Appears in C. Pautasso, T. Gschwind (Eds.):
2nd ECOWS Workshop on Emerging Web Services Technology – WEWST 2007,
CEUR Workshop Proceedings (CEUR-WS.org) Vol-313, ISSN 1613-0073, 2008.

SSL-over-SOAP: Towards a Token-based Key Establishment Framework for Web Services

Sebastian Gajek, Lijun Liao, Bodo Möller, and Jörg Schwenk

Horst Görtz Institute for IT Security, Ruhr-Universität Bochum
{sebastian.gajek | lijun.liao | joerg.schwenk}@nds.rub.de,
bmoeller@crypto.rub.de

Abstract. Key establishment is essential for many applications of cryptography. Its purpose is to negotiate keys for other cryptographic schemes, usually for encryption and authentication. In a web services context, WS-SecureConversation has been specified to make use of negotiated keys. The most popular key establishment scheme in the Internet is the (handshake protocol of the) Secure Socket Layer or Transport Layer Security protocol (SSL/TLS). However, SSL/TLS has primarily been designed to secure HTTP, by encrypting and authenticating TCP connections. It is thus not usable to negotiate keys in SOAP connections with intermediaries. We propose SSL-over-SOAP, a family of key establishment protocols for Web services. It is based the design of the SSL handshake, so security analysis results for standard SSL/TLS apply to our new proposal. We have implemented this protocol in the framework of WS-Trust and WS-SecureConversation.

Keywords: Web Services Security, Key Establishment Protocol

1 Introduction

1.1 Motivation

Security is important for any distributed computing environment: Many passive and active attacks have been described against such systems. Particularly challenging are service-oriented environments where the architecture is implemented based on a range of technologies, and where applications are created as loosely coupled and interoperable services. The Internet and its underlying infrastructure is the most pervasive IT system ever built—accordingly, more and more applications are implemented as Web services. Thus, preserving the privacy and integrity of these messages in service-oriented architectures becomes a challenging part of business integration, and secure message exchange a requirement for the proliferation of Web services.

Because of the complexity of XML based security standards, the well-known *Secure Socket Layer (SSL)* or *Transport Layer Security (TLS)*[1] protocol has

[1] TLS is the official name for the more recent protocol versions in the SSL/TLS family. We use the traditional name SSL as an umbrella term since our ultimate goal is to move the protocol ideas away from the transport layer.

become a de facto security standard for Web services. SSL provides a protected TCP channel that can be used by higher order protocols, such as e.g. SOAP over HTTP for Web Services. Because of its eminent role for the Web (as well as for other Internet protocols), the SSL protocol has been examined intensely [13–15], without finding any severe security flaws.

However, classical SSL has some shortcomings when deployed in the context of Web Services. First, SSL is a point-to-point security protocol. Web Services, by contrast, are loosely coupled applications: that is, messages may pass through multiple intermediary nodes, and the bindings to service endpoints may change. In order to establish a secure SSL channel between two service endpoints, each intermediary connection must be protected by SSL, and the application must be able to decide which of the intermediary SSL certificates are trustworthy. Second, SSL is a transport layer security protocol: SSL-protected messages are secured while in transit on the network; after reception, the message plaintext (as recovered by the SSL layer) is forwarded to the application logic. Third, SSL is not aware of the message structure, so messages are protected in an all-or-nothing fashion. Higher layers do not directly benefit from SSL session keys. One benefit of XML security technologies, in contrast, is to provide element-wise signing and encryption: intermediaries can read and alter information only as they are permitted to.

1.2 Motivating Example

Consider an example where a business flow requires passing an invoice through multiple parties. The invoice contains some vital information that only the ultimate receiver is allowed read; however, certain parts of the invoice are to be processed by intermediary parties. SSL fails because intermediaries have access to the complete invoice in plaintext (or they would not be able to examine the invoice at all). By contrast, SSL-over-SOAP allows for establishing a session key between sender and ultimate receiver. This key can be used to authenticate and encrypt the invoice information to protect it from intermediaries. The sender can choose which pieces of information to encrypt.

1.3 Contribution

The WS-* family of security schemes [1] aims to provide a security framework that addresses all the security issues around web services. In particular, WS-SecureConversation [2] defines how to use session keys in WS-Security, but does not specify any specific key exchange protocols.

In this work, we close this gap by re-specifying the SSL handshake protocol and the SSL record layer at the SOAP level, creating a new cryptographic protocol to be used with WS-SecureConversation. We thus do not need the “classical” SSL at TCP level any more. Instead, we are able to provide all security services offered by SSL (confidentiality, authentication, security of key establishment) at SOAP level.

Incorporating the practically proven SSL protocol technology into the WS-* family of security scheme allows us to design a protocol framework that benefits from both technologies.

The main contribution of the SSL protocol to the web services world is *security*. In SSL, key agreement and authentication are closely connected, and explicit key confirmation is provided by the `Finished` messages at the end of the handshake protocol. By contrast, it is easy to show the the authenticated variant of the Diffie-Hellman key exchange [7] is vulnerable to man-in-the-middle attacks in combination with XML wrapping attacks [12].

We propose SSL-over-SOAP as the first member of a family of practical Web Services key establishment protocols. SSL-over-SOAP provides sufficient protocol flexibility for the security requirements of today's business models. As a first step, we have implemented the following:

- The SSL-over-SOAP handshake protocol is a key transport protocol based on X.509 binary security tokens. It is implemented in the WS-Trust framework.
- The SSL-over-SOAP record layer protects the complete body of SOAP messages, and the `Finished` messages. It provides confidentiality (XML encryption) and authentication (HMAC from XML signature) within the framework of WS-SecureConversation.

1.4 Related Work

Hada and Maruyam [9] propose a session authentication protocol for Web Services. Although they consider the aspect of session resumption, they do not design a key establishment scheme. Follow-up work by Zhang and Xu [18] similarly does not regard key establishment. Herzberg [10] introduces the secure XML transport protocol (SeXTP), which is a ping pong protocol based on XML Encryption and XML Signature. The work does not fit into the Web Services terminology, as it dates back from a time when the WS-standards were in pending state. Although the author illustrates that present XML security standards are capable to negotiate a shared secret, [10] only mentions the Diffie-Hellman key exchange. In contrast to this, our protocol provides a framework for a wide variety of different algorithms and authentication mechanisms, and is open to extensions. Fang et al. [8] have implemented the AuthA protocol for Web services. AuthA is a password-based authenticated key exchange protocol. This protocol is restricted to the use of passwords. By contrast, SSL-over-SOAP provides protocol flexibility. That is, SSL-over-SOAP captures the requirements of different security models and allows the use of modular authentication mechanisms, such as passwords, digital certificates, or Kerberos ticket.

1.5 Organization

The paper is structured as follows. We shortly review the relevant Web Services technologies in Section 2. Then, we present our proposal by first formulating SSL in terms of SOAP message exchanges in Section 3, and subsequently describing

a concrete instantiation of this framework in Section 4. We discuss the protocol's security in Section 5. Finally, we conclude our work in Section 6.

2 WS-* Building Blocks

2.1 Notation

We use the following XML syntax style:

- Instead of writing an element <AAA></AAA>, we drop the tag from the closing bracket and write <AAA></> or <AAA/>.
- When writing an element that spans several lines, we rely on indentation to delimit the body, omitting the closing bracket. For example, <AAA><BBB></BBB></AAA> is written as

```
<AAA>
  <BBB />
```

- We omit the namespace definition in the messages and use the following prefixes:

Prefix	Namespace
ds	http://www.w3.org/2000/09/xmldsig#
soap	http://schemas.xmlsoap.org/soap/envelope/
tls	http://www.example.org/tls#
wsse	http://docs.oasis-open.org/wss/2004/01/ oasis-200401-wss-wssecurity-secext-1.0.xsd
wst	http://schemas.xmlsoap.org/ws/2005/02/trust
wsc	http://schemas.xmlsoap.org/ws/2005/02/sc
wsu	http://docs.oasis-open.org/wss/2004/01/ oasis-200401-wss-wssecurity-utility-1.0.xsd
xenc	http://www.w3.org/2001/04/xmlenc#

2.2 SOAP and WS-Security

SOAP is a mechanism for inter-application communication between systems across the Internet, where system implementations can be written in arbitrary languages. SOAP messages are in XML to allow the exchange of structured information.

WS-Security [3] describes how to use XML Signature [17], XML Encryption [16], and security tokens in SOAP messages. (Note that [17] specifies the use of symmetric-key authentication, not just public-key digital signatures: The term "signature" or "digital signature" is extended to cover symmetric authentication.) To this purpose, WS-Security defines a <Security> element to be added to the SOAP header as a container for all security related content. For WS-Security, it is strongly recommended to identify signed elements via ID attributes (*not* via XPath expressions). A typical WS-Security message is as follows:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security/>
  <soap:Body>
```

2.3 WS-SecureConversation

WS-SecureConversation (the Web Services Secure Conversation Language) specifies secure communication between services. It defines the message structure for establishing and sharing security contexts, and for deriving keys from security contexts. As with WS-Security, WS-SecureConversation is only a building block and does not provide a complete security solution.

The core element of WS-SecureConversation is the <wsc:SecurityContextToken> element. It consists of the mandatory child element <wsc:Identifier> and several optional elements. The security context is addressed by a UUID specified in <wsc:Identifier>.

2.4 WS-Trust

WS-Trust enables applications to construct trusted SOAP message exchanges, which is determined by security tokens. A typical security token request consists of:

```
<wst:RequestSecurityToken Context? ...>
  <wst:TokenType/>?
  <wst:RequestType/>
  ...
```

The optional element <wst:TokenType> describes the requested token type. The mandatory element <wst:RequestType> specifies the class of function. It allows to add additional elements for the special purpose. The <wst:RequestSecurityTokenResponse> specifies the response to a security token request and is used to retrieve a security token.

3 SSL-over-SOAP Framework

3.1 Design Goals

We lift the SSL framework from the transport layer to the world of Web Services, using the WS-* framework. We use SOAP instead of TCP for handshake messages transfer. WS-Trust and WS-SecureConversation provide the framework to describe handshake protocol messages, and WS-Security allows us to put security related metadata into the <Security> header element. Our specification of SSL-over-SOAP complies with the recent version of SSL, namely TLS 1.1 [6].

For the handshake protocol, we put the handshake messages into payloads of the SOAP exchange, such that all elements are contained in the body part of the SOAP message—never in the header. This is important to allow us to duplicate the renegotiation feature of the original SSL protocol: An SSL handshake can also be carried out over a connection that is already protected using SSL/TLS (for example, to transfer client certificates in encrypted form). In the case of SSL-over-SOAP, this means that we want to be able to apply WS-Security and WS-SecureConversation even to those SOAP messages constituting a new handshake. In such a situation, the cryptographic parameters used in the WS-Security header stem from the original session, while the parameters of the next session are negotiated using the handshake protocol.

Additionally, we define how the `Finished` message is generated. In the SSL/TLS standards, this is done by computing a pseudorandom function taking as input the exchanged secret and a concatenation of all message bytes exchanged for the current handshake, and then sending this message over the new cryptographically secured channel. Such authentication is important to thwart man-in-the-middle attacks: If an attacker has modified some handshake message to influence parameter negotiation, then verifying the peer's `Finished` messages will reveal that something is wrong (assuming that only ciphersuites providing reasonably strong authentication are ever negotiated).

In SSL-over-SOAP, we use the same pseudorandom function for the `Finished` message. Its inputs are the new master secret and the concatenation of the (serialized canonicalized) bodies of the SOAP messages for the handshake. Putting parts of the handshake messages into the SOAP header instead of the body might make the protocol vulnerable to attacks, or could lead to a much more complex computation of the `Finished` message (see Section 5 for more discussions).

3.2 ClientHello (Message 1)

For convenience, we will refer to the initiating service as "client" and to the responding service as "server", thus adopting SSL terminology. In SSL-over-SOAP, the initial message is the `ClientHello` (Fig. 1). Using the framework of WS-Trust, we embed the messages into the <wst:KeyExchangeToken> within the <wst:RequestSecurityToken> element. The <wst:RequestKET> indicates that an additional message from the server is required to complete the key establishment. The <wst:RequestSecurityToken> has an attribute @tls:Id with a UUID value so that we can reference to this message later.

The <tls:ClientHello> specifies the SSL version of the client (<tls:Version>), the ciphersuites (<tls:CipherSuites>) and compression methods (<tls:CompressionMethods>) as well as the client's nonce (<wst:Entropy>). The version number 3.2 indicates that TLS version 1.1 is used [6, Section 6.2.1]. While the ciphersuite in SSL is identified by two bytes, it is identified here by a URI. For example, the ciphersuite TLS_RSA_WITH_AES_256_CBC_SHA-1 is identified by http://www.example.org/tls#tls_RSA_with_AES256_SHA-1.

```
<soap:Envelope>
 <soap:Body>
  <wst:RequestSecurityToken tls:Id='uuid:UUID-msg1'>
   <wst:TokenType>.../sc/sct</>
   <wst:RequestType>.../trust/KET</>
   <wst:RequestKET/>
   <wst:KeyExchangeToken>
    <tls:ClientHello>
     <tls:Version>3.2</>
     <tls:CipherSuites>
      <tls:CipherSuite>...#tls_RSA_with_AES256_SHA1</>
     <tls:CompressionMethods>
      <tls:CompressionMethod>...#compression_null</>
     <wst:Entropy>
      <wst:BinarySecret Type='.../Nonce'>M7o9...MO0o=</>
```

Fig. 1. `ClientHello` message sent from Client to Server in order to initiate the handshake

3.3 ServerHello, Certificate, ServerKeyExchange, CertificateRequest (Message 2)

The second message is the server's response to the security token request (Fig. 2). The response contains mandatory and optional elements. As with SSL, the choice of elements depends on the ciphersuite selected (<tls:ServerKeyExchange>) and on whether the server requests client authentication (<tls:CertRequest>).

<tls:ServerHello> contains the SSL protocol version of the server (<tls:Version>), the session ID (<tls:SessionID>), the ciphersuite selected by the server (<tls:CipherSuite>), the compression method (<tls:CompressionMethod>), and the server nonce (<wst:Entropy>). The session ID is adopted from SSL to manage the session and used to execute the abbreviated handshake.

Fig. 2 illustrates an example message where client and server are mutually authenticated on the basis of X.509v3 Certificates. Such certificates are handled by including a <wsse:BinarySecurityToken> into the <tls: Certificates > element. The type X509PKIPathv1 indicates that this token specifies a certificate chain.

<tls:ServerKeyExchange> would contain key material for DH key exchange. In Fig. 2, client and server opt for key transport based on RSA where the server's public key is provided by the certificate. Hence, <tls:ServerKeyExchange> is an empty tag, and could be omitted.

The <tls:CertificateRequest> element is used to signal the client that is has to authenticate using a X.509 security token. The tag contains security policies which specify the client certificate's requirements. In Fig. 2, the server requires client authentication and opts for a client certificate that is issued for RSA signatures from the Certificate Authority Test Root CA.

The SSL/TLS message `ServerHelloDone` serves only as a delimiter. We omit `ServerHelloDone` in SSL-over-SOAP, since we combine multiple SSL elements into one <wst:KeyExchangeToken>.

```
1 <soap:Envelope>
2  <soap:Body>
3   <wst:RequestSecurityTokenResponse tls:Id='uuid:UUID-msg2'>
4    <wst:TokenType>.../trust/KET</>
5    <wst:RequestedSecurityToken>
6     <wst:RequestKET/>
7     <wst:KeyExchangeToken>
8      <tls:ServerHello>
9       <tls:Version>3.2</>
10      <tls:SessionID>Vz2e...4WU=</>
11      <tls:CipherSuite>...#tls_RSA_with_AES256_SHA1</>
12      <tls:CompressionMethod>...#compression_null</>
13      <wst:Entropy>
14       <wst:BinarySecret Type='.../Nonce'>ihsK...7CYA=</>
15     <tls:ServerKeyExchange>
16     <tls:Certificates>
17      <wsse:BinarySecurityToken
18         ValueType='...#X509PKIPathv1'>MIIC...iw==</>
19     <tls:CertRequest>
20      <tls:Certtype>...#rsa_sign</>
21      <tls:CAs>
22       <tls:CA>CN=Test Root CA</>
```

Fig. 2. `ClientHello, ServerKeyExchange, Certificate, and CertificateRequest` message sent from Server to Client. Server and Client agree on the ciphersuite "#tls_RSA_with_AES256_SHA1".

3.4 ClientKeyExchange, Certificate, CertificateVerify, Finished (Message 3)

Here the message structure becomes a little more complicated, since we have to combine unprotected parts (`ClientKeyExchange`, `Certificate`, `CertificateVerify`) and protected parts (`Finished`) into one SOAP message in order to comply with the SSL protocol specification (Fig. 3).

The <tls:PreMasterSecret> (lines 34–37) within <tls:ClientKeyExchange> (lines 33–37) contains the encrypted premaster secret (recall that client and server negotiated the RSA ciphersuite). The premaster secret is encrypted with the server's public key. Since client authentication has been requested in the previous message, the client makes use of the <tls: Certificates > element (lines 38–40) to send a certificate chain containing its certificate and the certification authority's certificates. The <tls: CertificateVerify > (lines 41–50) has only

one child element, <ds:Signature> (lines 42–50). For more details, see Section 4. The <ds:Reference> elements in lines 46–47 reference the exchanged messages (messages 1 and 2) via the @URI with the prefix *urn:uuid*. The last two <ds:Reference> elements in lines 48–48 reference the <tls:ClientKeyExchange> (lines 33–37) and the <tls: Certificates > (lines 38–40) in the same message. The <wst:SecurityContextToken> (lines 28–29) specifies that the master secret should be addressed by the UUID specified, *UUID-sct*.

After choosing a premaster secret, the client computes the master secret and derives the session keys. In order to confirm the correct generation of session keys, it computes the content `Finished` message as follows:

$$\begin{aligned} \textit{client finished} = \texttt{PRF}_{\textit{master secret}}\big(&\text{“}\texttt{client finished}\text{”}, \\ &\textit{MD5}(\textit{exchanged messages}) \,||\, \textit{SHA1}(\textit{exchanged messages})\big)[0...11] \end{aligned}$$

In the SSL framework, the exchanged messages are those visible at the handshake layer and do not include record layer headers. Hence in SSL-over-SOAP, the exchanged messages are the SOAP bodies in messages 1 and 2, and the SOAP body except the <tls:Finished> in this message. The SOAP bodies are first canonicalized with the algorithm Exclusive C14N and then concatenated. The result is then used as the input of the hash algorithms MD5 and SHA-1.

The client then constructs the `Finished` message by encoding the result of the TLS PRF: <tls:Finished>Base64(client finished)</>. The <tls:Finished> is then encrypted and signed in the following way:

The client inserts a <wsse:Security> (lines 3–23) into the SOAP header <soap:Header>. Then, the client adds a <wst:SecurityContextToken> (lines 4–5) to <wsse:Security>. This token has the same properties as the token within the SOAP body (lines 28–29). However, we may not move this security token to the body, since only if it is located in the SOAP header, the token can be used for decryption and signature verification. Then, the client computes a derived key token <wsc:DerivedKeyToken> that specifies the client_write_key, and is used to encrypt the content of <tls:Finished> (lines 51–56).The <wsc:DerivedKeyToken> (lines 17–21) and a new <xenc:ReferenceList> (lines 22–23) that locates the encrypted <tls:Finished> are then added to the <wsse:Security> element. Finally, the client computes a derived key token <wsc:DerivedKeyToken> (lines 6–10) that specifies the client_write_MAC_secret used to sign the encrypted <tls:Finished> with the client_MAC_secret. The signature is represented by a <ds:Signature> (lines 11–16). Both elements are then added to the <wsse:Security>.

Note that in message 3 and in message 4 (see Sect. 3.5), the session ID is added to the <wst:KeyExchangeToken> so that the receivers (the server in message 3 and the client in message 4) can chain the messages.

3.5 Finished (Message 4)

The server's <wst:RequestSecurityTokenResponse> contains the <tls:Finished> message (see Fig. 4). To compute the content of the `Finished`

```
<soap:Envelope>
 <soap:Header>
  <wsse:Security>
   <wsc:SecurityContextToken wsu:Id='Id-sct'>
    <wsc:Identifier>uuid:UUID-sct</>
   <wsc:DerivedKeyToken wsc:Algorithm='...#TLS_PRF'
       wsu:Id='Id-clientMACKey'>
    <wsse:SecurityTokenReference>
      <wsse:Reference URI='#Id-sct'/>
    <wsc:Length>20</><wsc:Offset>0</>
   <ds:Signature>
    <ds:SignedInfo>
     <ds:Reference URI='#Id-finished'>
      <ds:KeyInfo>
       <wsse:SecurityTokenReference>
        <wsse:Reference URI='#Id-clientMACKey'/>
   <wsc:DerivedKeyToken wsc:Algorithm='...#TLS_PRF'
       wsu:Id='Id-clientWrtKey'>
    <wsse:SecurityTokenReference>
     <wsse:Reference URI='#Id-sct'/>
    <wsc:Length>32</><wsc:Offset>40</>
   <xenc:ReferenceList>
    <xenc:DataReference URI='#Id-EncFinished'/>
 <soap:Body>
  <wst:RequestSecurityTokenResponse>
   <wst:TokenType>.../trust/KET</>
   <wst:RequestedSecurityToken>
    <wsc:SecurityContextToken>
     <wsc:Identifier>uuid:UUID-sct</>
    <wst:KeyExchangeToken>
     <tls:SessionID>Vz2e...4WU=</>
     <wst:RequestKET/>
     <tls:ClientKeyExchange Id='Id-cke'>
      <tls:PreMasterSecret>
       <xenc:EncryptedKey>
        <xenc:EncryptionMethod Algorithm='...#rsa-1_5'/>
        <ds:KeyInfo/><xenc:CipherData />
     <tls:Certificates Id='Id-certs'>
      <wsse:BinarySecurityToken ValueType=
        '...#X509PKIPathv1'>MIIC...y/Z/</>
     <tls:CertificateVerify>
      <ds:Signature>
       <ds:SignedInfo>
        <ds:CanonicalizationMethod />
        <ds:SignatureMethod Algorithm='...#rsa-sha1'/>
        <ds:Reference URI='urn:uuid:UUID-msg1'/>
        <ds:Reference URI='urn:uuid:UUID-msg2'/>
        <ds:Reference URI='#Id-cke'/>
        <ds:Reference URI='#Id-certs'/>
       <ds:SignatureValue>RR4p...vFvA=</>
     <tls:Finished Id='Id-finished'>
      <xenc:EncryptedData Id='Id-EncFinished'>
       <ds:KeyInfo>
        <wsse:SecurityTokenReference>
         <wsse:Reference URI='#Id-clientWrtKey'/>
       <xenc:CipherData/>
```

Fig. 3. `ClientKeyExchange`, `Certificate`, `CertificateVerify`, and `Finished` sent from Client to Server.

message, the server uses the same `PRF` function as the client except the following differences: 1) the label is "server finished"; 2) for the exchanged message, we refer to the SOAP bodies in messages 1, 2 and 3. In the SSL framework, the headers in the record layer are not considered, hence we first decrypt the message 3 and use the decrypted SOAP body (namely the <tls:Finished> is not encrypted).

The message allows the client to verify that the server has received all the previous messages from the client. As with the previous message, the server's `Finished` has the same SOAP header structure, but differs in the keys used, i.e. the server uses server_write_MAC_secret for the HMAC and server_write_key for the encryption.

```
1 <soap:Envelope>
2  <soap:Header>(Similar as in Message 3)</>
3  <soap:Body>
4   <wst:RequestSecurityTokenResponse>
5    <wst:TokenType>.../trust/KET</>
6    <wst:RequestedSecurityToken>
7     <wst:KeyExchangeToken>
8      <tls:SessionID>Vz2e...4WU=</>
9       <tls:Finished wsu:Id='Id-Finished2'>
10       <xenc:EncryptedData Id='Id-EncFinished2'/>
```

Fig. 4. `Finished` message sent from Server to Client that confirms the negotiated ciphersuite and derived session keys.

4 A Token-based Protocol

We present additional details for a concrete Web Services protocol that instantiates the framework described in the previous section. Specifically, we show a protocol variant using X.509 certificates as security tokens. The framework can similarly be instantiated using other security token types, requiring other protocol variants: for example, password-based authenticated key exchange using a scheme such as "SOKE" [4] would use user name tokens. Recall that during the execution of the protocol, the services endpoints may decide which authentication token to use.

The X.509v3 binary token authentication protocol is illustrated in Fig. 5. The protocol's goal is to negotiate a tuple of session keys between two services $\mathcal{W}_1$ and $\mathcal{W}_1$, while the services authenticate on the basis of X.509v3 binary tokens. Assuming that in a setup stage the tokens have been stored in credential stores. We denote the certified public pairs of $\mathcal{W}_1$ and $\mathcal{W}_2$ by (pk_1, sk_1) and (pk_2, sk_2), respectively.

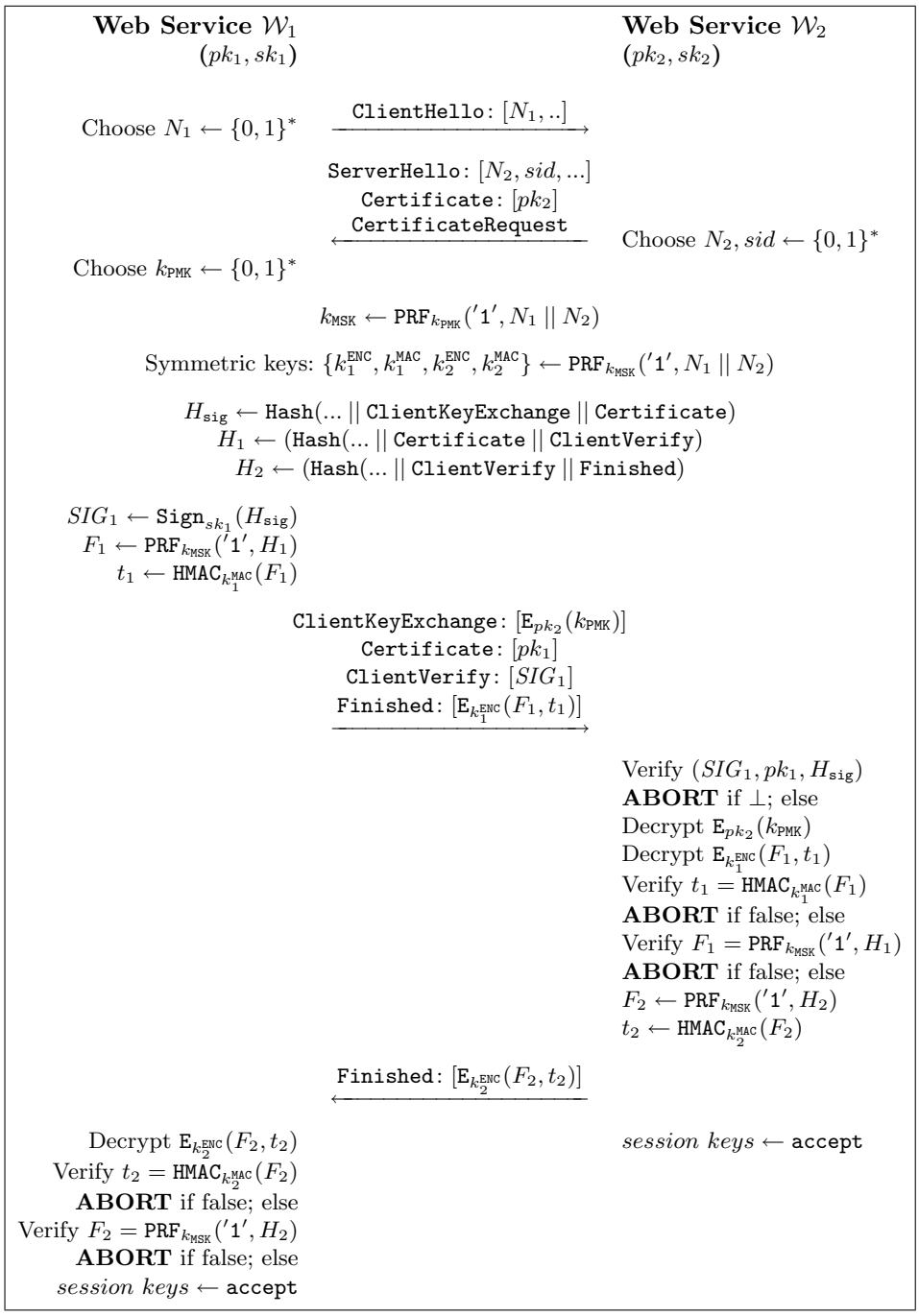

Fig. 5. Key establishment protocol based on X.509v3 tokens

$\mathcal{W}_1$ initiates the handshake. It randomly chooses a client nonce N_1 and forwards this parameter to $\mathcal{W}_2$. Then, $\mathcal{W}_1$ chooses a nonce N_2 and appends to the nonce its certified public key pk_2. Upon receiving the message, $\mathcal{W}_1$ randomly chooses a premaster secret $k_{\mathtt{PMK}}$ and encrypts the premaster secret with the public key pk_2 from $\mathcal{W}_2$. The premaster secret $k_{\mathtt{PMK}}$ is used to derive the master secret $k_{\mathtt{MSK}}$, using the pseudo-random function $\mathtt{PRF}$ parameterized by the services' nonces and the labeling string "`master secret`", abbreviated with "1" in the protocol description. This is the pseudorandom function as specified by SSL. (Other PRF constructions could be used.)

$\mathcal{W}_1$ applies the master secret $k_{\mathtt{MSK}}$ to compute the tuple of session keys $\{k_1^{\mathtt{ENC}}, k_1^{\mathtt{MAC}}, k_2^{\mathtt{ENC}}, k_2^{\mathtt{MAC}}\}$. Here $k_i^{\mathtt{ENC}}$ and $k_i^{\mathtt{MAC}}$ are encryption and authentication keys, respectively. $\mathcal{W}_1$ feeds the pseudorandom function $\mathtt{PRF}$ with the labeling string "key expansion" and the concatenation of the services' nonces N_1 and N_2. In addition, $\mathcal{W}_1$ proves its identity by signing the digest of previously negotiated messages SIG_1 using its certified private key sk_1. Finally, $\mathcal{W}_1$ confirms the session key generation, using the pseudorandom function $\mathtt{PRF}$ that takes as input the master secret $k_{\mathtt{MSK}}$, the string "`client finish`" and the hash value H_1 of all previous messages. It then authenticates and encrypts the output $\mathtt{F_1}$ with the session keys $\{k_1^{\mathtt{ENC}}, k_1^{\mathtt{MAC}}\}$.

Upon receiving the message, $\mathcal{W}_1$ decrypts the premaster secret and verifies $\mathtt{SIG_1}$, using the client's certified public key pk_1. It ensures that it is connected to $\mathcal{W}_1$, i.e. it checks that $\mathcal{W}_1$ is a valid endpoint. Otherwise $\mathcal{W}_2$ aborts the session. In the positive case, $\mathcal{W}_2$ derives in analogy to $\mathcal{W}_1$ the same master secret $k_{\mathtt{MSK}}$ and the same tuple of session keys $\{k_1^{\mathtt{ENC}}, k_1^{\mathtt{MAC}}, k_2^{\mathtt{ENC}}, k_2^{\mathtt{MAC}}\}$. Then, $\mathcal{W}_2$ decrypts $\mathtt{E}_{k_1^{\mathtt{ENC}}}(F_1, t_1)$ and verifies that tag $t_1 := \mathtt{HMAC}_{k_1^{\mathtt{MAC}}}(F_1)$, where F_1 is the hash over all previous messages. If this verification fails, $\mathcal{W}_2$ aborts the protocol. Otherwise, $\mathcal{W}_2$ confirms the negotiated session keys using the pseudorandom function $\mathtt{PRF}$ that takes as input the master key $k_{\mathtt{MSK}}$, the labeling string "server finish", and the hash value over all previous messages H_2. It then authenticates and encrypts the output F_2 with the session keys $\{k_2^{\mathtt{ENC}}, k_2^{\mathtt{MAC}}\}$.

Finally, when $\mathcal{W}_1$ receives the message $\mathtt{E}_{k_2^{\mathtt{ENC}}}(F_2, t_2)$, it decrypts the message and verifies that tag $t_2 := \mathtt{HMAC}_{k_2^{\mathtt{MAC}}}(F_2)$, where F_2 is the hash value of all previously received messages. If the verification fails, $\mathcal{W}_1$ aborts the session. Otherwise, $\mathcal{W}_1$ and $\mathcal{W}_2$ start to use the negotiated keys $\{k_1^{\mathtt{ENC}}, k_1^{\mathtt{MAC}}, k_2^{\mathtt{ENC}}, k_2^{\mathtt{MAC}}\}$ for symmetric cryptography.

5 Security Discussion

Although our SSL-over-SOAP protocol on the outside looks very different from standard transport-layer SSL/TLS, the handshake quite closely follows the original protocol. We have replaced the SSL/TLS data formats using an XML-based approach, but without changing the cryptographic essence. Thus, previous analyses of the SSL/TLS handshake as appearing in [13–15] apply similarly: the long experience with SSL/TLS provides evidence that our proposal is cryptographically sound as well.

To get a feel for the cryptographic approach in these protocols (both the original SSL/TLS and our SSL-over-SOAP), observe that most of the handshake negotiation is not cryptographically authenticated immediately. Besides signatures in certificates, authentication appears only in the form of digital signatures if either a `ServerKeyExchange` message is used (the server signs its key share along with the client and server random nonces, thus binding the key share to the current handshake), or if a `CertificateVerify` message is used (in which the client presents a signature on the handshake so far to authenticate itself to the server).

Many typical scenarios involve neither message. An attacker can manipulate the handshake protocol messages being exchanged to influence the handshake outcome: For example, if the client offers multiple ciphersuites in the `ClientHello` message, an attacker could remove the client's preferred ciphersuites from the list, leaving the server with fewer ciphersuites to choose from—such as just those ciphersuites that are the easiest to break. This changes only in the moment when the `Finished` messages are exchanged. These messages cover the complete handshake as well as the resulting master secret, thus retroactively authenticating everything in the current handshake, provided that the master secret could only be known to the legitimate protocol participants. (For example, in an RSA-based handshake, the client encrypts the premaster secret for the server's certified public key, thus ensuring that the premaster secret and thus the master secret remains secret from any attacker.) Accordingly, it is a fundamental security requirement the any party engaging in a handshake only be willing to negotiate ciphersuites that can be assumed to provide security in this sense. Any further security properties, notably those of application data encryption, rely on this.

The `Finished` message is the first piece of data to be encrypted and authenticated under the newly negotiated keys and algorithms, thus also providing a verification that negotiation succeeded as intended and that both parties now are indeed using compatible cryptography. Once the `Finished` messages have been verified, application data is encrypted and authenticated the same way. In the standard SSL/TLS protocol, symmetric authentication is added to the plaintext before encryption.

This is done differently in our SSL-over-SOAP setting (see Fig. 3), where symmetric authentication (following the XML Digital Signatures specification) is applied to the ciphertext. This change is not cryptographically trivial, but does not harm the protocol. The combination of symmetric authentication with encryption can be considered *authenticated encryption* [5]. As discussed in [5], for general composition of an encryption scheme with a MAC, the "encrypt-then-MAC" approach does the best job of providing authenticated encryption. ("MAC-then-encrypt" as used in standard SSL/TLS in general has some problems, although these do not apply to the standard ciphersuites [11].) That is, while SSL-over-SOAP differs from standard SSL/TLS in its use of symmetric cryptography, the approach used in SSL-over-SOAP is in fact cryptographically sound.

6 Conclusion and Outlook

The SSL-over-SOAP approach provides a practical framework for key establishment for Web Services. We use the experience with the practically proven SSL/TLS protocol family for this purpose. This allows us to transfer SSL/TLS protocol ideas to reuse them for Web Services, while giving us much more flexibility and security than direct use of SSL/TLS at the transport layer. Our prototype implementation has shown the feasibility of implementing complex cryptographic protocols within the WS-* framework.

In this paper, we only looked at one basic form of an SSL handshake as an example—an RSA-based handshake (involving an encrypted premaster secret). The SSL-over-SOAP approach applies to many more protocol variants. For example, we can directly transfer the work that has been done in [4] for password-based authenticated key exchange in TLS, where parties rely on low-entropy secrets instead of certificates for authentication. So besides X.509v3 binary token authentication as described in Section 4, we can also specify password token authentication using the "SOKE" scheme from [4]. We plan to complete an open source software library for SSL-over-SOAP, which will offer ciphersuites for both for X.509v3 binary token authentication and for password token authentication.

Our experiences with SSL-over-SOAP should be considered as a starting point for the definition of other key agreement protocols, e.g., the IPsec OAKLEY protocol, or group key agreement protocols. However, security analyses of such protocols can not be directly transferred to the web services world, e.g. considering XML wrapping attacks. Necessary conditions for key agreement protocols to be secure in an XML context (e.g. explicit key confirmation) have to be researched.

References

1. Security in a Web Services World: A Proposed Architecture and Roadmap, April 7, 2002. `http://www.ibm.com/developerworks/library/specification/ws-secmap/`.
2. Web Services Secure Conversation Language Specification (WS-SecureConversation), February 1, 2005. `ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf`.
3. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004) Working Draft, November 7, 2005. `http://www.oasis-open.org/committees/download.php/15251/oasis-wss-soap-message-security-1.1.pdf`.
4. M. Abdalla, E. Bresson, O. Chevassut, B. Möller, and D. Pointcheval. Provably secure password-based authentication in TLS. In S. Shieh and S. Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pages 35–45, 2006.
5. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976, pages 531–545, 2000.
6. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol, version 1.1. RFC 4346. `http://www.ietf.org/rfc/rfc4346.txt`, 2006.

7. W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.
8. L. Fang, S. Meder, O. Chevassut, and F. Siebenlist. Secure password-based authenticated key exchange for web services. In *SWS '04: Proceedings of the 2004 workshop on Secure web service*, pages 9–15, New York, NY, USA, 2004. ACM Press.
9. S. Hada and H. Maruyama. Session authentication protocol for web services. In *SAINT-W '02: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops*, page 158, Washington, DC, USA, 2002. IEEE Computer Society.
10. A. Herzberg. Secure XML transport protocol. Lecture Notes, Chapter 14, 2000. `http://www.cs.biu.ac.il/~herzbea/Chapters/Chapter%2014%20XML%20Security.pdf`.
11. H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139, pages 310–331, 2000.
12. M. McIntosh and P. Austel. Xml signature element wrapping attacks and countermeasures. In *ACM Workshop on Secure Web Services*, 2005.
13. J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *7th USENIX Security Symposium*, 1998.
14. L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Computer and System Security*, (3):332–351, 1999.
15. B. Schneier and D. Wagner. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, 1996.
16. W3C Consortium. XML-encryption syntax and processing, 2002. `http://www.w3.org/TR/xmlenc-core`.
17. W3C Consortium. XML-signature syntax and processing, 2002. `http://www.w3.org/TR/xmldsig-core`.
18. D. Zhang and J. Xu. Multi-party authentication for web services: Protocols, implementation and evaluation. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 227–234, Los Alamitos, CA, USA, 2004. IEEE Computer Society.