



WS-Security 101

Created by Charles Skinner

www.eherenow.com

3/14/2006

Messages have been sent “securely” for years without the use of WS-Security. The data was protected during the transport over HTTP by using SSL as the encryption-based technology. The message was authenticated by storing some type of identifying information within the payload of the message. Since the payload was encrypted, there was little concern about data changing during the transport.

It was discovered, however, that SSL methodology significantly limits messaging functionality. When using SSL, an application must decrypt the entire message before it can read any part of the message. This means that in order to perform services against a message, such as authentication or complex routing, the system will have to suffer a processor cost of decrypting the entire message. Moreover, there is an increased risk because the entire message is decrypted and sensitive information is available to parts of the application that have no need for the data. In addition, SSL does not play well when sending a message over non-HTTP protocols such as MSMQ, TCP/IP Sockets or even email.

It is clear that a better security solution must be established in order to satisfy today’s security requirements.

Introducing WS-Security

WS-Security is a standard developed to allow storage of security tokens in a SOAP message. These security tokens are stored in a new security SOAP header element. WS-Security leverages established XML Security standards (*i.e.*, *XML Encryption and XML Signature*; cryptographic algorithms, such as *SHA-1* and *Triple-DES*; and authentication mechanisms, such as *X.509* and *Kerberos*) to accomplish three major tasks: security token propagation, message integrity, and message confidentiality. Its power is in its flexibility. WS-Security provides support for multiple security tokens, multiple trust domains, multiple signature formats, and multiple encryption technologies.

All of the major web service development environments have provided tools to implement WS-Security with little effort (e.g., WSE from Microsoft and Apache's WSS4J).

Best Practices

It would be nice if all of our security concerns could all be solved by using this new standard. Unfortunately, as with any standard, it is only as good as how they are implemented.

Here are some basic rules of thumb that will help keep you on the right path:

Authentication of the Server

How horrible would it be if your clients were sending their entire request to a spoofed server?

In order to prevent this, the client should require the server to prove knowledge of its private key. To accomplish this you can use X.509 certificates with SSL or WS-Security.

Do NOT expose Signatures or encryption generated by the Username Token

If you make this information available to hackers, then you make yourself more vulnerable to offline dictionary attacks. You should use strong session key established during server authentication. If you sign anything with the Username token, ensure that the signature itself is encrypted with that same strong session key.

User Token Passwords should be Hashed

Use the simple SHA-1 hash on the password before sending the information across the wire. This will prevent exposure of the client's password as the Username token flows through the pipeline.

Protect against replay attacks

Although sending hashed passwords solves the problem of a hacker reading the passwords, it does not prevent a hacker from using the hash to send additional requests if the message is intercepted. To protect against such an attack, you will have to verify that each message is new and unique. To accomplish this, you should use a nonce (unique message id) and a timestamp in the message to compare against a cache. The passwords SHA1 hash should be calculated with the nonce and time stamp included to prevent spoofing of the nonce and the timestamp.

Example:

```
Password_Digest = Base64 (SHA-1(nonce + created + Password))
```

Note: One common problem with this approach is that the server will have to know the password to rehash for the compare in order to verify the password. One method is to hash the password in the database with a Scope URI. The client will also know the Scope URI and will perform the same hash before sending the password.

Server Side:

```
Database Password = Base64 (SHA-1(Scope URI + Password))
```

Client Side:

```
Password Digest = Base64 (SHA-1(nonce + created + Base64 (SHA-1  
(Scope URI + Password))))
```

Do NOT store passwords in plain text

If you store passwords in plain text, then a hacker that gains access to the server would have access to all your user passwords. Since many users use the same passwords for all accounts, the result of such an attack can be far reaching and devastating to your user.

You should store the passwords into your data store as a hashed password verifier to prevent this nightmare.