FHA: Institut für Informatik

Implementation Notes of the Security Specs for IVO

Carlo U. Nicola April, 26th 2005

1 Signing on to the IVO

Each user is a member of a community (on its definition and size see [1]). The user signs on by sending a user ID (IVOID) and a password to a local program that acts as proxy for the user (Ag^U where: $U = IVOID$). The program generates (RSA, DSA-algorithm) an asymmetric key pair K_S^U, K_{Pu}^U. and then passes K_{Pu}^U (the public key) and the user ID to the SO (Sign On) program of the community server. The server generates a X509 certificate, signs it with its CA and sends the certificate back to Ag^U, encrypted symmetrically with the password of the user. Of course the last step hinders not only that the password wanders (perhaps in clear!) on the community's network but more important that only the legitimate user can extract the certificate.

It is important to remember that the SO program should maintain a complete list of all the session certificates that are currently in use. A line in this list consists of: K_{Pu}^U, IVOID, $cert^U$. A timeout mechanism or better an expiration time for certificates assures that older certificates not in use anymore are purged from the list.

The software agent Ag^U can be either a program running on the user's computer or a servlet embedded in a web portal.

2 Authenticating to the service

Each time the user requests a service, Ag^U interacts with that specific service with a new set of keys and a new certificate on behalf of the user. The actual authentication's steps are (*verbatim* from the proposal of Guy Rixon [1]):

1. The service establishes that the request comes from an entity holding a certain private key.
2. The service determines the individual account to which the private key is bound.
3. The service determines the community that defines the individual account.
4. The service determines the group or groups associated with the individual account.

In this protocol the secret key K_S^U must be sent to the service through a secure channel (TLS). Since K_S^U should never leave the main store of the user's computer, I would implement the first step in the authentication, as follows:

1. The software proxy of the IVO user (Ag^U) sends on an unsecure channel to the service (Srv):
$$e_{K_S^U}(nonce, Hash(nonce)) + cert^U \longrightarrow Srv$$
The reason why the nonce and the hash of the nonce are sent both encrypted, lies in the construction of the nonce. Even if we assume the nonce is a random number (probably produced by `SecureRandom.getInstance("SHA1PRNG")`) it is better to hide this fact to prevent future attacks (most random number series produced by such algorithms are quite predictable . . .). In our case a simple random number will not do: the nonce should contain a minimal information about the user and the community he is coming from, therefore the case for encryption is more stringent. The decoding process is not too difficult if one keeps in mind that in all practical hash functions the hash values have a constant byte size.

2. The service verifies the user's credentials and message integrity by decrypting the message with K_{Pu} (extracted from the cert);

3. The service gets the DN of the user's community and the address of the attribute server of that community from the cert.

4. The service looks up the group attributes of the IVO user from the attribute server (shibboleth SOAP service)

The clear advantages of this protocol are:

1. All information with the exception of that in the last step, flows securely on unsecured channels;

2. Only one copy of K_S^U is present at any given time in the system, namely on the local user's (or web portal) machine.

3 Delegation of credentials

Sometimes a service can only be get at through subsidiary agents. The question then arises, how to delegate the user session credentials in a secure manner to the subsidiary agent. It is important for the delegation mechanism to be modeled after the authentication protocol I discussed in the previous section. The reason is obvious: we can thus reuse in the concrete implementation the same code. The stages are as follows :

1.1 Ag^{U} : $e_{K_S^U}(nonce, hash(nonce))$ (for details of the *nonce* construction see below).

1.2 Ag^{U} : $e_{K_S^U}(nonce, hash(nonce))$, $U \longrightarrow Ag^{Del_U}$.

2.1 Ag^{Del_U} : $K_S^{Del_U}, K_{Pu}^{Del_U}$

2.2 Ag^{Del_U} : $msg, hash(msg)$ where: $msg = U, K_{Pu}^{Del_U}$, $e_{K_S^U}(nonce, hash(nonce))$, Del_{ID} $\longrightarrow$ SO.

3.1 SO checks: $e_{K_S^U}(nonce, hash(nonce))$ (it maintains a list with all the relevant parameters in this case it uses K_{Pu}^{U})

3.2 if verify OK:

3.3 SO signs: new $cert^{Del_U}$ containing $K_{Pu}^{Del_U}$.

3.4 SO: $cert^{Del_U} \longrightarrow Ag^{Del_U}$.

3.5 else verify NOK: Error msg to community manager.

The nonce should contain some information that can identify the delegating agent. For example:

```
#IVOID#DN Community#Usergroup#Random string#Time stamp#
```

4 Security of SOAP messages

5 Implementation details

I now discuss a concrete implementation of this design as a proof of concept.

6 Literature

[1] Guy Rixon, April, 18th 2005:

http://software.astrogrid.org/SoftwareArchitectureFor2005.html