

A Server Authentication Procedure Proposal

(D.Rotondi@Computer.Org)

Introduction

As stressed by the workshop *Call for Participation*, there is an urgent need of an authentication procedure able to support end-users in correctly identify their service web site. This procedure must be standard, simple enough to be usable and understandable by non-technically savvy end-users and easily deployable.

The next pages sketch a simple authentication procedure having the following objectives:

- minimize the requirements on the end-users so that the usual authentication process is not upset and is minimally changed;
- be simple enough to be implemented both as a browser's internal feature or as a browser plug-in, so that it can be quickly deployed;
- be *safe* even if end-users' data are stolen from the web server(s) or the end-users' data are lost;
- be flexible enough to support different *authentication contents* so that each implementor and/or service provider can tailor the contents to its needs;
- minimize the computation on the server side;
- be able to operate even on an open HTTP connection, so that unprotected sites can still continue to operate;
- minimize the information the user has to provide and the end-user's data than can be intercepted.

The authentication procedure is based on the following assumptions:

- the information for the initial *enrollment phase* are provided to the end-users using an out-of-band, non compromised communication channel (e.g.: standard mail);
- the service enrollment site is *safe*;
- the service X.509 private key(s) is (are) not compromised;
- the end-user's workstation is *safe*.

The described authentication procedure envisages a user's specific *secret share* provided by the user to the web service site, which has the burden to store it and provide it back to the user each time he/she wants to access the service.

Normally on the client machine a complementary *share* is safely stored and the two *shares*, when recombined, demonstrate to the user the identity of the remote service. Depending on the nature of the shares even a client *share-less* configuration is possible.

The actual management of the *shares* on the client machine is a local matter, and is not described in the following, even if *visual cryptography techniques* seems the most appropriate in creating the *secret shares*. A less secure, but normally adequate, technique could also envisage a *service secret share* created as a random selection of pieces within a *puzzle*.

The proposed approach, anyway, can be used virtually with any kind on information (e.g.: text, static and animated pictures, sound, etc.). The nature of the *secret share* provided to the remote server is never disclosed, thus improving the overall process security and deployment flexibility.

The Shared Data

As anticipated, a remote web service is provided with an encrypted copy of the user specific *secret data* and additional info needed both to protect the subsequent interactions and to provide to the client the key recovery information. These items are partially, or totally, updated each time a user completes an authentication.

The *secret data* is created using the following set of information items:

- the *secret share*: the core secret component; when combined with the *share* held by the user (or, exceptionally, by itself), is able to provide evidence of the remote server identity;
- the *Service X.509 Certificate*: used to provide to the server the *session key* generated by the client;
- a *TimeStamp*: the date and time of the end-user last access;
- a *Nonce*: a cryptographically strong random number.

The way the *secret shares* are created is a local matter and is not further investigated. The above data items are combined together to create the following *UsrSecretData*:

$$UsrSecretData = \{Secret_MIMEType, Secret_Share, ServiceX509Cert, TimeStamp, Nonce_r\}$$

To protect the *secret data* (and all client-server data exchanges) the following items and functions are required:

- a *UserID*: an *identifier* through which the service identify the user (e.g.: the user *Login ID*);
- a *UsrServiceName*: a data generated by the remote server, essentially used to improve the randomness of the generated keys. It must satisfy cryptographic requirements (length, randomness, etc.), even if it could be used to improve the information provided to the end-user during the authentication process. For example this item could be structured into a relatively small component with end-user's useful info (i.e.: *Acme Inc. Online Store*) and a completely random and user specific component;
- a *UserPassword*: generated and owned by the end user and never transmitted to the remote server. Password requirements and constraints are local matters. To improve the strength of the user password (and the bad habit to use the same password for many different services and purposes), it is never used alone (as described in the following);
- a *secure hash function* ($H(x)$): selected so that its digest's length is equal to the symmetric encryption key length (e.g.: SHA256 and AES256);
- a symmetric block cipher algorithm ($E_s(data; Key)$);
- an asymmetric key encryption algorithm ($E_a(data; Key)$);
- a *large prime number* N and a *primitive root* g as defined in the **RFC2945** "The SRP Authentication and Key Exchange System".

Given these definitions, the proposed authentication procedure makes use of the following additional items:

- a random *Salt* used, on the client side, to generate the *session key*;
- a *session key* (K_{ses}) calculated as follows (the symbol "|" indicates concatenation):

$$x = H(Salt \parallel H(UserID \parallel ":" \parallel Password))$$

$$K_{ses} = H(g^x \bmod N)$$

- a *user secret key* (K_{us}) calculated as:

$$y = H(Salt \parallel UsrServiceName \parallel H(UserID \parallel ":" \parallel Password))$$

$$K_{us} = H(g^y \bmod N)$$

- a symmetrically encrypted version of the *UsrSecretData*:

$$UsrEncryptedData = E_s(UsrSecretData; K_{us})$$

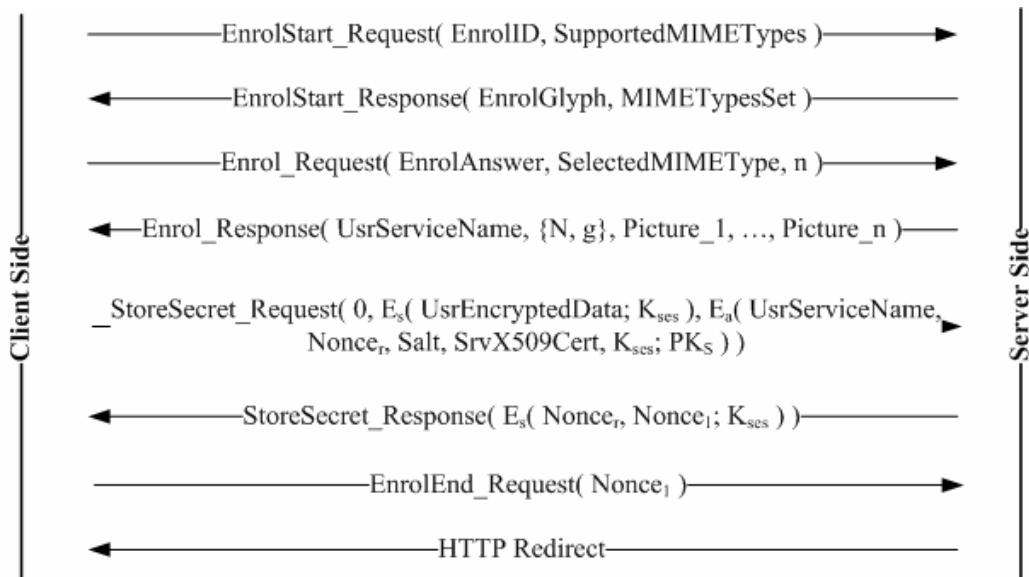
The Enrollment Phase

The enrollment phase requires the out-of-band provision to the end user of the following items:

- an *EnrollID*: a random identifier uniquely associated to the end user. It is required only for the *Enrollment Phase*;
- a predefined, but variable, server response (called *EnrolGlyph* in the figure). This response must be used both to provide to the end user evidence he/she is contacting the right server and to *challenge* the end user (for example using a combination of a user specific picture and symbols selected among a predefined set as indicated in the out-of-band preliminary communication);
- a predefined, but *EnrolGlyph* dependent, user answer (*EnrolAnswer*) that must be provided in order to proceed.

All enrollment phase's data exchanges must be carried out using an SSL/TLS protected session.

The following picture reports the proposed data exchanges (*the client/server must immediately abort the connection and discard all data if a wrong answer is received*):



The first request just provides to the server the user's *EnrollID* and specify the kinds of data (*SupportedMIMETypes*) the client is able to use as secret. The server response challenges the end-users and, at the same time, provides the supported MIME types as a subset of the end-user's indication. The subsequent client request replies to the server challenge and specify both the MIME type it has select (*SelectedMIMETYPE*) and how many data samples (*n*) the client wants. The server response provides the *UsrServiceName*, defined above, the server specific large prime and primitive root ($\{N, g\}$) and the required samples (*Picture_i*).

The client now has all the elements to create the *shares* (the client is not constrained to use any of the *Picture_i*, nor to use the declared MIME type) and all other required items.

To this end, the client generates the *shares* interacting with the end-user (for example presenting a *puzzle* and giving the opportunity to select some *puzzle pieces*), two random numbers (*Salt* and *Nonce_r*), acquires the SSL/TLS X.509 certificate and creates both the *session* and *user keys* and the *UsrEncryptedData*.

Finally, it calculates the following two values (note that A_1 brings a double encrypted data):

$$[A_1]: E_s(UsrEncryptedData; K_{ses}), [A_2]: E_a(UsrServiceName, Nonce_r, Salt, SrvX509Cert, K_{ses}; PK_S)$$

using ad hoc items encoding (for example as defined in the W3C Recommendation "**XML Encryption Syntax and Processing**") and the server X.509 *public key*.

The client, finally, transmits these elements to the server, which decrypts the second value, checks that the client made use of the correct parameters (*UsrServiceName*, *SrvX509Cert*) and recovers the *session key*, the *Salt* and *Nonce_r*.

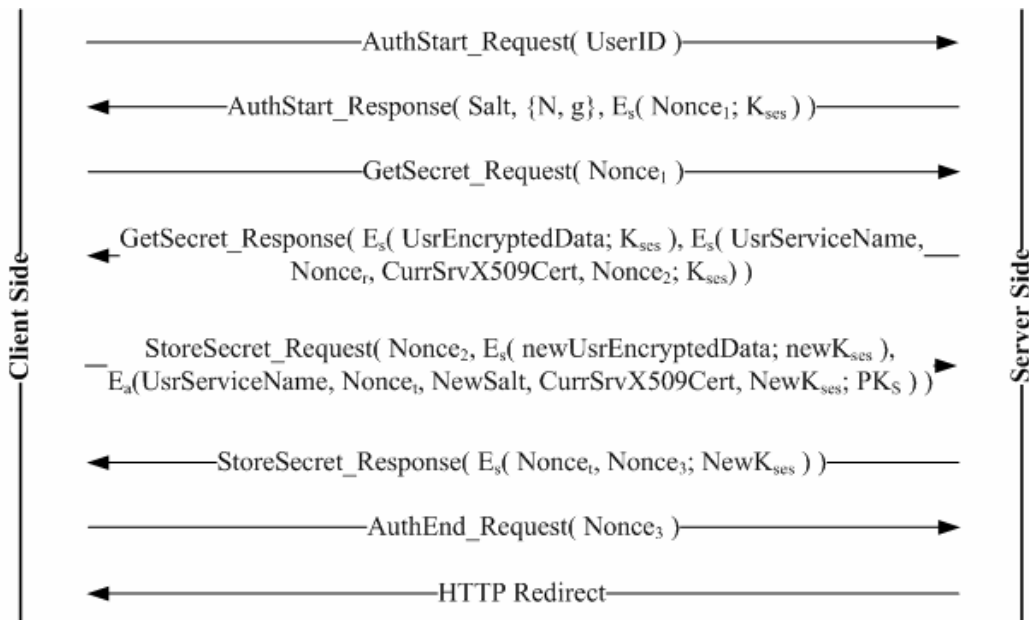
The server stores the two values A_1 and A_2 in its database as end-user associated data and replies to the client. This response challenges the client providing a further (*session key* encrypted) random value (*Nonce₁*).

The *enrollment phase* completes after the final redirect as depicted in the previous figure.

The Authentication Phase

The authentication phase must be considered as an integration of the normal login phase, rather than a substitution. It must precede the login phase and, therefore, must not necessarily require an SSL/TLS protected session.

The following picture reports the proposed data exchanges (*the client/server must immediately abort the connection and discard all data if a wrong answer is received*):



The client sends the *userID* (i.e.: the *login ID*) to the server receiving back a set of parameters, through which the client can reconstruct the *key* used in the last session, and an encrypted challenge.

If the user provides the right *UserID* and *Password* the client can recalculate the *session key*, decrypt the challenge and reply to the server.

If the challenge response is correct, the server recovers the end-user's A_1 and A_2 values from its database, computes $[A_3]: E_s(UsrServiceName, Nonce_r, CurrSrvX509Cert, Nonce_2; K_{ses})$ and sends A_1 and A_3 .

The client, now, has all the elements to recover the *user secret key* (K_{us}) and decrypt A_1 . The client must check the elements in A_1 are congruent with the one provided in A_3 . If the server X.509 certificate has been changed since last session, the client will request the certificate chains starting from the one in A_1 till the one presented in the *GetSecret_Response* (these requests/responses are not depicted in the previous figure).

After completing these checks, the client must interact with the end-user (and possibly recover the *local secret share*) to give evidence of the server identity and explicitly ask the user authorization to proceed or abort the connection.

If the user recognizes the secret, the client must update the information in the *UsrSecretData*, generate a new *user secret key* and *session key*, calculate the new A_1 and A_2 and send them to the server as indicated in the previous figure (the *StoreSecret_Request* will be accepted by the server if it provides the right *Nonce₂*).

The server performs the same checks and operations already described in the *enrollment phase* and replies with a new challenge to the client.

The client, finally, must decrypt the final server challenge and submit a final request that will end the authentication process redirecting the client browser to the usual login server page.

As it is clear, the *authentication phase* only exposes the *UserID* and, if successful, forces an update of all the relevant data (*user secret data* stored on the server side, *session* and *user keys*).

Further Requirements

The indicated phases will normally be activated by the end-user selecting some link/object in the web service page. To simplify the start of these phases, especially when managed by a browser's plug-in, the definition of ad-hoc *MIME-Types* would be recommended.

On the server side, instead, A_1 and A_2 **must** be stored as received (i.e. encrypted) so that losing (or stealing) users' related data does not compromise any users' information (unless the server X.509 private key is, at the same time, compromised).

Weaknesses

The proposed procedure is not immune from weaknesses. In particular it strongly depends on the secrecy of the *UserPassword* and of the server X.509 private key.

Therefore, it remains on the responsibility of the users and service managers to avoid disclosure/compromise of these data.

An additional threat is represented by the use of the same password both for the *authentication phase* and *service login phase*. Indeed, if the password is not safely stored on the service side, an attacker could easily acquire all user's related data and spoof the user.

The *enrollment phase*, finally, is not immune from *man-in-the-middle* attack.

These weaknesses, anyway, do not seem so serious especially taking into account the anti-phishing goal of the procedure.

Conclusions

Even if not immune from deficiencies and weaknesses, this proposal aims at speeding up the definition of a standardized anti-phishing procedures that can be quickly implemented and deployed and has minimal impacts on the existing web services and operational habits and procedures.

The proposal needs to be further refined (e.g.: minimal set of *MIMETypes* browsers and servers must support, supported encryption/hashing algorithms, encodings) and structured. It must, therefore, be considered just as a contribution to the overall discussion.