Salted Challenge Response Authentication Mechanism

In cryptography, the **Salted Challenge Response Authentication Mechanism** (**SCRAM**) is a family of modern, password-based challenge–response authentication mechanisms providing authentication of a user to a server. As it is specified for Simple Authentication and Security Layer (SASL), it can be used for password-based logins to services like SMTP and IMAP (e-mail), or XMPP (chat). For XMPP, supporting it is mandatory.[1]

Motivation

Alice wants to log into Bob's server. She needs to prove she is who she claims to be. For solving this authentication problem, Alice and Bob have agreed upon a password, which Alice knows, and which Bob knows how to verify.

Now Alice could send her password over an unencrypted connection to Bob in a clear text form, for him to verify. That would however make the password accessible to Mallory, who is wiretapping the line. Alice and Bob could try to bypass this by encrypting the connection. However, Alice doesn't know whether the encryption was set up by Bob, and not by Mallory by doing a man-in-the-middle attack. Therefore, Alice sends a hashed version of her password instead, like in CRAM-MD5 or DIGEST-MD5. As it is a hash, Mallory doesn't get the password itself. And because the hash is salted with a challenge, Mallory could use it only for one login process. However, Alice wants to give some confidential information to Bob, and she wants to be sure it's Bob and not Mallory.

For solving this, Bob has registered himself to a certificate authority (CA), which signed his certificate. Alice could solely rely on that signature system, but she knows it has weaknesses. To give her additional assurance that there is no man-in-the-middle attack, Bob creates a proof that he knows the password (or a salted hash thereof), and includes his certificate into this proof. This inclusion is called channel binding, as the lower encryption channel is 'bound' to the higher application channel.

Alice then has an authentication of Bob, and Bob has authentication of Alice. Taken together, they have mutual authentication. DIGEST-MD5 already enabled mutual authentication, but it was often incorrectly implemented.[2]

When Mallory runs a man-in-the-middle attack and forges a CA signature, she could retrieve a hash of the password. But she couldn't impersonate Alice even for a single login session, as Alice included into her hash the encryption key of Mallory, resulting in a login-fail from Bob. To make a fully transparent attack, Mallory would need to know the password used by Alice, or the secret encryption key of Bob.

Bob has heard of data breaches of server databases, and he decided that he doesn't want to store the passwords of his users in clear text. He has heard of the CRAM-MD5 and DIGEST-MD5 login schemes, but he knows, for offering these login schemes to his users, he would have to store weakly hashed, un-salted passwords. He doesn't like the idea, and therefore he chooses to demand the passwords in plain text. Then he can hash them with secure hashing schemes like bcrypt, scrypt or PBKDF2, and salt them as he wants. However, then Bob and Alice would still face the problems described above. To solve this problem, they use SCRAM, where Bob can store his password in a salted format, using PBKDF2. During login, Bob sends Alice his salt and the iteration count of the PBKDF2 algorithm, and then Alice uses these to calculate the hashed

password that Bob has in his database. All further calculations in SCRAM base on this value which both know.

Protocol overview

Although all clients and servers have to support the SHA-1 hashing algorithm, SCRAM is, unlike CRAM-MD5 or DIGEST-MD5, independent from the underlying hash function.[3] All hash functions defined by the IANA (https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml) can be used instead. As mentioned in the Motivation section, SCRAM uses the PBKDF2 mechanism, which increases the strength against brute-force attacks, when a data leak has happened on the server. Let `H` be the selected hash function, given by the name of the algorithm advertised by the server and chosen by the client. 'SCRAM-SHA-1' for instance, uses SHA-1 as hash function.

Password-based derived key, or salted password

The client derives a key, or salted password, from the password, a salt, and a number of computational iterations as follows:

```
SaltedPassword = Hi(password, salt, iteration-count) = PBKDF2(HMAC, password, salt, iteration-count, output length of H)
```

Messages

RFC 5802 names four consecutive messages between server and client:

client-first
The *client-first* message consists of a GS2 header (comprising a channel binding flag, and optional name for authorization information), the desired `username`, and a randomly generated client nonce `c-nonce`.
server-first

The server appends to this client nonce its own nonce `s-nonce`, and adds it to the *server-first* message, which also contains a `salt` used by the server for salting the user's password hash, and an iteration count `iteration-count`.
client-final

After that the client sends the *client-final* message containing *channel-binding*, the GS2 header and channel binding data encoded in base64, the concatenation of the client and the server nonce, and the client proof, `proof`.

server-final

The communication closes with the *server-final* message, which contains the server signature, `verifier`.

Proofs

The client and the server prove to each other they have the same `Auth` variable, consisting of:

`Auth = client-first-without-header + , + server-first + , + client-final-without-proof` (concatenated with commas)

More concretely, this takes the form:

`= r=c_nonce,[extensions,]r=c_nonce || s_nonce,s=salt,i=iteration_count,[extensions,]c=base64(channel_flag,[a=authzid],channel_binding),r=c_nonce || s_nonce[,extensions]`

The proofs are calculated as follows:

```
ClientKey = HMAC(SaltedPassword, 'Client Key')
ServerKey = HMAC(SaltedPassword, 'Server Key')
ClientProof = p = ClientKey XOR HMAC(H(ClientKey), Auth)
ServerSignature = v = HMAC(ServerKey, Auth)
```

where the XOR operation is applied to byte strings of the same length, `H(ClientKey)` is a normal hash of `ClientKey`. `'Client Key'` and `'Server Key'` are verbatim strings.

The server can authorize the client by computing `ClientKey` from `ClientProof` and then comparing `H(ClientKey)` with the stored value.

The client can authorize the server by computing and comparing `ServerSignature` directly.

Stored password

The server stores only the username, `salt`, `iteration-count`, `H(ClientKey)`, `ServerKey`. The server has transient access to `ClientKey` as it is recovered from the client proof, having been encrypted with `H(ClientKey)`.

The client needs only the `password`.

Channel binding

The term *channel binding* describes the man-in-the-middle attack prevention strategy to 'bind' an application layer, which provides mutual authentication, to a lower (mostly encryption) layer, ensuring that the endpoints of a connection are the same in both layers. There are two general directions for channel binding: *unique* and *endpoint* channel binding. The first ensures that a specific connection is used, the second that the endpoints are the same.

There are several channel binding types, where every single type has a *channel binding unique prefix*.[4] Every channel binding type specifies the content of the *channel binding data*, which provides unique information over the channel and the endpoints. For instance, for the *tls-server-end-point* channel binding, it is the server's TLS certificate.[5]

An example use case of channel binding with SCRAM as application layer, could be with Transport Layer Security (TLS) as lower layer. TLS protects from passive eavesdropping, as the communication is encrypted. However, if the client doesn't authenticate the server (e.g. by verifying the server's certificate), this doesn't prevent man-in-the-middle attacks. For this, the endpoints need to assure their identities to each other, which can be provided by SCRAM.

The *gs2-cbind-flag* SCRAM variable specifies whether the client supports channel binding or not, or thinks the server doesn't support channel binding, and *c-bind-input* contains the *gs2-cbind-flag* together with the *channel binding unique prefix* and the *channel binding data* themselves.

Channel binding is optional in SCRAM, and the *gs2-cbind-flag* variable prevents from downgrade attacks.

When a server supports channel binding, it adds the character sequence '-PLUS' to the advertised SCRAM algorithm name.

Strengths

- Strong password storage: When implemented in a right way, the server can store the passwords in a salted, iterated hash format, making offline attacks harder, and decreasing the impact of database breaches.[6]
- Simplicity: Implementing SCRAM is easier[7] than DIGEST-MD5.[8]

- International interoperability: the RFC requires UTF-8 to be used for usernames and passwords, unlike CRAM-MD5.[7][9]
- Because only the salted and hashed version of a password is used in the whole login process, and the salt on the server doesn't change, a client storing passwords can store the hashed versions, and not expose the clear text password to attackers. Such hashed versions are bound to one server, which makes this useful on password reuse.[10]

References

1. *"RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core" (http://tools.ietf.org/search/rfc6120#page-162)* .
2. *Kurt Zeilenga (19 May 2010). "SCRAM in LDAP Better Password-based Authentication" (http://people.apache.org/~elecharny/ldapcon/Kurt%20Zeilenga-paper.pdf) (PDF). Retrieved 24 January 2014.*
3. *"RFC 5802, section 4" (http://tools.ietf.org/html/rfc5802#section-4)* .
4. *"RFC 5056 section-7.1" (https://tools.ietf.org/html/rfc5056#section-7.1)* .
5. *"RFC 5929 section 4" (https://tools.ietf.org/html/rfc5929#section-4)* .
6. *"SCRAM: A New Protocol for Password Authentication" (http://www.isode.com/whitepapers/scram.html)* . *19 May 2010. Retrieved 24 January 2014.*
7. *Tobias Markmann (2 December 2009). "Scram DIGEST-MD5!" (http://ayena.de/blog/scram-digest-md5/)* . *Retrieved 23 January 2014.*
8. *"DIGEST-MD5 to historic" (http://tools.ietf.org/html/draft-ietf-kitten-digest-to-historic-04)* .
9. *CRAM-MD5 to Historic (http://tools.ietf.org/html/draft-ietf-sasl-crammd5-to-historic-00)*
10. *Tobias Markmann (9 June 2010). "Sleep Tight at Night Knowing That Your Passwords Are Safe" (http://ayena.de/blog/sleep-tight-at-night-knowing-that-your-passwords-are-safe/)* .

External links

- RFC 5802 (https://datatracker.ietf.org/doc/html/rfc5802) , SCRAM for SASL and GSS-API
- RFC 7677 (https://datatracker.ietf.org/doc/html/rfc7677) , SCRAM-SHA-256 and SCRAM-SHA-256-PLUS
- RFC 7804 (https://datatracker.ietf.org/doc/html/rfc7804) , SCRAM in HTTP
- GNU Network Security Labyrinth (http://josefsson.org/talks/gnu-network-security-labyrinth.pdf) (presentation similar to Motivation section)

Last edited 6 months ago by 84.250.14.116