If there are multiple such AP-REQs in a security context token exchange, then each such AP-REQ must assert the same sub-session key.

    2. Subsequent AP-REQs need not carry a sub-session key; acceptors MUST ignore sub-session keys from subsequent AP-REQs.

2. GSS_Accept_sec_context() MUST NOT set the prot_ready_state return flag until it has successfully decrypted an AP-REQ’s Ticket and Authenticator from the initiator. If the acceptor requests additional context tokens and signals PROT_READY at that point, then it too will be PROT_READY.

Replay protection for early prot_ready per-message tokens depends on the initiator always generating a fresh sub-session key for every security context’s initial context token, on the acceptor always generating a fresh sub-session key for its key confirmation token, and on either a replay cache or the challenge/response token provided for in this document:

- An attacker cannot replay an early per-message token without also replaying the corresponding initial security context token (as otherwise the initiator-asserted sub-session keys won’t match), and replay protection for the initial security context token provides replay protection for any subsequent early per-message tokens.

- Per-message tokens made after full security context establishment are protected against replay by the use of the acceptor’s sub-session key hierarchy (since the initiator must then use that key).

- AP-REPs and key-confirmation KRB-ERROR2s are protected against replays to initiators by the use of the initiator’s sub-session

key.

- o Initial security context tokens (and error-recovery AP-REQs) are protected against replay either by a replay cache on the acceptor side, or by the use of additional context tokens for challenge/response replay cache avoidance (see Section 6).

3.5. Other Requirements, Recommendations, and Non-Requirements

All error PDUs in an AP exchange where the AP-REQ has the continue-needed-ok ap-options flag MUST be KRB-ERROR2 PDUs.

Whenever an acceptor is able to decrypt the Ticket from an AP-REQ and yet wishes or has to output a KRB-ERROR2, then the enc-part of the KRB-ERROR2 MUST be encrypted in either the initiator’s sub-session key (from the Authenticator) or the Ticket’s session key (if the acceptor could not decrypt the Authenticator).

4. ASN.1 Module for New Protocol Elements

A partial ASN.1 module appears below. This ASN.1 is to be used as if it were part of the base Kerberos ASN.1 module (see RFC4120), therefore the encoding rules to be used are the Distinguished Encoding Rules (DER) [CCITT.X690.2002], and the environment is one of explicit tagging.

```
KerberosExtraContextTokens DEFINITIONS ::=
BEGIN
EXPORTS ad-continue-challenge,
        AD-CONTINUE-CHALLENGE,
        KrbErrorEncPartFlags,
        KRB-ERROR2,
        ErrorFlags;
IMPORTS UInt32, Int32, KerberosTime,
        Microseconds, KerberosFlags,
        Checksum, EncryptedData,
        EncryptionKey, KerberosString,
        AuthorizationData, TYPED-DATA,
        HostAddresses, Ticket FROM KERBEROS5;

APOptions       ::= KerberosFlags
        -- reserved(0),
        -- use-session-key(1),
        -- mutual-required(2),
        -- continue-needed-ok(TBD)

-- Challenge (for use in Authenticator)
ad-continue-challenge     Int32 ::= -5 -- <TBD>
AD-CONTINUE-CHALLENGE ::= OCTET STRING

-- AP options, integrity-protected
ad-ap-options             Int32 ::= -6 -- <TBD>
AD-AP-OPTIONS         ::= KerberosFlags

KrbErrorEncPartFlags ::= ENUMERATED {
        use-null-enctype(0),
        use-initiator-subkey(1),
        use-ticket-session-key(2),
        ...
}

-- Application tag TBD
KRB-ERROR2          ::= [APPLICATION 55] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (55), -- TBD
        enc-part-key    [2] KrbErrorEncPartFlags,
```

```
            enc-part        [3] EncryptedData -- EncKRBErrorPart
    }

    -- Alias type name
    AP-REP2             ::= KRB-ERROR2

    ErrorFlags ::= ENUMERATED {
            final(0),
            continue-needed(1),
            ...
    }

    -- Application tag TBD
    EncKRBErrorPart    ::= [APPLICATION 56] SEQUENCE {
            continue-challenge  [0]  AD-CHALLENGE-RESPONSE,
            stime               [1]  KerberosTime,
            susec               [2]  Microseconds,
            subkey              [3]  EncryptionKey OPTIONAL,
            seq-number          [4]  UInt32 OPTIONAL,
            error-code          [5]  Int32,
            e-flags             [6]  ErrorFlags,
            e-text              [7]  UTF8String OPTIONAL,
            e-data              [8]  OCTET STRING OPTIONAL,
            e-typed-data        [9]  TYPED-DATA OPTIONAL,
            -- For recovery from KRB_AP_ERR_BADADDR:
            your-addresses      [10] HostAddresses OPTIONAL,
            ad-data             [11] AuthorizationData OPTIONAL,
            tgt                 [12] Ticket OPTIONAL, -- for user2user
            ...
    }

    END
```

Figure 1: ASN.1 module (with explicit tagging)

5. Recoverable Errors and Error Recovery

The following Kerberos errors can be recovered from automatically using this protocol:

- o KRB_AP_ERR_TKT_EXPIRED: the initiator should get a new service ticket;
- o KRB_AP_ERR_TKT_NYV: the initiator should get a new service ticket;
- o KRB_AP_ERR_REPEAT: the initiator should build a new AP-REQ;
- o KRB_AP_ERR_SKEW: see Section 8;
- o KRB_AP_ERR_BADKEYVER: the initiator should get a new service ticket;
- o KRB_AP_PATH_NOT_ACCEPTED: the initiator should get a new service ticket using a different transit path;
- o KRB_AP_ERR_INAPP_CKSUM: the initiator should try again with a different checksum type.

Error codes that denote PDU corruption (and/or an active attack) can also be recovered from by attempting a new AP-REQ, though subsequent AP-REQs may fail for the same reason:

- o KRB_AP_ERR_BAD_INTEGRITY
- o KRB_AP_ERR_BADVERSION
- o KRB_AP_ERR_BADMATCH
- o KRB_AP_ERR_MSG_TYPE
- o KRB_AP_ERR_MODIFIED

Other error codes that may be recovered from:

- o KRB_AP_ERR_BADADDR: the acceptor SHOULD include a list of one or more client network addresses as reported by the operating system, but if the acceptor does not then the continue-needed e-flag MUST NOT be included and the error must be final.

5.1. Authenticated Errors

The following errors, at least, can be authenticated in AP exchanges:

- KRB_AP_ERR_TKT_EXPIRED
- KRB_AP_ERR_TKT_NYV
- KRB_AP_ERR_REPEAT
- KRB_AP_ERR_SKEW
- KRB_AP_PATH_NOT_ACCEPTED
- KRB_AP_ERR_INAPP_CKSUM
- KRB_AP_ERR_BADADDR

6. Replay Cache Avoidance

By using an additional AP-REQ and a challenge/response nonce, this protocol is immune to replays of AP-REQ PDUs and does not need a replay cache. Acceptor implementations MUST not insert Authenticators from extra round trips into a replay cache when there are no other old implementations on the same host (and with access to the same acceptor credentials) that ignore critical authorization data or which don't know to reject initial AP-REQs that contain a challenge response nonce.

In the replay cache avoidance case where there's no actual error (e.g., time skew) the acceptor's KRB-ERROR2 will have KDC_ERR_NONE as the error code, with the continue-needed e-flag.

6.1. Replay Cache Avoidance without Extensions

Many Kerberos services can avoid the use of a replay cache altogether, but it's tricky to know when it's safe to do so. For Kerberos it's safe to not use a replay cache for AP-REQs/ Authenticators when either:

- o the application doesn't require replay detection at all and
  - * no other acceptor/service application shares the same long-term service keys for its service principal

or

- o the application protocol always has the initiator/client send the first per-message token (or KRB-SAFE/PRIV PDU) which can then function as a challenge response, and
  - * no other acceptor/service application shares the same long-term service keys for its service principal

It is difficult to establish the second part of the above conjunctions programmatically. In practice this is best left as a local configuration matted on a per-service name basis.

For example, it's generally safe for NFSv4 [RFC3530] to not use a replay cache for the Kerberos GSS mechanism, but it is possible for multiple Kerberos host-based service principals on the same host to share the same keys, therefore in practice, the analysis for NFSv4 requires more analysis. The same is true for SSHv2 [RFC4251] (SSHv2 implementations share the same service principal as other non-GSS Kerberos applications that do sometimes need a replay cache).

7. User-to-User Authentication

There are two user2user authentication cases:

1. the KDC only allows a service principal to use user2user authentication,

2. the service principal does not know its long-term keys or otherwise wants to use user2user authentication even though the KDC vended a service ticket.

In the first case the initiator knows this because the KDC returns KDC_ERR_MUST_USE_USER2USER. The initiator cannot make a valid AP-REQ in this case, yet it must send some sort of initial security context token! For this case we propose that the initiator make an AP-REQ with a Ticket with zero-length enc-part (and null enctype) and a zero-length authenticator (and null enctype). The acceptor will fail to process the AP-REQ, of course, and SHOULD respond with a continue-needed KRB-ERROR2 (using the null enc-type for the enc-part) that includes a TGT for the acceptor.

In the second case the initiator does manage to get a real service ticket for the acceptor but the acceptor nonetheless wishes to use user2user authentication.

In both cases the acceptor responds with a KRB-ERROR2 with the KRB_AP_ERR_USER_TO_USER_REQUIRED error code and including a TGT for itself.

In both cases the initiator then does a TGS request with a second ticket to get a new, user2user Ticket. Then the initiator makes a new AP-REQ using the new Ticket, and proceeds.

8. Acceptor Clock Skew Correction

An initiator in possession of a (short-lived) valid service ticket for a given service principal... must have had little clock skew relative to the service principal’s realm’s KDC(s), or the initiator must have been able to correct its local clock skew. But the acceptor’s clock might be skewed, yielding a KRB_AP_ERR_SKEW error with a challenge. The client could recover from this by requesting a new service ticket with this challenge as an authorization data element. The acceptor should be able to verify this in the subsequent AP-REQ, and then it should be able to detect that its clock is skewed and to estimate by how much.

9. Security Considerations

This document deals with security.

The new KRB-ERROR2 PDU is cryptographically distinguished from the original mechanism's acceptor success security context token (AP-REQ).

Not every KRB-ERROR2 can be integrity protected. This is unavoidable.

Because in the base Kerberos V5 GSS-API security mechanism all errors are unauthenticated, and because even with this specification some elements are unauthenticated, it is possible for an attacker to cause one peer to think that the security context token exchange has failed while the other thinks it will continue. This can cause an acceptor to waste resources while waiting for additional security context tokens from the initiator. This is not really a new problem, however: acceptor applications should already have suitable timeouts on security context establishment.

There is a binding of preceding security context tokens in each additional AP-REQ, via the challenge-response nonce. This binding is weak, and does not detect all modifications of unauthenticated plaintext in preceding security context tokens.

[[anchor1: We could use the GSS_EXTS_FINISHED extension from draft-ietf-kitten-iakerb to implement a strong binding of all context tokens.]]

Early prot_ready per-message tokens have security considerations that are beyond the scope of this document and which are not exhaustively described elsewhere yet. Use only with care.

10. IANA Considerations

[[anchor2: Various allocations are required...]]

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

[CCITT.X680.2002] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.

[CCITT.X690.2002] International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

11.2. Informative References

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, April 2003.

[RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[I-D.swift-win2k-krb-user2user] Swift, M., Brezak, J., and P. Moore, "User to User Kerberos Authentication using GSS-API", draft-swift-win2k-krb-user2user-03 (work in progress), February 2011.

Authors' Addresses

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com

Roland Charles Dowdeswell
Dowdeswell Security Architecture

Email: elric@imrryr.org

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 30, 2015

N. Williams
Cryptonector

October 27, 2014

Public Key-Based Kerberos Cross Realm Path Traversal Protocol Using Kerberized Certification Authorities (kx509) and PKINIT

draft-williams-kitten-krb5-pkcross-05

Abstract

This document specifies a protocol for obtaining cross-realm Kerberos tickets using existing, related protocols: kerberized certification authorities (kx509) and public key cryptography initial authentication in Kerberos (PKINIT). The resulting protocol has a number of desirable properties, primarily that it allows Kerberos to scale to large numbers of realms.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Kerberos [RFC4120] supports meshes of many realms. The individual relationships between realms must be manually keyed, usually with keys derived from passwords. A full mesh wouldn’t scale, therefore the protocol calls for hierarchical trust universes. In practice non-hierarchical but also non-fully-meshed relationships are used, and these generally require distribution of trust routing information to clients, services, and KDCs. With referrals it is possible to reduce the need for client-side trust routing information, but KDCs still need it, as do services (unless they accept KDC trust path policy and the KDC applies it via the TRANSITED-POLICY-CHECKED ticket flag).

These manually-exchanged keys are very difficult to rollover safely, and when they are changed the result is often outages -- controlled outages where foreseen, but outages nonetheless.

Manual cross-realm keying does not scale, and has very poor security properties. We seek to remediate this using public key cryptography, building on existing Kerberos specifications.

Distribution of trust routing (traditionally known as "capaths") and trust path validation (also "capaths") information is difficult; there is no standard protocol for it. Maintenance of it is a thoroughly manual process.

Many years ago there was a proposal for exchanging cross-realm keys using a public key infrastructure (PKI) [RFC5280]; that proposal went by the name "PKCROSS". We appropriate that long-dead proposal’s name, but the protocol specified here is very different from the original proposal.

PKCROSS can make Kerberos scale to large numbers of realms, will remove the need for manual keying of cross-realm TGS principals, will further reduce the need for maintenance and distribution of trust routing information, and will tend to reduce the complexity of trust path validation.

1.1. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. The PKCROSS Protocol

We provide two variants of the PKCROSS protocol: one that is client-driven, and another that is driven by a Ticket Granting Service (TGS) on behalf of its clients. The latter is based on the former, with the TGS acting as a client. We begin with the client-driven case. DNS-Based Authentication of Named Entities (DANE) [RFC6698] can and should be used for realm CA certificate validation.

2.1. Client-Driven PKCROSS

A Kerberos client in with a ticket-granting ticket (TGT) for any one source realm (usually but not necessarily the client’s own realm) wishing to acquire a TGT for a destination realm may use this protocol instead of the traditional cross-realm ticket-granting service (TGS) exchanges as follows:

1. Generate private key to a public key cryptosystem;

2. Request a certificate from the kx509 [RFC6717] service run by the source realm;

3. Request a TGT from the destination realm using PKINIT [RFC4556] and the client certificate obtained in step #2.

If the destination realm issues the requested Ticket then it SHOULD include the client’s certificate in an AD-CLIENT-CERTIFICATE authorization-data element, and it MUST do so if it does not validate the client’s certificate to an acceptable trust anchor. The AD-CLIENT-CERTIFICATE authorization-data MUST be in a KDC-signed authorization-data container [XXX add reference to CAMMAC].

[[anchor1: QUESTION: Should the PKINIT request in step #3 be a TGS-REQ with PKINIT pre-auth data?]]

[[anchor2: QUESTION: Should the PKINIT request in step #3 be required to be used within a FAST tunnel?]]

2.2. TGS-Driven PKCROSS

A TGS can bootstrap ephemeral cross-realm trust principals on behalf of its clients. This allows the cost of PKCROSS to be amortized over many clients, and it allows participation by clients that do not support client-driven PKCROSS (or whose PKCROSS requests are rejected by the target).

In this mode the TGS uses the client-driven PKCROSS protocol, modified as follows:

- o the TGS’s client certificate MUST have an id-pkinit-san Subject Alternative Name (SAN) identifying the source TGS as krbtgt/SOURCE@SOURCE

- o the TGS’s client certificate MUST have an Extended Key Usage (EKU) of id-pkcross-issuer (TBD)

The resulting TGT -which we shall term an "issuer TGT" (ITGT)- and its session key can then be used by the source TGS to create cross-realm TGTs for the source-to-target trust principal ("krbtgt/TARGET@SOURCE").

This ITGT will be used to mint tickets as described below.

2.2.1. Issuing cross-realm TGTs issued for PKCROSS-keyed cross-realm TGS principals

Cross-realm TGTs issued by a source TGS using an ITGT will not be quite like normal Kerberos Tickets: their encrypted part contains an AP-REQ using the ITGT acquired by the source TGS, and this AP-REQ is "encrypted" with the null enctype, The AP-REQ’s Authenitcator MUST contain an authorization-data element that carries a) the name of the client principal, b) the session key that the client should be using with the cross-realm TGTs issued.

```
AD-PKCROSS-TGT-INFO ::= SEQUENCE {
    cname [0] Principal,    -- the client’s realm is the
                            -- crealm from the ITGT’s EncTicketPart
    key   [1] EncryptionKey
}
```

Figure 1: AD-PKCROSS-TGT-INFO

2.2.2. Handling impatient clients

Because the process of acquiring an ITGT might be slow, a TGS doing so on behalf of a client could use a mechanism for instructing the client to be patient. Existing clients would not handler a new error code by waiting, therefore there is not much that can be done to keep an impatient client from retrying at another KDC.

The existing KDC_ERR_SVC_UNAVAILABLE error code cannot be used as often this causes the client to immediately retry the request at another KDC. A new error code for indicating estimated time to completion of request would be handy, but out of scope for this document.

Note that there is a denial of service (DoS) attack by clients on

willing source KDCs: the clients can ask the KDCs to acquire cross-realm ITGTs for many target realms. Ideally the quality of service for the Kerberos authentication service (AS) with PKINIT (and/or other slow pre-authentication mechanisms) should be separate from that of the Kerberos TGS co-located with it, and the PKCROSS-capable TGS as well, so as to be able to throttle low-priority requests when under load.

2.3. Stapled DANE

[[anchor3: TBD. We should use Google's serialization of DNS RRsets needed for DANE validation. We will need a label for the TLSA RRs for kx509 issuers.]]

2.4. Validation

KDCs processing PKINIT requests crossing realms MUST apply either or both of:

- o PKIX certificate validation

- o DANE certificate validation

KDCs MUST reject PKINIT requests from clients of foreign realms whose certificates cannot be validated, unless the client request the anonymous principal name in the target's realm.

2.5. Transit Path

The combined Kerberos/PKIX/DNSSEC transit path MUST be represented in any tickets issued using PKCROSS (see below). As usual, each realm's KDCs in the mix can set the transit policy checked flag if a client's transit path is acceptable per the realm's KDCs' local policy.

Two validation mechanisms are available: all PKIX [RFC5280] validation methods, and DANE [RFC6698]. DANE validation records SHOULD be stapled onto the client certificates by the issuing kx509 CA; alternatively, clients can staple <http://src.chromium.org/viewvc/chrome/trunk/src/net/base/dnssec_chain_verifier.cc?pathrev=167227> onto their PKINIT requests using an authorization-data element, AD-PKINIT-CLIENT-DANE.

Additionally, when PKIX certificate validation is used, the trust path should be encoded in an AD-INITIAL-VERIFIED-CAS authorization data element, per-PKINIT.

2.5.1. Transit path representation

The notional transit path for a ticket issued by a target realm's KDCs includes:

- o the source realm (never expressed in the 'transited' field of Kerberos Tickets)

- o all realms in the ITGT's transited field (in the TGS-driven PKCROSS case)

- o all issuers in the validation path for the kx509-issued certificate, which are

  - * all issuers in the certificate's PKIX validation path when PKIX validation is used

  - * all DNS zone domainnames transited from the source realm's domainname to the root zone

- o the target realm (also never expressed in the 'transited' field)

When using DANE for validation of the issuer's certificate the target SHOULD represent the transit path as hierarchical from the source realm's domain to the root domain, then direct from there to the target's realm.

The notional transit path for a given client principal MUST be encoded as usual, using the Kerberos X.500 and domain-style representations of PKIX issuer names and DNS domainnames as faithfully to the original as possible.

[[anchor4: QUESTION: Do we need a 100% faithful representation of the transit path?]]

2.6. Exchange of Long-Term Cross-Realm Symmetric Keys

A KDC can acquire a TGT using PKCROSS whose session key then becomes the long-lived, persistent symmetric key for a cross-realm principal from the source realm to the target realm ("krbtgt/TARGET@SOURCE").

To do this the KDC MUST set the USE-SESSION-KEY-AS-REALM-KEY KDCOptions flag (TBD) in its request for an ITGT from the target realm. As usual, the target realm's KDC MUST validate the client principal's certificate. The target realm's KDC MUST NOT return a TGS-REP until the new principal is committed to its principal database, and MUST set the endtime of the ITGT to the time at which the source realm may begin using the new symmetrically-keyed

principal.

The source realm’s KDC MUST commit the new principal to its principal database and MUST NOT begin using the new principal’s long-term keys until the new principal is available to all KDCs for the source realm and the endtime of the ITGT passes.

Target KDCs SHOULD require manual pre-approval of such new cross-realm principals. In small, isolated environments a KDC MAY be configured to pre-approve all such new principals.

By default, source KDCs SHOULD NOT automatically request long-term keying of cross-realm principals.

3. Security Properties

The proposed PKCROSS protocol has several useful properties described below.

3.1. Automatic Cross-Realm Keying

No more manual keying of cross-realm principals via exchanging passwords in-person on a telephone call (or similar).

3.2. Scalability

Kerberos with commonplace symmetrically-keyed hierarchical cross-real trusts can scale to a large universe of realms, but only if there are top-level realms that are willing to pair-wise trust and "child" realms. Such top-level realms do not exist in practice, leading to an O(N^2) scaling problem for most two-label realms.

Leveraging a PKI, such as a PKIX PKI [RFC5280] or a DNSSEC PKI [RFC4033] removes the need for either top-level realms (which are not likely to ever be operated as commercial or even non-profit entities) or O(N^2) pair-wise cross-realm symmetric keying.

The cost of this is having to add PKI trust paths to Kerberos trust paths (though the resulting trust path length need not be much different than before).

3.2.1. Simplified trust routing

For clients, relying on referrals (and TGS-driven PKCROSS) and/or client-driven PKCROSS will greatly reduce the need for client-side trust routing information.

Even KDCs won’t need trust routing information.

3.2.2. Simplified trust path validation

For services that accept hierarchical trust paths, PKCROSS will greatly reduce the complexity of trust path / transit validation. Such services that also trust DANE/DNSSEC will need no trust path validation information for any clients using PKCROSS to reach the service. In many cases a very simple policy expressed in terms of whitelists of top-level domains (TLDs) or near top-level domains traversed, trust anchor sets, and trust of all zero-length transit paths, will suffice.

3.3. Privacy Protection relative to home realm

This protocol protects the privacy of client principals vis-a-vis their home realms, when the clients use the client-driven PKCROSS protocol.

This feature is generally and naturally available in PKI, and as this protocol is based on a kerberized certification authority, this protocol inherits this privacy feature from PKI.

The realms visited by the client may, of course, inform the client’s home realm, but in the event that they don’t, the client does gain this small measure of privacy. Of course, the privacy-conscious client SHOULD attach an OCSP Response [RFC6960] to its PKINIT request, per [RFC4557].

4. Application Programming Interface Considerations

Improved scalability for Kerberos realm traversal implies larger Kerberos universes, and the larger a universe of trust the more important it is to have useful and expressive local policy for evaluating the trustworthiness of any given transit path. Because in most applications local policy should be a component external to the application, there is mostly no impact on APIs here. However, an implementation may wish to provide applications with interfaces for specifying policies, either named or by value.

4.1. GSS-API Considerations

The naming attributes [RFC6680] defined in [I-D.williams-kitten-generic-naming-attributes] provide access to information about transit paths.

Note that information about how PKCROSS was used to establish symmetrically-keyed cross-realm principals is lost and will not appear in the transit path in tickets issued by KDCs reached via such cross-realm principals.

5. Security Considerations

[[anchor5: All the security considerations of Kerberos and PKI apply. Security considerations are discussed throughout this document.]]

Scaling up the universe of realms reachable via any trust path necessarily dilutes trust overall, but not for specific paths. On the other hand, by shortening transit path lengths trust can be improved, though some short transit paths will have been symmetrically keyed using this PKCROSS protocol and therefore will be longer than they appear to be. These are subjective notions of trust, of course.

5.1. Loss of Cross-Realm Principal Trust Establishment Information

Once a cross-realm principal is symmetrically keyed the transit path used to automatically key that principal will no longer appear in subsequent cross-realm tickets issued by the target.

The Kerberos transit path encodes only realm names (including X.500-style names, thus PKIX certificate subject and issuer names), and lacks any public key information that might be useful for pinning. However, the certificate validation path for each realm in a transit path SHOULD be included in the transit path.

5.2. On the Need for a Common Transit Path Policy Language

There are no standard ways to express authorization policies for trust transit paths for either Kerberos nor PKI. A standard language for this would be extremely useful. Such a language should allow for the expression of policies for both, clients and services. Such a language should allow for the expression of complex realm/domain/other naming, and should allow for HSTS-style pinning [add references -Nico]. Such a language should allow for multiple paths where desired, and should allow for more than path rejection: it should also allow for reducing the entitlements assigned to a peer/realm for authorization purposes.

The need for a standard transit path policy expression language is not new, and such a language is broadly and generally needed. Therefore such a language is outside this document’s scope.

PKCROSS can greatly simplify the process of validating a ticket’s trust path, first by shortening the number of realms involved to two (in the typical case), maybe three, second by making the actual trust path (PKI or DANE) hierarchical, thus hopefully leaving much less policy to express in a transit path policy language: whitelists of domains and sub-domains, perhaps. But a common language would still

be desirable.

5.3. On the Need for Trust Routing

A common language for trust routing is not necessary in a purely hierarchical world, as in DANE. But since it’s likely that there will be some non-hierarchical, non-zero-length transit paths in many deployments for a long time to come, a common language for trust routing would be desirable as well. Routing protocols normally used for network addresses could be used for discovery and distribution of trust routing information as well. But note that there are subtle differences between trust routing and trust path validation, even though in traditional Kerberos deployments the same information is used for both, with the trust validation policy effectively being that the client must have taken the shortest, highest-priority path specified in "capaths" configutation.

6. IANA Considerations

[[anchor6: Allocate the new KDCOptions flag (USE-SESSION-KEY-AS-REALM-KEY) and authorization-data element (AD-CLIENT-CERTIFICATE), as well as the new EKU id-pkcross-issuer.]]

7. TODO

- Provide a normative reference for DANE stapling.

8. Acknowledgements

Although the author arrived at this "kx509 + PKINIT == PKCROSS" idea independently, it is not an original idea. Henry Hotz and Jeffrey Altman each conceived the same idea years earlier. It is a relatively obvious idea when taking into account efforts to bridge disparate security mechanisms and credentials infrastructures.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4556] Zhu, L. and B. Tung, "Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 4556, June 2006.

[RFC4557] Zhu, L., Jaganathan, K., and N. Williams, "Online Certificate Status Protocol (OCSP) Support for Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)", RFC 4557, June 2006.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC6680] Williams, N., Johansson, L., Hartman, S., and S. Josefsson, "Generic Security Service Application Programming Interface (GSS-API) Naming Extensions", RFC 6680, August 2012.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

[RFC6717] Hotz, H. and R. Allbery, "kx509 Kerberized Certificate Issuance Protocol in Use in 2012", RFC 6717, August 2012.

[I-D.williams-kitten-generic-naming-attributes] Williams, N., "Generic Naming Attributes for the Generic Security Services Application Programming Interface (GSS-API)", draft-williams-kitten-generic-naming-attributes-01 (work in progress), August 2013.

9.2. Informative References

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, June 2013.

Author’s Address

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com