IVI-6.5: SASL Mechanism Specification

May 19, 2022 Edition

Revision 1.0

Important Information

This specification (IVI-6.5: SASL Mechanism Specification) is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at `www.ivifoundation.org`.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through the web site at `www.ivifoundation.org`.

Warranty

The IVI Foundation and its member companies make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The IVI Foundation and its member companies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Trademarks

Product and company names listed are trademarks or trade names of their respective companies.

No investigation has been made of common-law trademark rights in any work.

Table of Contents

SASL Mechanism Specification

IVI SASL Mechanism Revision History

This section is an overview of the revision history of the IVI-6.5 specification. Specific individual additions/modifications to the document in draft revisions are denoted with diff-marks, "|", in the right hand column of a line of text for which the change/modification applies.

Table 1-1. IVI SASL Mechanism Specification Revisions

Revision Number	Date of Revision	Revision Notes
Revision 1.0	May 19, 2022	Initial Version

1. Overview of SASL Mechanism Specification

The IVI-6.5 SASL Mechanism specification specifies the implementation of SASL (RFC 4422 at the time of this writing) mechanisms used by IVI communication protocols. At the time of this writing, IVI-6.1 High-Speed LAN Instrument Protocol (HiSLIP) is the only IVI-defined protocol that utilizes SASL.

The SASL protocol provides a way for a client and server to negotiate a mechanism whereby the client can offer credentials to the server in order to establish a connection that provides client authentication. However, some of the standard mechanisms that are negotiated by SASL provide enough flexibility in their configuration that a client and server may not be able to communicate without some external agreement as to the configuration of the mechanisms.

This specification defines the aspects of the SASL mechanisms that are necessary to enable a VISA client to connect to a HiSLIP server. Note that:

- The details regarding the mechanisms are contained in the respective mechanism specifications.
- The negotiation of the mechanisms is performed using SASL, in which the server offers a list of mechanisms to the client.
- Some aspects regarding which mechanisms are enabled and therefore proffered by the server to the client are specified by the LXI Security Extended Function. So, although this specification requires entities to support certain mechanisms, servers are permitted to accept configurations that disable them.

Although there are numerous interrelated specifications, this specification does not require compliance with SASL or the LXI Security Extended Function.

2. Nomenclature

This document usually refers to clients and servers, consistent with the definition of the mechanisms which are the subject of this specification. However, for the purposes of this specification, the *client* is typically the VISA library, and the *server* is an instrument (or instrument-like device).

When a statement applies to both a client and server, then the object is referred to as an *entity*.

3. Username Requirements

In order to ensure interoperability across entities, the following rules shall be observed:

- Encoding of UTF-8 and ASCII shall be supported. Note that SASL implementations for SCRAM use RFC 4013 (string prep) to handle translation.
- Clients/servers must be able to send/accept username and password up to 255 octets.
- Minimum number of characters shall be 1.
- The following characters shall be accepted: <all characters supported by the encoding>
    - Clients shall not filter out characters used to indicate domain, such as backslash "\" and commercial at "@".
- Username must not contain NUL character ('\0')

Usernames shall be transported while preserving case. The case sensitivity of the consuming protocols is outside the control of the specification.

3.1 Authorization ID

Clients shall permit sending the provided username as the PLAIN username with an empty authorization ID. Clients may be configured to send an arbitrary authorization ID, but the server behavior is not guaranteed by this specification.

Servers shall accept a username without an authorization ID.

4. Password Requirements

Entities shall not restrict passwords beyond the limitations imposed by the environment in which the passwords are used. That environment may include: the underlying operating system, authentication mechanisms, password representation, storage mechanisms, et cetera. For instance, a Windows implementation imposes limitations on the password, however the entity implementations shall not further restrict accepted values unless required to by some other mechanism.

5. Anonymous Mechanism Requirements

Clients should identify themselves by including RFC 4505 message field. Clients should use a string that is a valid syntax for an e-mail address, such as *account@hostname* or perhaps *anonymous@hostname*.

6. SCRAM Mechanism Requirements

The SCRAM mechanism refers to the Salted Challenge Response Mechanism as defined by RFC 5802 as updated by RFC 7677. In general SCRAM enhances security by ensuring that both the client and server are in possession of the client credentials. SCRAM includes 'PLUS' mechanisms that provide channel binding to protect against man-in-the-middle attacks and non-PLUS mechanisms that do not provide channel binding.

Entities shall support SCRAM-SHA-256-PLUS and SCRAM-SHA-256.

These SCRAM versions use the SHA-256 hash algorithm. SHA-256 is considered secure at the time of this writing.

At the time of this writing, The SCRAM mechanisms are defined in RFC 5802 and RFC 7677. Entities should take into consideration both subsequent versions of these RFCs and backwards compatibility.

In general implementation may support additional SCRAM mechanisms and configurations, however, the selections required in this chapter are intended to ensure interoperation between compliant clients and servers. Therefore, the configurations specified here shall be the default operation. Explicit user configuration of entities to behave differently is permitted but is beyond the scope of this specification. Such configurations shall not preclude configurations that permit entities to comply with this specification.

The following sections describe the required SCRAM mechanisms.

6.1 Iteration Count

Servers are permitted to select an iteration count. That is, the number of times the channel binding token is hashed. The iteration count should be set high enough that brute force attacks are discouraged due to the total compute time. Per RFC 7677 the compute time should be about 100ms for current client computers. At the time of this writing, RFC 7677 suggests 4096 as the minimum iteration count.

Clients shall accept any iteration count indicated by the server.

6.2 The SCRAM-SHA-256-PLUS Channel Binding

SCRAM-SHA-256-PLUS supports several different channel bindings. As part of the SCRAM channel binding protocol, the client requests a type of channel binding. However, RFC 5802 does not provide a way for the client to determine the channel bindings supported by the server. Therefore, this specification requires specific channel binding to ensure interoperability.

In general, it is useful for entities to support several channel bindings, however this specification only requires limited channel bindings to simplify implementation and guarantee interoperability. RFC 5929 specifies *tls-server-end-point* and *tls-unique* channel bindings.

All entities that support SCRAM shall support both *tls-server-end-point* and *tls-unique. tls-server-end-point* can be used with TLS 1.3, at the time of this writing, *tls-unique* is not defined for TLS 1.3

SCRAM clients that comply with this specification shall prefer *tls-server-end-point* channel binding.. That is, unless explicitly configured otherwise by the user, compliant clients shall initially request *tls-server-end-point* channel binding. Clients may successively attempt to connect with other channel bindings if *tls-server-end-point* fails.

6.2.1 Selection of the Prefix for the Channel Binding Value

SCRAM Channel Binding values include a prefix string. That string should be chosen for compatibility with common industry implementations. As shown in the sections below, the prefix for *tls-unique* shall be “tls-unique:” and the prefix for *tls-server-end-point* shall be “tls-server-end-point:”.

6.2.2 Generation of the *tls-unique* Channel Binding Value

Entities shall calculate the *tls-unique* channel binding value using algorithms that are substantially equivalent to the following. External calls are as defined by OpenSSL and Microsoft Windows.

```
std::string CSslSocket::GetTlsUnique()
```

```
{
  // See https://paquier.xyz/postgresql-2/channel-binding-openssl/
  if (!m_ssl)
    return std::string();

  std::string sResult = "tls-unique:";
  if (m_bClient == (SSL_session_reused(m_ssl) != 0))
    sResult += GetPeerFinished();
  else
  sResult += GetFinished();
  return sResult;
}

std::string CSslSocket::GetFinished()
{
  char msg[256];
  if (m_ssl)
  {
    size_t count;
    count = SSL_get_finished(m_ssl, msg, sizeof(msg));
    if (count)
    {
      return std::string(msg, count);
    }
  }
  return std::string();
}

std::string CSslSocket::GetPeerFinished()
{
  char msg[256];
  if (m_ssl)
  {
    size_t count;
    count = SSL_get_peer_finished(m_ssl, msg, sizeof(msg));
    if (count)
    {
      return std::string(msg, count);
    }
  }
  return std::string();
}
```

6.2.3 Generation of the *tls-server-end-point* Channel Binding Value

Entities shall calculate the *tls-server-end-point* channel binding value using algorithms that are substantially equivalent to the following:

```
std::string CSslSocket::GetTlsEndpoint()
{
```

```
const EVP_MD   *algo_type = NULL;
char            hash[EVP_MAX_MD_SIZE];  /* size for SHA-512 */
unsigned int    hash_size;
int             algo_nid;
X509           *server_cert;

// See https://paquier.xyz/postgresql-2/channel-binding-openssl/
if (!m_ssl)
  return std::string();

/* Get certificate data, be careful that this could be NULL */
if (m_bClient)
  server_cert = SSL_get_peer_certificate(m_ssl);
else
  server_cert = SSL_get_certificate(m_ssl);

/*
 * Get the signature algorithm of the certificate to determine the
 * hash algorithm to use for the result.
 */
if (!OBJ_find_sigid_algs(X509_get_signature_nid(server_cert), &algo_nid, NULL))
  return std::string(); //elog(ERROR, "could not find signature algorithm");

/* Switch to the hashing algorithm to use */
switch (algo_nid)
{
    case NID_sha512:
        algo_type = EVP_sha512();
        break;

    case NID_sha384:
        algo_type = EVP_sha384();
        break;

    /*
     * Fallback to SHA-256 for weaker hashes, and keep them listed
     * here for reference.
     */
    case NID_md5:
    case NID_sha1:
    case NID_sha224:
    case NID_sha256:
    default:
        algo_type = EVP_sha256();
        break;
}

/* generate and save the certificate hash */
if (!X509_digest(server_cert, algo_type, (unsigned char*)hash, &hash_size))
```

```
    return std::string(); //  elog(ERROR, "could not generate server certificate hash");

  std::string sResult = "tls-server-end-point:";
  sResult += std::string(hash, hash_size);
  return sResult;
}
```