[SRD-PROTO]:
Secure Remote Delegation Protocol

Revision History

Revision summary			
Author	**Date**	**Revision history**	**Comments**
Marc-André Moreau	12/01/2016	0.1	Initial draft

Contents

1 Introduction

The Secure Remote Delegation (SRD) protocol is used as a means of delegating complete user credentials (username and password) to a remote system for user logon. The goal of this protocol is to provide a secure way of delegating credentials when a regular challenge-response protocol like NTLM or Kerberos cannot be used for mutual authentication.

2 Protocol Details

This section describes the Secure Remote Delegation (SRD) protocol details.

2.1 Common Details

This section provides details on protocol variables and how they are used.

KeySize: The size of the Diffie-Hellman keys.

Generator: The Diffie-Hellman generator, also known as the 'g' parameter.

Prime: The Diffie-Hellman prime, also known as the 'p' parameter.

ClientNonce: A 32-byte client random (nonce).

ServerNonce: A 32-byte client random (nonce).

ClientPrivateKey: The Diffie-Hellman client private key, also known as 'a'.

ClientPublicKey: The Diffie-Hellman client public key, also known as 'A' in A = g^a mod p.

ServerPrivateKey: The Diffie-Hellman server private key, also known as 'b'.

ServerPublicKey: The Diffie-Hellman server public key, also known as 'B' in B = g^b mod p.

SecretKey: The Diffie-Hellman shared secret key, also known as 's'. The client computes it using s = B^a mod p. The server computes it using s = A^b mod p.

DelegationKey: A 32-byte secret key given by SHA256(ClientNonce, SecretKey, ServerNonce).

IntegrityKey: A 32-byte secret key given by SHA256(ServerNonce, SecretKey, ClientNonce).

IV: A 32-byte initialization vector given by SHA256(ClientNonce, ServerNonce). Only the first 16 bytes of this initialization vector are used.

CertData: The X.509 server certificate used for the TLS connection, in DER format. If the server sends a TLS certificate chain, only the last certificate of the chain should be used.

ClientCbt (32 bytes): The 32-byte client channel binding token (CBT), given by HMAC_SHA256(IntegrityKey, (ClientNonce, CertData)).

ServerCbt (32 bytes): The 32-byte server channel binding token (CBT), given by HMAC_SHA256(IntegrityKey, (ServerNonce, CertData)).

Username: The logon username, in UPN ('@') or Down-Level ('\') format. The domain name is optional and can be omitted.

Password: The logon password that corresponds to the logon username.

BlobType: The authentication payload, or blob type. The current version of the protocol only supports the Logon blob type.

BlobData: The authentication payload, or blob data, specific to the blob type. This contains the authentication credentials or other sensitive data that needs to be securely delegated.

MsgBuffers: An array of all message buffers as they appear on the wire. If the MAC field is saved in the buffer, the SRD_FLAG_MAC flag can be used to skip it when computing the MAC.

MAC (32 bytes): A 32-byte message authentication code (MAC), given by the HMAC_SHA256 of all messages up to the current message, excluding all MAC fields, using the IntegrityKey as the key.

This section describes common pseudocode functions used in both the client and server logic.

RAND(size): generates a random number of the given size

ModExp(a, p, m): a^p % m, or 'a' to the power 'p' modulo 'm'.

ModpGroup(size): retrieves a Diffie-Hellman generator and prime of the given size from RFC3526. The only supported sizes in the current protocol version are 2048, 4096 and 8192 bits.

(DelegationKey, IntegrityKey, IV) = DeriveKeys(SecretKey, ClientNonce, ServerNonce):

DelegationKey = SHA256(ClientNonce, SecretKey, ServerNonce)

IntegrityKey = SHA256(ServerNonce, SecretKey, ClientNonce)

IV = SHA256(ClientNonce, ServerNonce)

ComputeCbt(key, nonce, data): HMAC_SHA256(key, (nonce, data))

ComputeMac(key, buffers, count): HMAC_SHA256(key, buffers) where buffers is an array of 'count' message buffers, excluding the MAC fields.

EncryptBlob(key, iv, data): AES256_CBC(key, iv, data)

DecryptBlob(key, iv, data): AES256_CBC(key, iv, data)

(username, password) = ObtainLogonData(): obtain logon credentials (username, password).

ValidateLogonData(username, password): validate logon credentials, return zero when successful.

EncodeLogonBlob(username, password): store username and password in two successive 128-byte fields. Truncate each field to 127 bytes, enforcing a null terminator, and fill the remaining bytes with random values.

(username, password) = DecodeLogonBlob(blobData): enforce null terminators at offsets 127 and 255 inside the blob. Interpret strings at offsets 0 and 128 as the username and password.

2.2 Client Details

This section describes the client protocol sequencing and processing rules.

2.2.1 Initialization

KeySize is set to 2048, 4096 or 8192 bits.

If the channel binding token is to be used, the CertData protocol variable is set to the TLS X.509 certificate of the server, in DER format.

The state transitions from the initial state to the Negotiate state.

2.2.2 Sending Initiate Message

The client sends the Initiate message to transition from Initiate state to the Offer state.

InitiateMsg.keySize = KeySize

MsgBuffers[0] = InitiateMsg

2.2.3 Receiving Offer Message

The client receives the Offer message to transition from the Offer state to the Accept state.

MsgBuffers[1] = OfferMsg

AssertEquals(OfferMsg.keySize, KeySize)

Generator = OfferMsg.generator

Prime = OfferMsg.prime

ServerPublicKey = OfferMsg.publicKey

ServerNonce = OfferMsg.nonce

2.2.4 Sending Accept Message

The client sends the Accept message to transition from the Offer state to the Confirm state.

ClientNonce = RAND(32)

ClientPrivateKey = RAND(KeySize)

ClientPublicKey = ModExp(Generator, ClientPrivateKey, Prime)

SecretKey = ModExp(ServerPublicKey, ClientPrivateKey, Prime)

(DelegationKey, IntegrityKey, IV) = DeriveKeys(SecretKey, ClientNonce, ServerNonce)

ClientCbt = ComputeCbt(IntegrityKey, ClientNonce, CertData)

AcceptMsg.keySize = KeySize

AcceptMsg.publicKey = ClientPublicKey

AcceptMsg.nonce = ClientNonce

AcceptMsg.cbt = ClientCbt

MsgBuffers[2] = AcceptMsg

AcceptMsg.mac = ComputeMac(IntegrityKey, MsgBuffers, 3)

2.2.5 Receiving Confirm Message

The client receives the Confirm message to transition from the Confirm state to the Delegate state.

MsgBuffers[3] = ConfirmMsg

ServerCbt = ComputeCbt(IntegrityKey, ServerNonce, CertData)

ExpectedMac = ComputeMac(IntegrityKey, MsgBuffers, 4)

AssertEquals(ConfirmMsg.cbt , ServerCbt)

AssertEquals(ConfirmMsg.mac, ExpectedMac)

2.2.6 Sending Delegate Message

The client sends the Delegate message to transition from the Delegate state to the Result state.

(Username, Password) = ObtainLogonData()

BlobType = Logon

BlobData = EncodeLogonBlob(Username, Password)

DelegateMsg.blobType = BlobType

DelegateMsg.blobData = EncryptBlob(DelegationKey, IV, BlobData)

MsgBuffers[4] = DelegateMsg

Delegate.mac = ComputeMac(IntegrityKey, MsgBuffers, 5)

2.3 Server Details

This section describes the server protocol sequencing and processing rules.

2.3.1 Initialization

The CertData protocol variable is set to the TLS X.509 certificate of the server, in DER format.

The state transitions from the initial state to the Negotiate state.

2.3.2 Receiving Initiate Message

The server receives the Initiate message to transition from the Initiate state to the Offer state.

MsgBuffers[0] = InitiateMsg

KeySize = InitiateMsg.keySize

(Generator, Prime) = ModpGroup(KeySize)

2.3.3 Sending Offer Message

The server sends the Offer message to transition from the Offer state to the Accept state.

ServerNonce = RAND(32)

ServerPrivateKey = RAND(KeySize)

ServerPublicKey = ModExp(Generator, ServerPrivateKey, Prime)

OfferMsg.keySize = KeySize

OfferMsg.publicKey = ServerPublicKey

OfferMsg.nonce = ServerNonce

MsgBuffers[1] = OfferMsg

2.3.4 Receiving Accept Message

The server receives the Accept message to transition from the Accept state to the Confirm state.

MsgBuffers[2] = AcceptMsg

AssertEquals(AcceptMsg.keySize, KeySize)

ClientPublicKey = AcceptMsg.publicKey

ClientNonce = AcceptMsg.nonce

SecretKey = ModExp(ClientPublicKey, ServerPrivateKey, Prime)

(DelegationKey, IntegrityKey, IV) = DeriveKeys(SecretKey, ClientNonce, ServerNonce)

ClientCbt = ComputeCbt(IntegrityKey, ClientNonce, CertData)

ExpectedMac = ComputeMac(IntegrityKey, MsgBuffers, 3)

AssertEquals(AcceptMsg.cbt, ClientCbt)

AssertEquals(AcceptMsg.mac, ExpectedMac)

2.3.5 Sending Confirm Message

The server sends the Confirm message to transition from the Confirm state to the Delegate state.

ServerCbt = ComputeCbt(IntegrityKey, ServerNonce, CertData)

ConfirmMsg.cbt = ServerCbt

ConfirmMsg.mac = ComputeMac(IntegrityKey, MsgBuffers, 4)

MsgBuffers[3] = ConfirmMsg

2.3.6 Receiving Delegate Message

The server receives the Delegate message to transition from the Delegate state to the Result state.

MsgBuffers[4] = DelegateMsg

ExpectedMac = ComputeMac(IntegrityKey, MsgBuffers, 5)

AssertEquals(AcceptMsg.mac, ExpectedMac)

BlobType = DelegateMsg.blobType

BlobData = DecryptBlob(DelegationKey, IV, DelegateMsg.blobData)

AssertEquals(BlobType, Logon)

(Username, Password) = DecodeLogonBlob(BlobData)

AuthStatus = ValidateLogonData(Username, Password)

3 Messages

This section describes the SRD protocol messages.

3.1 Transport

The SRD protocol is meant to be transport agnostic, but adaptable to different use cases.

3.1.1 Transport Layer Security (TLS)

When the underlying transport is TLS and the server certificate information is available to both the client and server applications, then the Channel Binding Token (CBT) feature SHOULD be enabled.

3.1.2 Hypertext Transfer Protocol (HTTP)

The HTTP authentication scheme name for SRD is "SRD".

Since the server certificate information is generally not available directly to a client-side or server-side web application, the Channel Binding Token (CBT) feature SHOULD NOT be enforced in this case. While HTTPS is recommended, SRD can be used with an insecure transport such as HTTP since it provides its own layer of security.

SRD is a multilegged authentication protocol, meaning that it requires the exchange of a series of messages to be completed. Since HTTP is inherently stateless, the client and server have no standard to associate messages belonging to the same authentication sequence. To solve this problem, the solution documented in [draft-montenegro-httpbis-multilegged-auth] is recommended.

SRD HTTP authentication works as follows:

The client sends an HTTP GET request.

The server responds with an HTTP 401 Unauthorized response with the following header fields:

- WWW-Authenticate: SRD
- Auth-ID: <auth-id-token>

The client sends a new HTTP GET request with the following header fields:

- Authorization: SRD <srd-msg1-base64>
- Auth-ID: <auth-id-token>

The server responds with an HTTP 401 Unauthorized response with the following header fields:

- WWW-Authenticate: SRD <srd-msg2-base64>
- Auth-ID: <auth-id-token>

The client sends a new HTTP GET request with the following header fields:

- Authorization: SRD <srd-msg2-base64>
- Auth-ID: <auth-id-token>

The server responds with an HTTP 401 Unauthorized response with the following header fields:

- WWW-Authenticate: SRD <srd-msg3-base64>
- Auth-ID: <auth-id-token>

The client sends a new HTTP GET request with the following header fields:

- Authorization: SRD <srd-msg4-base64>
- Auth-ID: <auth-id-token>

If authentication is successful, the server responds with an HTTP 200 OK response with the following header fields:

- WWW-Authenticate: SRD <srd-msg3-base64>
- Auth-ID: <auth-id-token>

The authentication context associated with the Auth-ID field is deleted after this step.

If authentication fails, the server responds with an HTTP 403 Forbidden response and the following header fields:

- Auth-ID: <auth-id-token>

This response can be sent by the server at any time. The authentication context associated with the Auth-ID field is deleted after this step.

3.2 Message Syntax

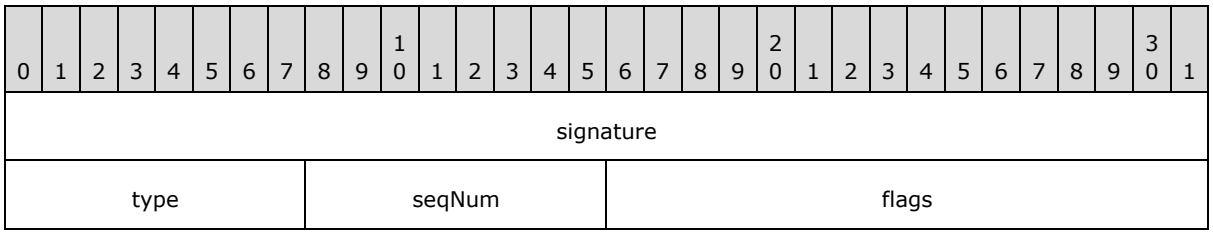
This section describes the protocol message encoding.

3.2.1 Protocol Messages

This section describes the encoding of protocol messages.

3.2.1.1 SRD_HEADER

The SRD_HEADER structure is shared by all SRD messages.



signature (4 bytes): This SRD message signature. This field MUST contain the null-terminated 4-byte string "SRD". As a 32-bit little-endian unsigned integer, the signature is equal to 0x00445253.

type (1 byte): The SRD message type.

Value	Meaning
SRD_INITIATE_MSG_ID 0x01	SRD_INITIATE_MSG
SRD_OFFER_MSG_ID 0x02	SRD_OFFER_MSG
SRD_ACCEPT_MSG_ID 0x03	SRD_ACCEPT_MSG
SRD_CONFIRM_MSG_ID 0x04	SRD_CONFIRM_MSG
SRD_DELEGATE_MSG_ID 0x05	SRD_DELEGATE_MSG

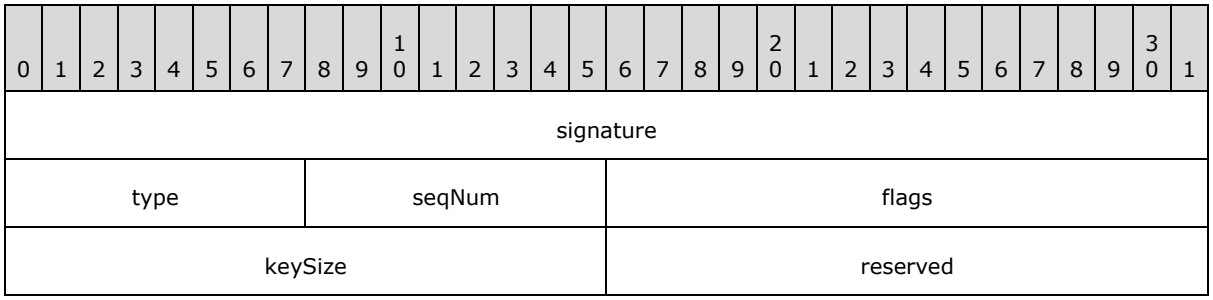
seqNum (1 byte): The SRD message sequence number. The sequence number starts at zero and is incremented for each message in the sequence.

flags (2 bytes): The SRD message flags.

Flag	Meaning
SRD_FLAG_MAC 0x0001	A 32-byte Message Authentication Code (MAC) is present at the end of the message.
SRD_FLAG_CBT 0x0002	The usage of a Channel Binding Token (CBT) is required.

3.2.1.2 SRD_INITIATE_MSG

The SRD_INITIATE_MSG structure is the 1st message of the SRD authentication sequence.



signature (4 bytes): This SRD message signature.

type (1 byte): The SRD message type, MUST be set to 1.

seqNum (1 byte): The SRD message sequence number.

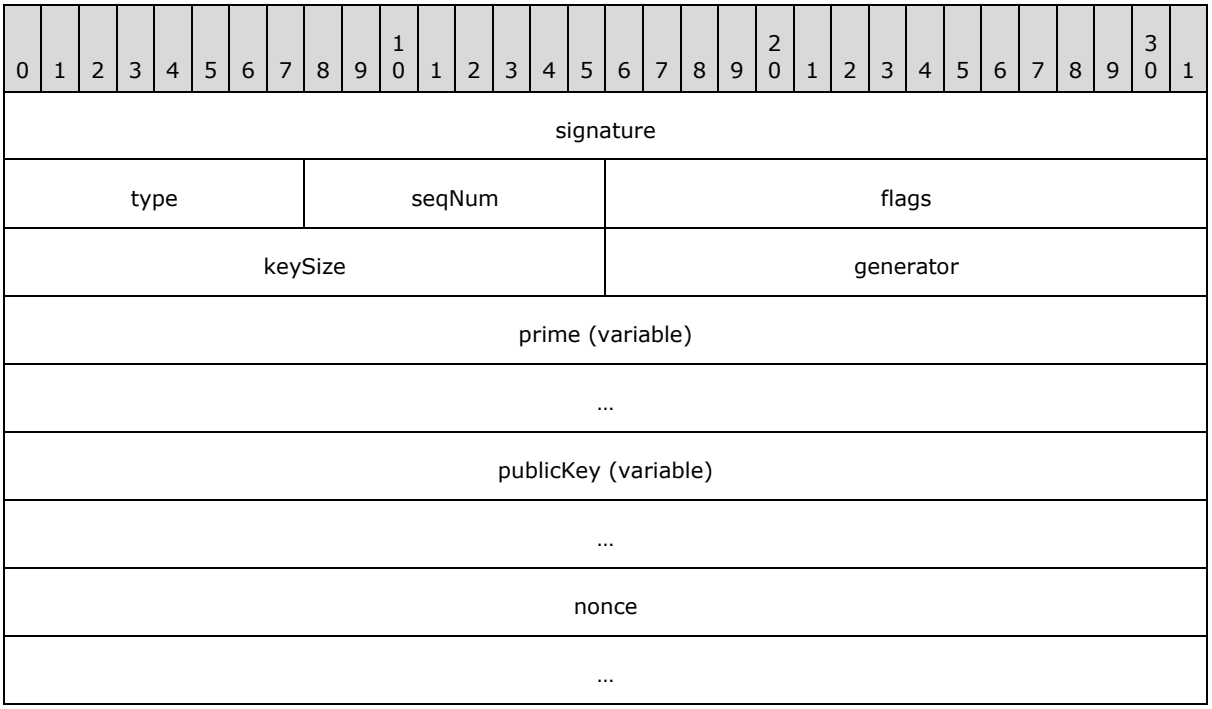
flags (2 bytes): The SRD message flags. The SRD_FLAG_MAC flag MUST NOT be set. The SRD_FLAG_CBT indicates support for the Channel Binding Token (CBT) feature.

keySize (2 bytes): The requested Diffie-Hellman key size. This field MUST be set to one of the following values: 256 (2048 bits), 512 (4096 bits) or 1024 (8192 bits). Key sizes of 1024 bits and smaller are not supported because they are considered weak and vulnerable to the logjam attack.

reserved (2 bytes): This field is unused and reserved for future use. It MUST be set to zero.

3.2.1.3 SRD_OFFER_MSG

The SRD_INITIATE_MSG structure is the 2nd message of the SRD authentication sequence.



signature (4 bytes): This SRD message signature.

type (1 byte): The SRD message type, MUST be set to 2.

seqNum (1 byte): The SRD message sequence number.

flags (2 bytes): The SRD message flags. The SRD_FLAG_MAC flag MUST NOT be set.

keySize (2 bytes): The Diffie-Hellman key size, MUST be set to the same value from the negotiate message.

generator (2 bytes): The Diffie-Hellman generator, as a 2-byte big endian number. This is known as the Diffie-Hellman 'g' parameter.

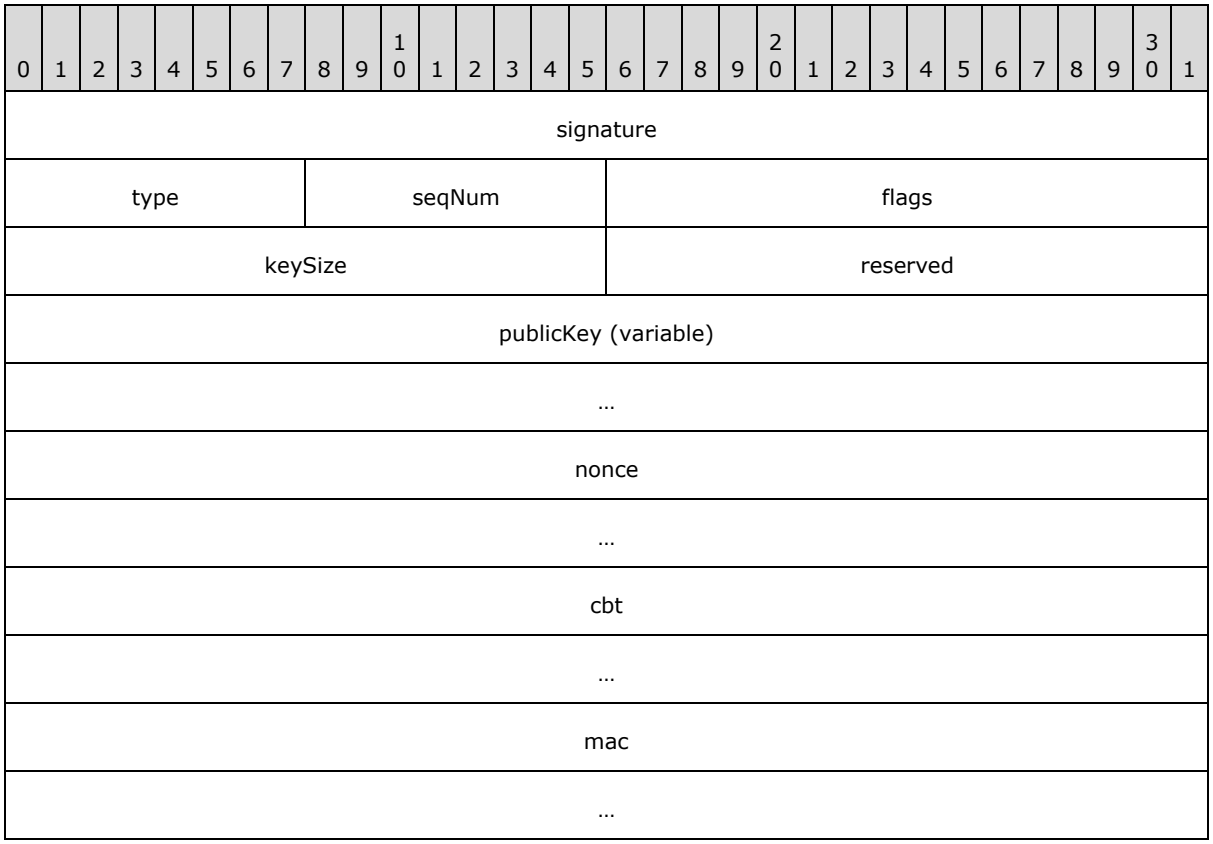
prime (variable): The Diffie-Hellman prime, as a big-endian number of the size given by the keySize field. This is known as the Diffie-Hellman 'p' parameter.

publicKey (variable): The Diffie-Hellman server public key, as a big-endian number of the size given by the keySize field. This is also known as 'B' in B = g^b mod p, where 'b' is the server Diffie-Hellman private key.

nonce (32 bytes): A 32-byte server random (nonce).

3.2.1.4 SRD_ACCEPT_MSG

The SRD_ACCEPT_MSG structure is the 3rd message of the SRD authentication sequence. It is sent by the client as a response to the server challenge.



signature (4 bytes): This SRD message signature.

type (1 byte): The SRD message type, MUST be set to 3.

seqNum (1 byte): The SRD message sequence number.

flags (2 bytes): The SRD message flags. The SRD_FLAG_MAC flag MUST be set. The SRD_FLAG_CBT indicates support for the Channel Binding Token (CBT) feature.

keySize (2 bytes): The Diffie-Hellman key size, MUST be set to the same value from the negotiate message.

reserved (2 bytes): This field is unused and reserved for future use. It MUST be set to zero.

publicKey (variable): The Diffie-Hellman client public key, as a big-endian number of the size given by the keySize field. This is also known as 'A' in A = g^a mod p, where 'a' is the client Diffie-Hellman private key.

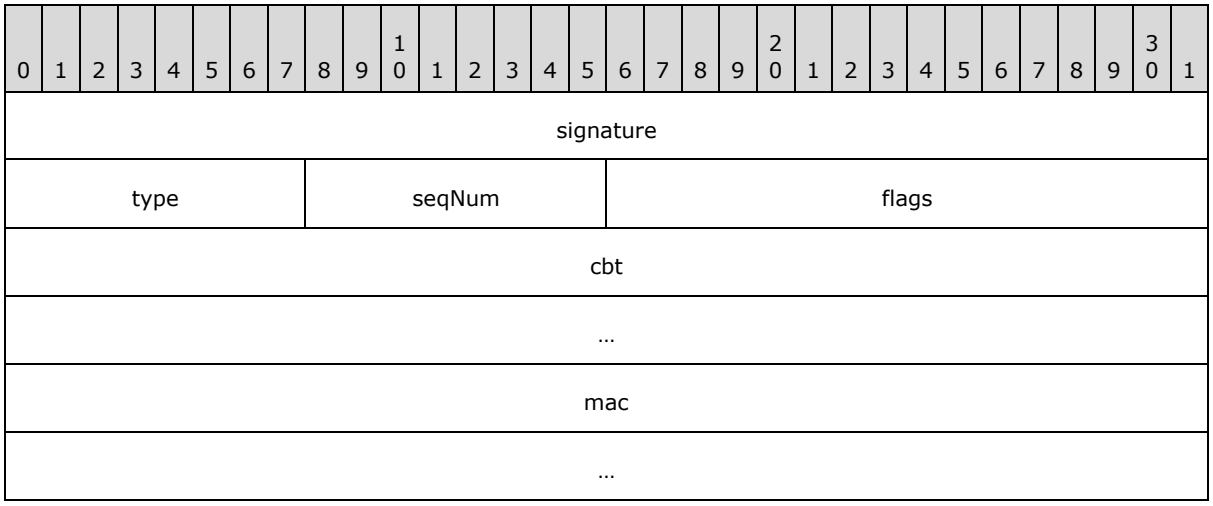
nonce (32 bytes): A 32-byte client random (nonce).

cbt (32 bytes): The 32-byte client channel binding token (CBT). The value is given by HMAC_SHA256(IntegrityKey, (ClientNonce, CertData)) and MUST be validated by the server. If the SRD_FLAG_CBT flag is not set, then the CBT is computed with an empty CertData value (zero length).

mac (32 bytes): A 32-byte message authentication code (MAC).

3.2.1.5 SRD_CONFIRM_MSG

The SRD_CONFIRM_MSG structure is the 4th message of the SRD authentication sequence.



signature (4 bytes): This SRD message signature.

type (1 byte): The SRD message type, MUST be set to 4.

seqNum (1 byte): The SRD message sequence number.

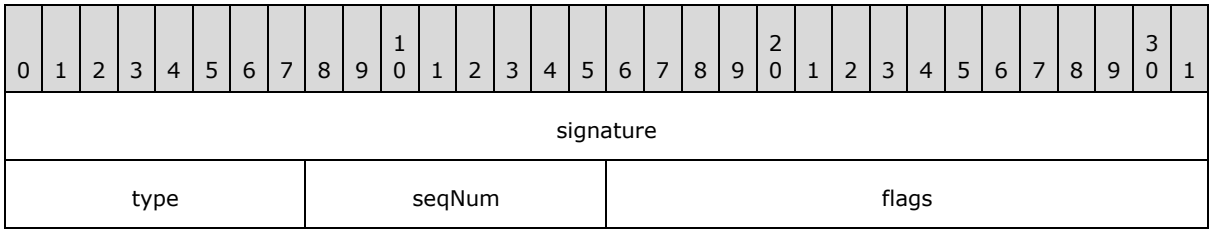
flags (2 bytes): The SRD message flags. The SRD_FLAG_MAC flag MUST be set. The SRD_FLAG_CBT indicates support for the Channel Binding Token (CBT) feature.

cbt (32 bytes): The 32-byte server channel binding token (CBT). The value is given by HMAC_SHA256(IntegrityKey, (ServerNonce, CertData)) and MUST be validated by the client. If the SRD_FLAG_CBT flag is not set, then the CBT is computed with an empty CertData value (zero length).

mac (32 bytes): A 32-byte message authentication code (MAC).

3.2.1.6 SRD_DELEGATE_MSG

The SRD_DELEGATE_MSG structure is the 5th message of the SRD authentication sequence.



size
blob (variable)
…
mac
…

signature (4 bytes): This SRD message signature.

type (1 byte): The SRD message type, MUST be set to 5.

seqNum (1 byte): The SRD message sequence number.

flags (2 bytes): The SRD message flags. The SRD_FLAG_MAC flag MUST be set.

reserved (4 bytes): This field is unused and reserved for future use. It MUST be set to zero.

size (4 bytes): The encrypted blob size, in bytes.

blob (variable): The encrypted blob structure (SRD_BLOB).

The blob structure is encrypted using AES-256 in CBC mode using the DelegationKey variable as the key and the first 16 bytes of the IV variable as an initialization vector.

mac (32 bytes): A 32-byte message authentication code (MAC).

3.2.2 Blob Payloads

This section describes the encoding of SRD blob payloads.

3.2.2.1 SRD_BLOB

The SRD_BLOB structure is used to encapsulate the delegated encrypted payload. The blob is encrypted using AES-256 in CBC mode using the DelegationKey variable as the key and the first 16 bytes of the IV variable as an initialization vector. The total blob size MUST be a multiple of the AES-256 block size (16). The number of padding bytes does not need to be minimal, additional padding bytes may be used to increase the total blob size to prevent possible hints on the real payload size.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
typeSize																typePadding															
dataSize																dataPadding															
type (variable)																															
…																															

data (variable)
…

typeSize (2 bytes): The blob type size, including the null terminator and excluding padding.

typePadding (2 bytes): The blob type padding, in bytes.

dataSize (2 bytes): The blob data size, excluding padding.

dataPadding (2 bytes): The blob data padding, in bytes.

type (variable): The blob type, encoded as a null-terminated UTF-8 string. The number of padding bytes is given by the typePadding field. Padding MUST be used to ensure that the end of this field is aligned to the AES-256 block size (16), relative to the beginning of the SRD_BLOB structure. Padding bytes MUST be ignored and SHOULD be filled with random data.

The blob type is used to identify a specific blob data format. Vendors can define their own blob types to fit their needs. The following table defines a list of known blob types:

Value	Meaning
"Basic"	Basic blob
"Logon"	Logon blob
"Change"	Change blob

data (variable): The blob data. The number of padding bytes is given by the dataPadding field. Padding MUST be used to ensure that the end of this field is aligned to the AES-256 block size (16), relative to the beginning of the SRD_BLOB structure. Padding bytes MUST be ignored and SHOULD be filled with random data.

3.2.2.2 Basic Blob

The "Basic" blob is the same as the HTTP Basic authentication scheme defined in RFC7617, with the following differences: the string is encoded as a null-terminated UTF-8 string, and base64 encoding is not used. In other words, the "Basic" blob is a simple string that contains the username and password pair separated by a colon (':') character:

<username>:<password>

The "Basic" blob is simple, but it does not work with credentials that contain the colon (':') character.

3.2.2.3 Logon Blob

The "Logon" blob encodes logon credentials.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
usernameLength																passwordLength															

username (variable)
…
password (variable)
…

usernameLength (2 bytes): The number of characters in the username field, excluding the null terminator.

passwordLength (2 bytes): The number of characters in the password field, excluding the null terminator.

username (variable): A null-terminated username.

password (variable): A null-terminated password.

3.2.2.4 Change Blob

The "Change" blob encodes a username, old password and new password to perform a password change.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
usernameLength																oldPasswordLength															
newPasswordLength																changeFlags															
username (variable)																															
…																															
oldPassword (variable)																															
…																															
newPassword (variable)																															
…																															

usernameLength (2 bytes): The number of characters in the username field, excluding the null terminator.

oldPasswordLength (2 bytes): The number of characters in the oldPassword field, excluding the null terminator.

newPasswordLength (2 bytes): The number of characters in the newPassword field, excluding the null terminator.

changeFlags (2 bytes): The change blob flags.

Flag	Meaning
SRD_CHANGE_BLOB_FLAG_LOGON 0x0001	Perform a logon before changing the password. This can be used to perform a password change on every logon, and therefore enforce single-use passwords.

username (variable): A null-terminated username.

oldPassword (variable): The old null-terminated password.

newPassword (variable): The new null-terminated password.