Network Working Group J. Myers
Internet Draft Netscape Communications
Document: draft-ietf-cat-sasl-gssapi-01.txt May 2000

SASL GSSAPI mechanisms

Status of this Memo

A revised version of this draft document will be submitted to the RFC editor as a Proposed Standard for the Internet Community. Discussion and suggestions for improvement are requested.

1. Abstract

The Simple Authentication and Security Layer [SASL] is a method for adding authentication support to connection-based protocols. This document describes the method for using the Generic Security Service Application Program Interface [GSSAPI] in the Simple Authentication and Security Layer [SASL].

This document amends section 7.2 of RFC 2222 [SASL], the definition of the "GSSAPI" SASL mechanism.

2. Organization of this Document

2.1. How to Read This Document

[TODO: is this section needed?]

2.2. Conventions Used in this Document

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as defined in "Key words for use in RFCs to Indicate Requirement Levels" [KEYWORDS].

2.3. Examples

[TODO: No examples included. Needed?]

Examples in this document are for the IMAP profile [IMAP4] of this specification. The base64 encoding of challenges and responses, as well as the "+ " preceding the responses are part of the IMAP4 profile, not part of the SASL specification itself.

3. Introduction and Overview

Each and every GSSAPI mechanism used within SASL is implicitly registered by this specification.

For backwards compatibility with existing implementations of Kerberos

V5 and SPNEGO under SASL, the SASL mechanism name for the Kerberos V5 GSSAPI mechanism [GSSAPI-KERBEROS] is "GSSAPI" and the SASL mechanism for the SPNEGO GSSAPI mechanism [SPNEGO] is "GSS-SPNEGO". The SASL mechanism name for any other GSSAPI mechanism is the concatenation of "GSS-" and the Base32 encoding of the first ten bytes of the MD5 hash [MD5] of the ASN.1 DER encoding [ASN1] of the GSSAPI mechanism's OID. Base32 encoding is described later in this document

SASL mechanism names starting with "GSS-" are reserved for SASL mechanisms which conform to this document.

The specification of all SASL mechanisms conforming to this document is in the "Specification common to all GSSAPI mechanisms" section of this document.

The IESG is considered to be the owner of all SASL mechanisms which conform to this document. This does NOT imply that the IESG is considered to be the owner of the underlying GSSAPI mechanism.

4. SPNEGO

Implementations SHOULD NOT use the Simple and Protected GSS-API Negotiation Mechanism [SPNEGO] underneath SASL.

A client which supports, for example, the Kerberos V5 GSSAPI mechanism only underneath SPNEGO underneath the "GSS-SPNEGO" SASL mechanism will not interoperate with a server which supports the Kerberos V5 GSSAPI mechanism only underneath the "GSSAPI" SASL mechanism.

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 the "GSS-SPNEGO" SASL mechanism, it may end up using mechanism Z when it should have used mechanism Y.

One reason a server or client might want to violate the above SHOULD directive is if it has a policy of only using mechanisms below a certain strength if their negotiation is protected. In such a case, it would only want to negotiate those weaker mechnisms through SPNEGO. In any case, there is no down-negotiation security consideration with using the strongest mechanism and set of options

the implementation supports, so for interoperability that mechanism and set of options MUST be negotiable without using the "GSS-SPNEGO" mechanism.

5. Base32 encoding

The Base32 encoding is designed to represent arbitrary sequences of octets in a form that needs to be case insensitive but need not be humanly readable.

A 33-character subset of US-ASCII is used, enabling 5 bits to be represented per printable character. (The extra 33rd character, "=", is used to signify a special processing function.)

The encoding process represents 40-bit groups of input bits as output strings of 8 encoded characters. Proceeding from left to right, a 40-bit input group is formed by concatenating 5 8bit input groups. These 40 bits are then treated as 8 concatenated 5-bit groups, each of which is translated into a single digit in the base32 alphabet. When encoding a bit stream via the base32 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first 8bit byte, and the eighth bit will be the low-order bit in the first 8bit byte, and so on.

Each 5-bit group is used as an index into an array of 32 printable characters. The character referenced by the index is placed in the output string. These characters, identified in Table 1, below, are selected from US-ASCII digits and uppercase letters.

Table 1: The Base32 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	9	J	18	S	27	3
1	B	10	K	19	T	28	4
2	C	11	L	20	U	29	5
3	D	12	M	21	V	30	6
4	E	13	N	22	W	31	7
5	F	14	O	23	X		
6	G	15	P	24	Y	(pad)	=
7	H	16	Q	25	Z		

8 I 17 R 26 2

Special processing is performed if fewer than 40 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 40 input bits are available in an input group, zero bits are added (on the right) to form an integral number of 5-bit groups. Padding at the end of the data is performed using the "=" character. Since all base32 input is an integral number of octets, only the following cases can arise: (1) the final quantum of encoding input is an integral multiple of 40 bits; here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding, (2) the final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by six "=" padding characters, (3) the final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be four characters followed by four "=" padding characters, (4) the final quantum of encoding input is exactly 24 bits; here, the final unit of encoded output will be five characters followed by five"=" padding characters, or (5) the final quantum of encoding input is exactly 32 bits; here, the final unit of encoded output will be seven characters

followed by one "=" padding character.

Because it is used only for padding at the end of the data, the occurrence of any "=" characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three and no "=" characters are present.

Any characters outside of the base32 alphabet are to be ignored in base32-encoded data.

6. Specification common to all GSSAPI mechanisms

Each SASL mechanism which uses a GSSAPI mechanism uses the following specification.

6.1. Client side of authentication protocol exchange

The client calls GSS_Init_sec_context, passing in

input_context_handle of 0 (initially), mech_type of the GSSAPI mechanism for which this SASL mechanism is registered, and targ_name equal to output_name from GSS_Import_Name called with input_name_type of GSS_C_NT_HOSTBASED_SERVICE and input_name_string of "service@hostname" where "service" is the service name specified in the protocol's profile, and "hostname" is the fully qualified host name of the server. The client then responds with the resulting output_token. If GSS_Init_sec_context returns GSS_S_CONTINUE_NEEDED, then the client should expect the server to issue a token in a subsequent challenge. The client must pass the token to another call to GSS_Init_sec_context, repeating the actions in this paragraph.

When GSS_Init_sec_context returns GSS_S_COMPLETE, the client takes the following actions: If the last call to GSS_Init_sec_context returned an output_token, then the client responds with the output_token, otherwise the client responds with no data. The client should then expect the server to issue a token in a subsequent challenge. The client passes this token to GSS_Unwrap and interprets the first octet of resulting cleartext as a bit-mask specifying the security layers supported by the server and the second through fourth octets as the maximum size output_message to send to the server. The client then constructs data, with the first octet containing the bit-mask specifying the selected security layer, the second through fourth octets containing in network byte order the maximum size output_message the client is able to receive, and the remaining octets containing the UTF-8 encoded [UTF8] authorization identity. The client passes the data to GSS_Wrap with conf_flag set to FALSE, and responds with the generated output_message. The client can then

consider the server authenticated.

6.2. Server side of authentication protocol exchange

The server passes the initial client response to GSS_Accept_sec_context as input_token, setting input_context_handle to 0 (initially). If GSS_Accept_sec_context returns GSS_S_CONTINUE_NEEDED, the server returns the generated output_token to the client in challenge and passes the resulting response to another call to GSS_Accept_sec_context, repeating the actions in this paragraph.

When GSS_Accept_sec_context returns GSS_S_COMPLETE, the server takes

the following actions: If the last call to GSS_Accept_sec_context returned an output_token, the server returns it to the client in a challenge and expects a reply from the client with no data. Whether or not an output_token was returned (and after receipt of any response from the client to such an output_token), the server then constructs 4 octets of data, with the first octet containing a bit-mask specifying the security layers supported by the server and the second through fourth octets containing in network byte order the maximum size output_token the server is able to receive. The server must then pass the plaintext to GSS_Wrap with conf_flag set to FALSE and issue the generated output_message to the client in a challenge. The server must then pass the resulting response to GSS_Unwrap and interpret the first octet of resulting cleartext as the bit-mask for the selected security layer, the second through fourth octets as the maximum size output_message to send to the client, and the remaining octets as the authorization identity. The server must verify that the src_name is authorized to authenticate as the authorization identity. After these verifications, the authentication process is complete.

6.3. Security layer

The security layers and their corresponding bit-masks are as follows:

```
   1 No security layer
   2 Integrity protection.
     Sender calls GSS_Wrap with conf_flag set to FALSE
   4 Privacy protection.
     Sender calls GSS_Wrap with conf_flag set to TRUE
```

Other bit-masks may be defined in the future; bits which are not understood must be negotiated off.

7. IANA Considerations

The IANA is directed to modify the existing registration for "GSSAPI" in the "sasl-mechanisms" so that this document is listed as the published specification. Add the descriptive text "This mechanism is for the Kerberos V5 mechanism of GSSAPI. Other GSSAPI mechanisms use

other SASL mechanism names, as described in this mechanism's published specification."

The IANA is advised that SASL mechanism names starting with "GSS-" are reserved for SASL mechanisms which conform to this document.

8. References

[ASN1] ISO/IEC 8824, "Specification of Abstract Syntax Notation One (ASN.1)"

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

[GSSAPI-KERBEROS] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996

[IMAP4] Crispin, M., "Internet Message Access Protocol - Version 4", RFC 1730, University of Washington, December 1994.

[KEYWORDS] Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997

[MD5] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992

[SASL] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997

[SPNEGO] Baize, E., Pinkas., D., "The Simple and Protected GSS-API Negotiation Mechanism", RFC 2478, December 1998

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998

9. Security Considerations

Security issues are discussed throughout this memo.

When a server or client supports multiple authentication mechanisms, each of which has a different security strength, it is possible for an active attacker to cause a party to use the least secure mechanism supported. To protect against this sort of attack, a client or server which supports mechanisms of different strengths should have a configurable minimum strength that it will use. It is not sufficient for this minimum strength check to only be on the server, since an active attacker can change which mechanisms the client sees as being supported, causing the client to send authentication credentials for its weakest supported mechanism.

The client's selection of a SASL mechanism is done in the clear and may be modified by an active attacker. It is important for any new SASL mechanisms to be designed such that an active attacker cannot obtain an authentication with weaker security properties by modifying

the SASL mechanism name and/or the challenges and responses.

SPNEGO [SPNEGO] has protection against many of these down-negotiation attacks, SASL does not itself have such protection. The section titled "SPNEGO" mentions considerations of choosing negotiation through SASL versus SPNEGO.

Additional security considerations are in the SASL [SASL] and GSSAPI [GSSAPI] specifications.

10. Author's Address

John G. Myers
Netscape Communications
501 E. Middlefield Road
Mail Stop SCA 15:201
Mountain View, CA 94043-4042

Email: jgmyers@netscape.com

Appendix A. Sample code

The following is an example program which converts mechanism OIDs (of the form "1.3.6.1.5.5.1") to SASL mechanism names. This sample program uses the reference MD5 implementation in [MD5].

```
#include <stdio.h>
#include "md5.h"

unsigned long parsenum(char **ptr)
{
    unsigned long rval = 0;
    while (**ptr >= '0' && **ptr <= '9') {
        rval = rval * 10 + *(*ptr)++ - '0';
    }
    return rval;
}

void encode(unsigned long val, unsigned char **buf)
{
    unsigned long tmpval;
    int noctets = 1;
    for (tmpval = val; tmpval >= 128; tmpval >>= 7) noctets++;
    while (--noctets) {
        *(*buf)++ = ((val >> (7 * noctets)) & 0x7f) | 0x80;
    }
    *(*buf)++ = val & 0x7f;
}

static char basis_32[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ234567";

void base32encode10(unsigned char *buf)
{
    int len = 10;
    while (len) {
        putc(basis_32[buf[0] >> 3], stdout);
```

```
        putc(basis_32[((buf[0] & 7) << 2) | (buf[1] >> 6)], stdout);
        putc(basis_32[(buf[1] & 0x3f) >> 1], stdout);
        putc(basis_32[((buf[1] & 1) << 4) | (buf[2] >> 4)], stdout);
        putc(basis_32[((buf[2] & 0xf) << 1) | (buf[3] >> 7)], stdout);
        putc(basis_32[(buf[3] & 0x7f) >> 2], stdout);
        putc(basis_32[((buf[3] & 3) << 3) | (buf[4] >> 5)], stdout);
        putc(basis_32[(buf[4] & 0x1f)], stdout);
        len -= 5;
    }
}

main(int argc, char **argv)
```



```
{
    char *oidstr;
    unsigned long val1, val2;
    unsigned char asn1buf[1024]; /* Had better be big enough */
    unsigned char *asn1next = asn1buf;
    unsigned char *asn1lennext;
    unsigned char *p;
    MD5_CTX md5ctx;
    unsigned char md5buf[16];

    if (argc != 2) {
        fprintf(stderr, "usage: %s oid\n", argv[0]);
        exit(1);
    }

    oidstr = argv[1];
    val1 = parsenum(&oidstr);
    if (*oidstr++ != '.') goto badoid;
    val2 = parsenum(&oidstr);
    if (*oidstr && *oidstr++ != '.') goto badoid;
    *asn1next++ = val1 * 40 + val2;

    while (*oidstr) {
        val1 = parsenum(&oidstr);
        if (*oidstr && *oidstr++ != '.') goto badoid;

        encode(val1, &asn1next);
    }
```

```
        /* Now that we know the length of the OID, generate the tag
         * and length
         */
        asn1lennext = asn1next;
        *asn1lennext++ = 6;
        encode(asn1next - asn1buf, &asn1lennext);

        printf("ASN.1 DER encoding:  ");
        for (p = asn1next; p < asn1lennext; p++) {
            printf("%02x ", *p);
        }
        for (p = asn1buf; p < asn1next; p++) {
            printf("%02x ", *p);
        }
        printf("\n");

        MD5Init(&md5ctx);
        MD5Update(&md5ctx, (unsigned char *)asn1next, asn1lennext - asn1next);
        MD5Update(&md5ctx, (unsigned char *)asn1buf, asn1next - asn1buf);
```



```
        MD5Final(md5buf, &md5ctx);

        printf("MD5 hash:            ");
        for (p = md5buf; p < md5buf + 16; p++) {
            printf("%02x ", *p);
        }
        printf("\n");

        printf("SASL mechanism name: GSS-");
        base32encode10(md5buf);
        printf("\n");

        exit(0);
    badoid:
        fprintf(stderr, "incorrect oid syntax\n");
        exit(1);
    }
```

Table of Contents