Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 11, 2018

D. Cridland
Surevine Ltd

January 7, 2018

Client Key SASL mechanism
draft-cridland-kitten-clientkey-00

Abstract

This document proposes a SASL mechanism which might be used to authenticate specific clients on devices owned by a user.

Status of This Memo

Copyright Notice

Table of Contents

1. 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].

2. Overview

Authentication within a "pure" SASL ([RFC4422]) environment - ie, without call-outs to SAML or OAuth - might include TOTP pathways such as [XEP-0388] proposes, and may also include multiple round-trips, typically to strengthen security on password-based protocols.

It seems desirable to design a SASL mechanism to handle the "reauthentication" case needed to avoid client-side storage of reusable password data, bypass TOTP and similar, and allow for low RTT counts. CLIENT-KEY is a SASL mechanism designed to be used when supported by an application protocol framework which allows users to enumerate and invalidate individual clients or devices. It is designed to be a single round-trip, use channel binding where available, and avoid storage of plaintext-equivalent credentials on the server.

2.1. Initial Flow

A typical interaction with a new client might look as follows:

1. On connecting, the client uses a traditional mechanism based on a password, such as SCRAM.

2. After authenticating successfully with SCRAM, the client is put through a TOTP challenge.

3. The client offers to the user to "remember this device" or similar. If the user wants to do so, the client performs device registration and obtains a "client key", storing it locally.

2.2. Subsequent Authentication

The next time the client need to authenticate, it can use CLIENT-KEY:

1. On connecting, the client uses CLIENT-KEY to authenticate.

2. The server notes that CLIENT-KEY has been used, and elides TOTP.

If its client key is due to expire, it MAY at this point re-register, generating a new client key.

3. Notation

This document uses relatively common notations for pseudocode:

H(message) The H function is a cryptographic hash function computing the digest of the message - in this document always SHA-256. The function returns some binary data. It is assumed to be both collision-resistant and too difficult to practically guess message from H(message).

HMAC(key, message) The HMAC function computes a MAC of the second argument, keyed by the first argument, according to the algorithm defined in [RFC2104]. It is assumed that given HMAC(key, message) and message, it is too difficult to practically guess key. Given only HMAC(key, message), it is assumed that guessing message is difficult within a reasonable time. The hash function used within the HMAC algorithm is H above.

BASE64(message) The BASE64 function returns a string which represents the message encoded according to [RFC4648].

NORMALIZE(string) The NORMALIZE function returns a string which has been processed by whatever one normalizes with these days.

R(n) The R function returns a sequence of n octets generated randomly with high entropy.

L(message) This function returns the number of octets in the message (ie, the message length in octets).

HASHLEN This constant is the equivalent of L(H("")) - it is the length of the output of the hash function.

XOR(msg1, msg2) The XOR function returns a bitwise XOR of msg1 against msg2. These two arguments MUST be the same length.

4. The CLIENT-KEY mechanism

4.1. Mechanism Name

This document defines two mechanisms, CLIENT-KEY and CLIENT-KEY-PLUS. Both are based on SHA-256. Future documents may offer alternative hash algorithms.

4.2. Commencing State

The client has information stored as follows:

ClientID The ClientID, an opaque string which uniquely identifies the device and client instance for that authorization-id.

Secret The Client Secret Key, a random sequence of HASHLEN octets.

ValidationKey The Client Validation Key, a random sequence of HASHLEN octets.

Counter A Counter which records the number of times the Secret has been used.

Expiry The Expiry of the client key, after which is it no longer valid.

If the client does not have these values stored, it obtains them by authenticating as the user via some other mechanism and registering as described below.

The server has information stored during this registration as follows:

ClientID As above.

Counter Also as above.

EncryptedSecret This has the value XOR(Secret, ValidationKey).

Validator This has the value HMAC(EncryptedSecret, ValidationKey).

Expiry The Expiry of the client key, after which is it no longer valid.

4.3. Client Initial Response

The client constructs an initial response as follows:

```
client-initial-response = gs2-header NUL authcid NUL client-id
                NUL client-hmac NUL client-validation-key
authcid = 1*UTF-8-char
client-id = 1*UTF-8-char
client-hmac = base64string
                ;  = BASE64(HMAC(Secret, client-hmac-input))
client-hmac-input = "Client Response" NUL authcid-norm
                NUL client-id NUL Counter
                [ NUL channel-binding-data ]
                ; optional channel binding if -PLUS is used.
client-validation-key = base64string
                ; = BASE64(ValidationKey)
authcid-norm = 1*UTF-8-char
                ; = NORMALIZE(username)
username = 1*UTF-8-char
```

The client and server both calculate the client-hmac by:

1. Creating a message as: "Client Response" NUL authcid NUL client-id NUL counter

2. If CLIENT-KEY-PLUS is used, append a NUL followed by the channel binding information.

3. Calculating an HMAC using SHA-256 of the message, keyed by the Secret.

4. Base64-encoding the result.

After the client sends the response, the counter is incremented.

4.4. Server Addition Data With Success

When the client's initial response is received, the server first validates the ValidationKey provided, by checking if HMAC(EncryptedSecret, ValidationKey) matches its stored Validator.

If this is not the case, the authentication attempt is rejected with no further action.

If it matches, then any failure from this point on MUST result in this key being revoked.

The server extracts Secret from EncryptedSecret as XOR(EncryptedSecret, ValidationKey), and calculates its own value of client-hmac. At this point, the Counter is updated - note that this step is performed prior to comparing the two client-hmac values.

Finally the two client-hmac values are compared. If the client's matches that calculated by the server, the authentication succeeds. Success data is passed back as follows:

```
server-success-data = base64string
                ;  = BASE64(HMAC(Secret, server-hmac-input))
server-hmac-input = "Server Response" NUL authcid
                NUL client-id NUL Counter
                [ NUL channel-binding-data ]
                ; optional channel binding if -PLUS is used.
```

On receipt of this, the client calculates its own version. If the computed value of server-success-data differs from that supplied by the server it should abort the connection.

5. Additional Application Protocol Support

5.1. Client Registration

A client obtains the key by sending a message to the server containing four items of information to the server:

1. A ClientID, which is a identifier unique within the scope of the authzid for the client instance, expressed as an opaque string. Good options for this include a UUID, better options include a hash of the device serial number or similar.

2. A Client Name, which is a (potentially non-unique) human-readable name for the client instance. For example, "MegaBrowser on Linux", or "SuperClient on MyPhone".

3. A ValidationKey, used within the mechanism to validate that the client knows the key, and decrypt the secret. This MUST be random, and consist of HASHLEN octets. An effective method for generating this is either R(HASHLEN) or H(R(40)).

4. A requested TTL, which gives the lifetime of the key. This might be short, for session-based keys, or longer for persistent keys.

The server then generates Secret, and calculates EncryptedSecret as XOR(Secret, ValidationKey). Secret MUST be HASHLEN random octets, and again an effective method might be R(HASHLEN) or H(R(40)). It then stores Validator as H(ValidationKey) and EncryptedSecret only.

The server then responds with a generated value of EncryptedSecret and a timestamp giving the expiry time. This is the only point at which the EncryptedSecret should be transferred.

The server MUST store only the items noted above, and most especially MUST NOT store Secret or ValidationKey.

5.2. Key Revocation

Any authenticated client may revoke a key belonging to the same user by sending a message to the server containing the ClientID corresponding to an existing key. This simply causes the record of the ClientID, Counter, EncryptedSecret and Validator to be removed.

5.3. Key Enumeration

Any authenticated client may enumerate keys belonging to the same user by sending a message to the server. The server responds with a list of items each containing a ClientID and the Client Name. Note that the key is not included.

6. Security Considerations

This document is concerned with security throughout. This section is concerned with specific threats and mitigations.

Our threat model assumes that an attacker can (with effort) obtain the complete server database, may observe network traffic between the client and server, and may obtain whatever data is stored on an individual client.

6.1. Exposure of key

The Secret transferred from the server to the client during client registration is clearly vulnerable to anyone able to observe the unencrypted data on the connection. The connection therefore MUST be protected by TLS or equivalent encryption.

It may also be extracted from the client at any point, since for use it needs to be stored in such a way that the Secret, ValidationKey

and Counter are able to be retrieved. The effect of such compromise can be mitigated by using relatively short expiry times, but it is naturally mitigated by use of the counter, which means that an attacker using the key causes the key to be invalidated on the original device, alerting the user to a compromise and a likely revocation cycle. This attack is undetectable if a long-expiry key is unused by the legitimate client; we therefore recommend short-expiry keys and that users are advised to revoke the keys of lost devices.

The Secret cannot be obtained due to a server breach as long as only the EncryptedSecret is stored. Servers MUST NOT store the Secret itself. Similarly, the ValidationKey MUST NOT be stored on the server.

6.2. Dangerous Implementation Shortcuts

If the server does not test that the HMAC(EncryptedSecret, ValidationKey) matches Validator, then an attacker who has obtained the server database can supply any value for ValidationKey and simply use XOR(EncryptedSecret,ValidatorKey) as their corresponding value for Secret. This would allow an attacker access based only on data obtained from the server.

A client or server using a weak random function R() may mean its chosen values for ValidationKey and Secret respectively are able to be guessed.

If the server does not revoke the key on mismatches after the ValidationKey is known to be correct, then an attacker can try multiple values for Counter, increasingly the likelyhood of discovering a match.

If the server revokes the key when the ValidationKey does not match the Validator, this opens a denial of service attack whereby an attacker can potentially revoke a user's keys.

7. References

7.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[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>.

[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>.

7.2. Informative References

[XEP-0388] Cridland, D., "Extensible SASL Profile", August 2017.

Author's Address

Dave Cridland
Surevine Ltd
PO Box 1136
Guildford GU1 9ND
UK

Phone: +44 845 468 1066
Email: dave.cridland@surevine.com