Build secure WS-Resources with WSRF::Lite and WS-Security

Skill Level: Intermediate

Mark McKeown (mark.mckeown@manchester.ac.uk)
Grid Architect
University of Manchester

Stephan Zasada (stefan@zasada.co.uk)
Grid Computing Researcher
University College London

04 Apr 2006

WSRF::Lite is an implementation of the Web Services Resource Framework (WSRF) in Perl. Learn how to secure Web services and WS-Resources built with WSRF::Lite, using two approaches: Transport Layer Security (TLS) and digitally signing Simple Object Access Protocol (SOAP) messages.

Section 1. Before you start

About this tutorial

This tutorial shows Web services and grid developers how to build secure WS-Resources using WSRF::Lite, a Perl implementation of the WSRF. It introduces WS-Security and TLS and discusses the advantages and disadvantages of each. In the tutorial, you'll use both approaches to build secure WSRF::Lite WS-Resources.

Objectives

In this tutorial, you will learn the options for securing WSRF::Lite WS-Resources using WS-Security and TLS. Explore the types of security threats you face when building Web services and how to counter them using the features of WS-Security and TLS. On completing the tutorial, you'll know how to build a secure WSRF::Lite WS-Resource using digitally signed SOAP messages, TLS, or a combination of the two.

Prerequisites

Basic knowledge of Perl will help you complete the tasks. And you should also have a basic understanding of Web services technology and Public Key Infrastructure (PKI). This tutorial is a follow-up to the IBM developerWorks tutorial "Build WS-Resources with WSRF::Lite." If you aren't familiar with WSRF, WS-Addressing, or WSRF::Lite, you'll find it useful to complete that tutorial first.

System requirements

To run the examples, install the following Perl modules on a Linux® system:

- SOAP::Lite version 0.65 or higher
- XML::DOM
- DateTime
- DateTime::Format::Epoch
- DateTime::Format::W3CDTF
- Crypt::OpenSSL::RSA
- MIME::Base64
- Digest::SHA1
- Crypt::OpenSSL::X509
- XML::CanonicalizeXML

You also need an X.509 digital certificate. If you don't have one, create your own using tools such as SimpleCA (see Resources for details).

Section 2. Background

What is WSRF?

The WSRF was introduced in January 2004 to provide a standard way to manage state through a Web services interface. WSRF is actually a group of associated specifications:

- WS-Resource

- WS-ResourceProperties
- WS-ResourceLifetimes
- WS-ServiceGroups
- WS-BaseFaults

The key concept in WSRF is the WS-Resource, an entity that maintains state between calls made to it and that can be accessed through a Web services interface. A client can query and modify the state of a WS-Resource through a set of operations defined in the WS-ResourceProperties specification. The lifetime of the WS-Resource can be controlled by a set of operations defined in the WS-ResourceLifetimes specification. WS-ServiceGroups defines a mechanism for building registries of WS-Resources that clients can search for particular WS-Resources. WS-BaseFaults provides a standard way of reporting faults from WS-Resources to clients. For more information, see Resources.

What is WS-Security?

WS-Security is a set of Organization for the Advancement of Structured Information Standards (OASIS) standards for enhancing SOAP messages to ensure message integrity and confidentiality. It provides mechanisms for attaching security tokens to SOAP messages (for example, X.509 certificates) and for signing or encrypting parts of a SOAP message. By signing part of a SOAP message and attaching a security token to validate the signature, the message receiver can verify that a message sent by a client hasn't been tampered with in transit and that the message is from whom it claims to be from.

WS-Security provides message-level security, in contrast to TLS. With TLS, a secure connection is created between the two parties before any application data is sent between them. TLS has the advantage of being simple to use. For example, a Web services provider can switch from using HTTP to HTTPS to create a secure environment for exchanging SOAP messages. The disadvantage of TLS is that it doesn't allow SOAP messages to pass through intermediaries; it only provides point-to-point security. If you're designing a system in which SOAP messages will pass through a number of intermediary sites for processing, you'll need WS-Security.

WS-Security allows sensitive parts of a SOAP message to be digitally signed or encrypted while leaving the rest of the message as machine- and human-readable XML. WS-Security also lets you attach a security token -- such as an X.509 certificate -- to a SOAP message. The token can be used for sender validation. If you need to send a SOAP message through a number of intermediaries but want to make sure that none of the intermediaries tamper with the message, you should digitally sign the SOAP message -- by using an X.509 certificate, for example. Whomever ultimately receives the message can use the signature to verify that the message hasn't been tampered with and can check that it came from the person with the X.509 certificate. The receiver can also store the digitally signed message as proof that the person with the X.509 certificate sent it. This isn't possible with

TLS.

The WS-Security specification defines the procedures for signing or encrypting part of a SOAP message. However, it gives limited guidance on what parts of the message should be signed or encrypted, which is left to the Web services developers, who should understand the security requirements of their application.

What is WSRF::Lite?

WSRF::Lite is an implementation of WSRF in Perl. It's built on SOAP::Lite, the popular Web services tool kit for Perl. WSRF::Lite provides support for the following specifications:

- WS-Addressing
- WS-ResourceProperties
- WS-ResourceLifetime
- WS-ServiceGroups
- WS-BaseFaults
- WS-Security

WSRF::Lite provides support for building secure WS-Resources using HTTPS or with digitally signed SOAP messages. For more information about WSRF::Lite, see Resources.

Section 3. Securing Web services

Understanding your application's security requirements

To use WS-Security effectively, you must understand the security requirements of your Web services application. Here are questions you need to address:

- Does the transport protocol provide any security?
- Will SOAP intermediaries be processing the message?
- Does the message or part of the message need to be encrypted?
- Does the message need to be recorded for repudiation purposes?
- What happens if someone steals a message and resends it?

- What happens if someone steals a message destined for one service and sends it to another?

In the simplest case of a client-server application that uses HTTP as a transport protocol, switching to HTTPS can be the best option. Doing so provides encryption so no one can read the message and also protects against replay attacks (in which an attacker steals a message and resends it). HTTPS also supports mutual authentication in that the client and server mutually authenticate with each other using their X.509 certificates before exchanging application data.

In a case where a SOAP message must pass through an intermediary, but you don't want it to read part of the message or modify the message, you might encrypt the sensitive part of it and sign all of the message. It's possible to combine TLS and message-level security -- to send signed messages over a TLS connection, for example. For detailed discussions of the threats faced by Web services and the countermeasures available, check out Resources.

Signing a SOAP message

WS-Security is built on a number of XML specifications: Canonical XML Version 1.0, Exclusive Canonicalization Version 1.0, XML-Signature Syntax and Processing, and XML-Encryption Syntax and Processing. The first two deal with converting XML into a standard canonical form. By canonicalizing two pieces of XML, you can compare them byte by byte for sameness. This is important for creating and validating digital signatures. The second two detail the procedures for digitally signing and encrypting XML. WS-Security builds on these specifications by defining how to integrate them with SOAP.

The first step in signing a SOAP message is to canonicalize the elements of the message to be signed. Because the form and structure of a SOAP message can legitimately change as it passes from the sender to the receiver, it's important that the XML can be converted to a standard form that can be compared for sameness.

Once the elements have been canonicalized, a digital hash is taken of the canonicalized elements. If any change is made to the elements, the hash of the elements will be different, and the receiver will be able to detect that they have been tampered with. After the hash has been taken of all the canonicalized elements, a new `SignedInfo` element is created that includes all the hashes from the canonicalized elements. This is the piece of XML that will be digitally signed. The `SignedInfo` is canonicalized, a hash is taken of its canonical form, and the hash is signed (using an X.509 certificate, for example). To allow the receiver to validate the signature, the sender includes the public part of the X.509 certificate in the message. WS-Security defines how you include the public part of an X.509 certificate in a SOAP message.

On receiving a signed SOAP message, the receiver checks to be sure the X.509 certificate included in the SOAP message is valid and acceptable. Next, the receiver uses the certificate to check the signature in the message. If the signature is correct, the receiver canonicalizes all the elements in the message the client has included in the signature, takes a hash of the canonical forms, and compares them to the hashes included in the `SignedInfo` element. If any of the hashes don't match, the receiver should reject the message.

For a detailed discussion of the process of signing a SOAP message, see Resources.

Section 4. Using TLS with WSRF::Lite

Installing WSRF::Lite

This is a brief introduction to installing WSRF::Lite and running a WSRF::Lite container. It's sufficient for the purposes of this tutorial, but for a more in-depth introduction, complete the "Build WS-Resources with WSRF::Lite" tutorial.

The first step to deploying WSRF::Lite is to install the necessary Perl modules listed in System requirements. You can do so using Comprehensive Perl Archive Network (CPAN) or by downloading the packages and installing manually using the following commands:

```
> perl Makefile.PL
> make
> make test
> make install
```

Next, download the latest version of WSRF::Lite. After the bundle has been untarred, you should see the following files and directories in the root directory of the WSRF::Lite distribution:

- client-scripts
- CPAN_help
- LICENSE
- MANIFEST
- README
- t
- TODO
- Container.pl
- lib
- Makefile.PL

- modules
- SContainer.pl
- test
- WSRF

You can install WSRF::Lite into the system Perl libraries by running `Perl Makefile.PL, make, make test, make install` or you can run the WSRF::Lite Container and examples from the WSRF::Lite distribution directory.

To run a WSRF::Lite Container -- the item that manages the WS-Resources -- create the directories /tmp/wsrf and /tmp/wsrf/data. You must also set the environmental variable `WSRF_MODULES` to point to the modules directory in your WSRF::Lite distribution, where the code for the sample WS-Resources resides. The sample WS-Resources are based on a counter. You can create, update, and query the counter WS-Resources using the sample scripts provided in the client-scripts directory. Once you've created the directories and set the environmental variable, start a WSRF::Lite Container by running the Container.pl script. The SContainer is the secure version of the container and uses HTTPS; the next section discusses how to configure it. The test directory contains a number of testing scripts you can use to check that the container is running correctly. The README file in the WSRF::Lite distribution provides detailed instructions for installing, testing, and using the WSRF::Lite distribution.

Using the WSRF::Lite SContainer

The WSRF::Lite SContainer uses HTTPS to provide TLS for the WS-Resources it hosts. To use the SContainer, you need an X.509 host certificate with an unencrypted private key. Whenever a client connects to the SContainer, it will be presented with this certificate. All communication between the client and the SContainer will be encrypted. To use the WSRF::Lite SContainer, you must modify the SContainer.pl script to point to the host X.509 certificate, and you must set the security policy. The relevant code is shown in Listing 1.

Listing 1. The SContainer.pl script

```
....
# Create a Secure Socket using the host certificate
my $d = HTTP::Daemon::SSL->new(
  LocalPort => $port,   #port to listen on

  Listen => SOMAXCON,   #Queue size for listen

  Reuse => 1,

  SSL_cert_file => '/home/zzcgumk/hostcert.pem',
  # public part of host X.509

  SSL_key_file => '/home/zzcgumk/hostkey.pem',
  # private part of host X.509

  SSL_ca_path => '/etc/grid-security/certificates/',
  # CA directory

  SSL_ca_file => '/etc/grid-security/certificates/01621954.0',
  #CA file for X.509

  SSL_verify_mode => 0x01 | 0x02 | 0x04
  # Authentication policy

) || do { print "Socket Error: Cannot create Socket\n"; exit; };
        ....
```

`SSL_cert_file` points to the file containing the public part of the host's X.509 certificate, and `SSL_key_file` points to the certificate's private key (which should be unencrypted). `SSL_ca_path` points to the directory where you keep the Certificate Authority (CA) certificates of all the CAs you accept; `SSL_ca_file` points to the CA certificate of the CA that issued the host certificate.

`SSL_verify_mode` sets the policy of the SContainer. The default (0x00) performs no authentication of the client. You can combine 0x01 (verify client), 0x02 (fail verification if no client certificate is presented), and 0x04 (verify client once) to change the default. The SContainer in Listing 1 is configured for mutual authentication. If the client doesn't present a valid X.509 certificate, the SContainer will refuse access.

Once an SContainer has validated a client certificate, it sets the environmental variable `SSL_CLIENT_DN` to the Distinguished Name (DN) of the client certificate and the environmental variable `SSL_ISSUER_DN` to the DN of the CA that issued the client certificate. This is the same approach the Apache Web server uses to pass client certificate information to a CGI script. The WS-Resource code can retrieve the client certificate information through the `$ENV{SSL_CLIENT_DN}` and `$ENV{SSL_ISSUER_DN}` variables.

Connecting to the SContainer

The sample client scripts provided with WSRF::Lite, in the client-scripts directory, need to be modified to point to the client's X.509 certificate. At the start of each script are the following lines:

```
#need to point to users certificates - these are only used
#if https protocol is being used.

# Directory where CA certificates are kept
$ENV{HTTPS_CA_DIR} = "/etc/grid-security/certificates/";

# Public part of the client X.509 certificate
$ENV{HTTPS_CERT_FILE} = $ENV{HOME}."/.globus/usercert.pem";

# Private part of the client X.509 certificate
$ENV{HTTPS_KEY_FILE}  = $ENV{HOME}."/.globus/userkey.pem";
```

`$ENV{HTTPS_CA_DIR}` points to the directory where the client keeps the CA certificates of trusted CAs. If the service presents a certificate that isn't signed by one of these CAs, the client aborts the communication. `$ENV{HTTPS_CERT_FILE}` points to the public part of the client's X.509 certificate, and `$ENV{HTTPS_KEY_FILE}` points to the certificate's private key. If the private key is encrypted, the client prompts you for the password to unlock the private key when it attempts to make the connection.

Given an HTTPS URI as the target service, the script automatically tries to use the identified certificate to mutually authenticate with the service. If the lines are commented out and the script is given an HTTPS URI, it doesn't attempt mutual authentication with the service. Only the server identifies itself to the client. If the service requires mutual authentication, it denies access to the client.

If the SContainer and client script have been configured properly, the DN of the client certificate appears in the SContainer output when you access a WS-Resource.

Section 5. Signing a SOAP message with WSRF::Lite

Elements that WSRF::Lite signs

This section explains how to sign a SOAP message using WSRF::Lite. By default, WSRF::Lite signs the following elements of a SOAP message (`wsa` elements are WS-Addressing SOAP headers, `soap` elements are SOAP elements and `wsse` elements are WS-Security SOAP headers):

wsa:To
: Destination of the SOAP message. Signing this prevents attackers from stealing the message and sending it to a different service.

wsa:From
: Where the message came from.

wsa:MessageID
: Unique identifier for the message. A service should reject any message that reuses a `MessageID` to prevent replay attacks.

wsa:Action
: Intended semantics of the message.

wsa:ReplyTo
: IWhere to send the response. Signing this prevents an attacker from changing where the response is sent.

wsa:RelatesTo
: wsa:MessageID of the original message, if this message is a response to a previous message.

wsse:Timestamp
: Length of time the message is valid.

wsse:BinarySecurityToken
: Security token used to sign the message; included in the signature. This prevents attackers from replacing the security token.

soap:Body
: Body of the message, which holds the application payload.

`wsa:MessageID` is signed to prevent replay attacks, in which an attacker steals a message and resends it to the service. This would require a service to record the `wsa:MessageID` of every message it ever processed. To address this issue, WS-Security introduces the `wsse:Timestamp` element. `wsse:Timestamp` declares the time the message is created and when it expires. If a service receives a message after its expiration time, it should not process it. A service only needs to store the `wsa:MessageID` of messages it has received that have not expired; it can discard the `wsa:MessageIDs` of messages that have expired. (Note that a clock skew between a client and server can cause a server to drop all messages received from that client, and this bug can be difficult to detect.)

It's important to note that to create secure SOAP messages using digital signatures, you must use WS-Addressing and sign the WS-Addressing elements.

Counter WS-Resource example

This section explains how to modify one of the sample WS-Resources provided with WSRF::Lite so it signs the SOAP responses it sends back to the client. For example, uncomment the following lines in the sample file modules/WSRF/Counter.pm:

```
# If these $ENV are set the SOAP message will be signed
# Points to the public key of the X509 certificate
$ENV{HTTPS_CERT_FILE} = "/home/zzcgumk/hostcert.pem";

# Points to the private key of the cert - must be unencrypted
$ENV{HTTPS_KEY_FILE}  = "/home/zzcgumk/hostkey.pem";

# Tells WSRF::Lite to sign the message with the X.509 certificate
$ENV{WSS_SIGN} = 'true';
```

`$ENV{HTTPS_CERT_FILE}` points to the public part of the X.509 certificate you want to use to sign the message, and `$ENV{HTTPS_KEY_FILE}` points to the certificate's private key (which should be unencrypted). To start, host the WS-Resource in the standard container.

Create a new Counter WS-Resource using the wsrf_createCounterResource.pl script from the client-scripts directory. Then, to check that the SOAP the WS-Resource returns is signed, invoke an operation on the WS-Resource using one of the client scripts from the client-scripts directory. The script will indicate whether the SOAP response was signed and which parts of the response were included in the signature.

Next, add the code from Listing 2 below to the sample Counter WS-Resource so it checks incoming messages and verifies whether they have been signed. Listing 2 is a modified version of the `add` operation from modules/WSRF/Counter.pm.

Listing 2. The add operation from the Counter WS-Resource

```
# add a value to the count
sub add {
  my $envelope = pop @_;     #get the SOAP envelope
# this is added to check if the SOAP message has been signed -
# if the verify dies because the message is not signed we catch it
# with the eval
eval{
  # the WSRF::WSS::verify checks for a signature,
  # if the message is not signed or
  # not signed correctly verify will die.
  # verify returns a hash containing the
  # the X509 used to sign the message, the timestamp
  # and the names of the elements signed in the message
  my %results = WSRF::WSS::verify($envelope);

  # The X509 certificate that signed the message
  print "X509 certificate=\n$results{X509}\n" if $results{X509};

  #print the name of each element that is signed
  foreach my $key ( keys %{$results{PartsSigned}} )
  {
    print "\"$key\" of message is signed.\n";
  }

  #print the creation and expiration time of the message
  print "Message Created at \"$results{Created}\".\n"
      if $results{Created};
  print "Message Expires at \"$results{Expires}\".\n"
      if $results{Expires};

};
if ($@) { print "SOAP Message not signed: $@\n"; }
  #catch if verify dies


  my ($class, $val) = @_;   #get the params to the operation

  # increment the counter
  $WSRF::WSRP::ResourceProperties{count} =
     $WSRF::WSRP::ResourceProperties{count} + $val;

  # return the new value of the counter
  return WSRF::Header::header($envelope),
         $WSRF::WSRP::ResourceProperties{count};

}
```

The `WSS::WSS::verify` function checks the SOAP message for a digital signature. The function stops if the SOAP message hasn't been signed or is signed incorrectly. In this example, if the message isn't signed correctly, the exception is caught and a note is made in the WS-Resource log. Output from `print` statements in a WS-Resource module go to a log file in the WSRF::Lite logs directory, modules/logs. If the message has been signed, `WSS::WSS::verify` includes details of the signature in the `%results` hash. In this example, the details of the signature are sent to the log file.

`WSS::WSS::verify` only checks whether the message has been signed correctly. It doesn't check whether the X.509 is acceptable, whether the message has expired, or whether the correct parts of the message have been signed. It's left to you to validate these items according to the service's security policy.

Client code

The example client scripts in the client-scripts directory check whether a SOAP response from a WS-Resource has been signed. If it has been signed, the script prints details of the signature, such as the message's expiration time, which parts were signed, and the X.509 certificate used to sign the message.

To make the client scripts sign SOAP messages, uncomment the following line in the scripts:

```
#$ENV{WSS_SIGN} = 'true';
```

The client scripts uses the same X.509 certificate shown earlier to sign the message. The private key of the certificate must be unencrypted for signing messages.

Once you have exchanged signed SOAP messages between the client scripts and the WS-Resource using the standard container, try doing so with the SContainer. Using the SContainer means the signed messages are encrypted by the underlying transport protocol when sent between the client and server. There are many possible ways to use WS-Security and TLS together. For example, you can turn off mutual authentication in the SContainer and rely on the signed messages for authenticating the client to the server.

Section 6. Summary

In this tutorial, you learned about TLS and WS-Security. You should now understand the types of security threats a Web service faces and how to use TLS and WS-Security to combat them.

You also learned the process of signing a SOAP message and the reasons why different elements of a SOAP message need to be signed to create a secure message. You now know how to use TLS and WS-Security with WSRF::Lite to build secure WS-Resources.

Resources

Learn

- Get the WSRF specifications from OASIS TC for the standardization of WSRF.
- The W3C provides information about XML canonicalization, signing, and encrypting, including the relevant specifications.
- Read another developerWorks article about PKI: "Understanding the Public Key Infrastructure."
- Sam Thompson discusses "Implementing WS-Security."
- For background on WS-Addressing, see Doug Davis' article "The hidden impact of WS-Addressing on SOAP."
- To learn about TLS and SSL, check out "SSL: it's not just for commerce anymore."
- Get the WS-Security specifications from the OASIS TC for the standardization of WS-Security.
- The Web Service Interoperability organization's Security Challenges, Threats and Countermeasures provides an excellent discussion of threats and countermeasures for Web services.
- Read "Build WS-Resources with WSRF::Lite" to learn more about WSRF::Lite and the sample Counter WS-Resources discussed in this tutorial.
- The developerWorks Grid computing zone provides tutorials and information about grid technology.

Get products and technologies

- Download the SimpleCA tools to help create and manage X.509 certificates.
- Download the latest version of WSRF::Lite. The site also provides more information about WSRF::Lite.

Discuss

- Join the WSRF::Lite mailing list to discuss WSRF::Lite with other developers.

About the authors

Mark McKeown
Since 2002, Mark McKeown has worked on grid computing at the University of Manchester. During this time, he developed WSRF::Lite and OGSI::Lite, the precursor to WSRF::Lite. His academic background is in physics, in which he has a bachelor's degree and a doctorate from Queens University Belfast.

Stephan Zasada
Stefan Zasada worked on implementing WS-Security in Perl for use by WSRF::Lite as part of his master's project at the University of Manchester. He is currently a doctorate student in the Centre for Computational Science and Computer Science department at University College London, working on lightweight grid middleware.