Security and Usability with Web Services and WS-Security

by

Brad Gronek gronek@fas.harvard.edu

and

Greg Robinson greg_robinson@fastmail.us

INTRODUCTION

One of the most pervasive issues in the field of software security technology is the well established notion that few, if any, secure software programs are easily usable and, as a result, are not effectively secure (Smetters & Grinter, 2002; Yee, 2002; Flechais, Sasse & Hailes, 2003, Whitten & Tygar, 1998). While many research efforts have approached the usability issue by presenting user interface critiques and improvements (Whitten & Tygar, 1998; Yee, 2002), an increasing amount of effort is being focused on critique and improvement of the underlying technologies and software development methodologies used to implement security in software (Smetters & Grinter, 2002; Shigetomi, Otsuka & Imai, 2003; Flechais et al., 2003). With the advent of the Web Services model and the more recent development of the WS-Security model for securing Web Services, it is becoming increasingly possible to use readily available Application Programming Interfaces (APIs) and software development patterns to create applications that support secure cross-platform communication.

As computers become more ubiquitous and mobile, users will increasingly demand seamless security in communications. Already, systems are being developed which will internet-enable automobiles and homes in such a way that users in a car will be able to control systems in the home. Obviously, such communications will need to be secure and seamless. In order to facilitate the development of such seamless systems, security mechanisms must be standardized such that cross-platform interconnected secure communications are realized. To date, there have been many attempts to standardize various subsections of software security mechanisms including the Java framework security classes, OpenSSL toolkit, and Kerberos authentication (Smetters & Grinter, 2002); however, until recently, no common framework has been created to support the full breadth of security services. With the advent of WS-Security, it is becoming possible to use a common API to create software that securely communicates among disparate operating systems and networks.

The goals of this paper are to educate the reader on software security, PKI concepts, Web Services and WS-Security; to demonstrate a simple, usable implementation of PKI with WS-Security in a peer-to-peer chat program; and to discuss remaining challenges to the security and usability problem.

BACKGROUND

Software Security Concepts

Many sources including Henry (1999) and Simson & Spafford (2002) list the following five services required by any complete security system:

- *Authentication* – Authentication involves establishing the identity of a participant. Tools used for authentication include authentication protocols such as Kerberos as well as cryptography tools to keep passwords secret.
- *Authorization* – Authorization is the limitation of resource only to desired participants.
- *Integrity* – Providing integrity is analogous to insuring that data is not altered between the sender and the receiver. Hashing algorithms such as SHA-1 are the primary tools used to provide integrity; however, a combination of digital signature and encryption can also provide this service.

- *Confidentiality* – Software provides confidentiality when data is readable only by authorized recipients. Tools used in software to insure confidentiality include various cryptographic algorithms which are discussed below.
- *Non-repudiation* – Software provides non-repudiation when a sender cannot deny having sent or authorized a message. This concept is similar to a person signing a contract without the need to use paper to archive the transaction. Tools used in non-repudiation include certain types of cryptographic algorithms used in a way that creates a *digital signature*. A more detailed description of this concept follows.

The uses of these services are extremely broad and context-sensitive. The development of authentication systems is a core part of almost any software development project and proprietary implementations are found in everything from operating systems to network environments, internet sites, financial systems and video games.

The other three services are less commonly found in application software as they are more complex in both concept and implementation. As programmers increasingly rely on frameworks such as Sun's Java and Microsoft's .NET, their time is increasingly focused on creating application domain logic and presentation rather than infrastructural components. Therefore, the other three services should become more common as simple framework implementations of these services are provided by vendors.

That said, the services of integrity, confidentiality and non-repudiation can all be implemented using cryptographic techniques described below.

Cryptography

Cryptographic techniques used to provide the above services can be broken down into symmetric and asymmetric algorithms. Each type of algorithm has its own strengths and weaknesses; however, used together, it is possible to implement security in high-performance systems that transfer data at rates exceeding 1 Gb/s (Daemen & Rijmen, 2003). Implementations of these algorithms are the core of publicly available secure software.

Symmetric algorithms use the same key to encrypt and decrypt data. They are able to encrypt and decrypt data very quickly; however, their reliance on a single key leaves open the issue of confidential exchange of keys prior to use. Strengths of symmetric algorithms include speed of encryption and decryption, variety of available algorithms and, in terms of infrastructure, a great deal of flexibility as to which algorithms are used. The greatest drawback of symmetric algorithms is the fact that, in order to initiate communication, the encryption key must be transferred securely prior to use. This is where, in providing confidentiality to communications, asymmetric algorithms come into play.

Asymmetric algorithms encrypt and decrypt data using two different keys. Data encrypted by one key is decrypted by the other key and vice versa. One key is designated as a *public* key while the other is designated as a *private* key. The public key is published or transferred to other conversants while the private key is kept secret. Strengths of asymmetric algorithms include their ability to provide multiple security services and the existence of a public key which can be used by anyone. Weaknesses of asymmetric algorithms include complexity of key exchange mechanisms and the fact that these algorithms take significant resources to perform their functions.

Using Cryptography To Provide Security

While symmetric cryptography only provides confidentiality, asymmetric cryptography can provide authentication, confidentiality and non-repudiation through varied use of public and private keys. The first step in all of these mechanisms is the exchange of public keys between the sender and receiver.

When providing *non-repudiation*, a message is encrypted by the sender using a private key. When received, the message is then decrypted using the sender's public key. Although the communication is not secret, the recipient knows that the data was encrypted by the owner of the private key who can therefore not deny having sent it.

When using asymmetric keys to provide *confidentiality*, the sender encrypts a message using the recipient's public key. The message can then only be decrypted by using the recipient's private key, which should only be in the hands of the recipient. Because asymmetric encryption algorithms are slow, real world systems will typically use asymmetric cryptography to confidentially transfer a key which is then used to employ symmetric encryption to carry out the secure communication.

When using asymmetric keys to provide *authentication*, the client digitally signs a login message using the procedure above. When the message is decrypted and the sender's certificate is verified by the server, the client is given access to the appropriate resources.

When using asymmetric keys to provide *integrity*, a combination of digital signature and encryption are employed. First, the message is digitally signed by the sender. Second, the message is encrypted using the recipient's public key. Upon receiving the message, the recipient first decrypts the message with his / her private key then decrypts the message with the sender's public key. Since the message was encrypted, it could not have been altered after being sent. Since the message was digitally signed, the sender can be verified.

Public Key Infrastructure and Trust

A public key infrastructure (PKI) is the collection of laws, policies, standards and software used to manage the creation, distribution, validation and invalidation of asymmetric keys (Henry, 1999; Josang & Sanderud, 2003). Public Key Infrastructures can be created for use within specific organizations as in MIT's PKI authentication system or created for public use as in the Certificate Authorities belonging to the AICPA WebTrust program including Verisign and Thawte. Two models have emerged for establishing trust among users.

The first such model involves a *web of trust*. In the web of trust model, participants trust each other based on their knowledge / trust of each other. If person A trusts person B and person B trusts person C, then person B can attest to person C's trustworthiness and, based on that, person A can trust person C. This model, which was used to implement Pretty Good Privacy (PGP), has been useful in establishing trusts in informal environments (Whitten and Tygar, 1998).

The second model is the PKI model where a central authority (CA) certifies public keys by first verifying the identity of a key holder, then digitally signing it. This certification is

presented in the form of an X.509 digital certificate which includes, among other information, the public key of the holder and information against which the holder can be identified (such as an email address) (Henry, 1999).

The process for registering involves the following steps:

1. The registrant generates its own public/private key pair or the CA generates the key pair and securely transfers the keys to the registrant (AICPA & CICA, 2000). The means for doing so are easily obtainable and can be accomplished in code using the OpenSSL API.
2. The registrant submits proof of identity to the CA and shows that he/she holds the private key for the given key pair (AICPA & CICA, 2000). This involves providing a photo ID, corporate tax information, or notarization signed using the registrant's private key.
3. The proof of identity is verified by the CA (AICPA & CICA, 2000). .
4. The CA issues a digitally signed certificate which is published for validation (AICPA & CICA, 2000).

This model provides a well defined infrastructure for the verification of signatures; however, it falls short by failing to present a well defined means of publishing and exchanging public keys. Currently, the most common method for publishing and managing public keys is to use an LDAP server; however, there is no ubiquitous repository for doing so in the PKI system (Henry, 1999). What is left is a reliable means of digitally signing and encrypting messages without a solution to the greatest barrier between implementation and usability – a simple way to exchange public keys to begin secure communication.

Web Services

Web Services is a software development architecture that enables XML-based cross-platform communication over the internet (Booth et al., 2004). The primary purpose of web services is interoperability of software over a network. A typical usage pattern for a web service involves publishing an API to a code library on the internet using an XML document describing the API in the Web Services Description Language (WSDL). Clients can consume this WSDL document to locate and call the methods described therein and, execute the code in the web service. Implementations in the .NET and Java frameworks make communication as easy as referencing the remote location from an application and calling the methods exposed by the service (Damiani, di Vimercati, & Samarati, 2002). This approach is commonly known as Remote Procedure Call (RPC).

Web Services are built around the Simple Object Access Protocol (SOAP). This protocol allows the exchange of XML documents over the Hypertext Transport Protocol (HTTP) as a way for applications running on different operating systems, different technologies and different programming languages to communicate with one another. SOAP defines a message in terms of an *envelope*, which contains a *header* and a *body* (Nadalin, Kaler, Hallam-Baker & Monzillo, 2004). The SOAP header addresses and describes the message and the body contains the message.

The potential for this technology to provide flexibility and simplicity to networked software development is enormous; however, the authors of the standard failed to include one important topic in the specification: security (Damiani, di Vimercati, & Samarati, 2002). Under the typical usage scenario of publishing a web service on the internet, services rely on Secure Socket Layer (SSL) to provide confidentiality to communications at the transport level; however,

as the Web Services architecture matures into a protocol-independent distributed software development platform, a much greater degree of flexibility is necessary.

In response to this and other needs, a consortium of companies called the OASIS Open group has developed various extensions to Web Services, the most important of which is the WS-Security specification (Nadalin et al., 2004). The goal of WS-Security is to provide a flexible means for establishing trust domains, signature formats, message encryption and security tokens (Nadalin et al., 2004). It is left to individual implementations to use these tools to provide confidentiality, authentication, integrity and non-repudiation.

WS-Security

WS-Security is one specification of the Web Services Enhancement (WSE), an add-on to the Microsoft .NET framework enabling developers to include the latest functionality of the ever-changing web services protocol specifications into applications. It is a set of class libraries that enables developers to easily implement and consume security practices in existing and architected web services implementations. WSE is designed to support security and policy, while also providing for the implementation of new custom security strategies. The specification addresses how a secure context can be maintained over a multi-point message path. The goal of the specification is to provide security, yet the specification is not large is size. New functionality was not created; rather existing functionality from the SOAP specification was leveraged (Chappell, 2003). The model is accessible both through code and policy configuration files, so it is easy for developers and non-developers alike to implement policy features (see example that follows).

WS-Security essentially defines three pieces of functionality:

1. The ability to persist *tokens* in the SOAP header. Tokens are credentials that are used for identification and cryptographic security purposes. Tokens are added to the header of the SOAP message and help to identify the sender to the receiver. The header is able to carry multiple security tokens, each for different receivers, which are distinguished by a URI associated to an actor, where an actor knows which security token to use. Creation and expiration information related to tokens is also stored to help distinguish stale data from valid data. WS-Security does not define how to perform authentication of these tokens, but provides a mechanism such that these tokens can be transported in the SOAP header; receivers can do as they please with the tokens. Agreeing to use WS-Security is not enough to guarantee functionality; parties involved also need to agree on which security tokens are to be used (Chappell, 2003). The specification allows for any type of token but three varieties are most commonly used:

    - *Username tokens*. This token is a representation of a username/password combination that might be used access a web service. The password is defined by an enumeration and can be sent to be plain text, encrypted or not password at all.

    - *Kerberos tokens*. These are tokens generated at a Windows login that perform Windows authentication. Native Windows users and groups can be used for authentication in other applications.

- *X.509 certificate tokens*. This token is a representation of public and private keys that are used in symmetric and asymmetric key exchange algorithms. Certificate information is transported from one destination to another through the SOAP message header.

2. The ability to digitally sign messages, using a private key. The *XML Digital Signature* is a standard allows the inclusion of digital signatures into arbitrary digital content, providing *integrity*, *authentication*, *authorization* and *non-repudiation* to the message (Bartell et al., 2002). The signature binds proof-of-possession to a hash of the message (Smith, 2004). When a message is received, a similar digest is created and the digest included in the message is then compared to the digest created on the receiving end, thus ensuring *integrity*. *Non-repudiation* is provided by signing the message with the private key, which involves mathematically relating the message digest to a (private key) security token. Verification of this signature by the receiver is done with the corresponding a (public key) security token The signature is specified for a number of fields by default and can be configured to sign specific element of a message.

3. The ability to encrypt messages, using a shared public key. *XML Encryption* allows encrypted elements to be included in SOAP messages, which can be designated at the elements or section level to include only specific parts of a document (Imamura T. et al., 2002). By encrypting the message, we are sure that no intermediary can read the sent message. Because SOAP message are XML documents written in plain text and exchanged between two parties, it becomes necessary to mathematically scramble and thus provide *confidentiality* between the sender and the receiver(s). Encrypting the message also helps to provide *integrity* for the transaction.

WS-Security defines a security header in the SOAP message header, to which the different elements of security can be embedded by accessing the SOAP envelope's security *context object*. Implementing various parts of the security protocol is done by adding security tokens, signature elements and encryption elements to this context. There are a set of filters in the framework that implement this functionality. The *output filter* is responsible for applying security-related features on the sending side of the operation. This filter ensures that when security items are added to a message, it translates to secure messages sent across the wire. The *input filter* is responsible for processing security elements on the receiving side, such that when security-related features are added to a message, the receiver can verify the message and contents based on those security tokens. Developers need not concern themselves with code to verify digital signature and encryption/decrypt operations; the framework was designed to handle this automatically. Other filters have been added to the framework to enforce concepts such as policy enforcement, routing rules and referral rules. Discussion of those concepts is beyond the scope of this paper.

IMPLEMENTATION EXAMPLE

As an illustration of the concepts described above, we implemented a secure chat application using secure web services. The original example is taken from a hands-on lab for learning web services (Microsoft, n.d, Skonnard, A., 2003). This example served as a starting point for messaging using web services, where secure messaging capabilities could be easily added to the interface and the underlying code. Because the secure capabilities are accessible

through either code or policy configuration files, it was decided for the purposes of illustration for the developer to use code examples. All code used to add security capability to the application was added to a singleton utility class to illustrate the independence of this code from the business logic and UI.

First, references to the various tokens used to perform security tasks are obtained. Sending a signed and encrypted message involves both a reference to the sender's private key as well as a reference to the receiver's public key.

```
X509SecurityToken  sendToken =
            GetX509Token(false,  PERSONAL_STORE, senderID);

X509SecurityToken  recToken =
            GetX509Token(false,, OTHER_PEOPLE_STORE, receiverID);
. . .
```

Figure 1 - Obtaining token references to the sender's private key and receiver's public key is necessary in order to provide signature and encryption to web services messaging. The GetX509Token helper returns a security token from the specified security certificate store with the second parameter and can be configured to prompt the user to locate a specific certificate with the first argument.

The figure above shows a call to the *GetX507Token* helper method, which is helper method to retrieve a reference of a token by a given id from the specified certificate store. The first argument specifies whether the user is prompted or not to locate the certificate; false here indicates that the method will search the local machine for the certificate. If the argument were set to true, as is used in other parts of the application, the program prompts the user with a popup to choose the desired certificate to be used (see below). The second argument indicates the certificate store location on the local computer. Finally, the third argument is the identification string for the certificate. This is a base64 string that can either be determined through code or it can be derived by using the "X509 Certificate Tool", included with the v2.0 distribution of WSE as a fat client.

With these token references, messages can now be easily signed and encrypted. By adding the sender's private key token to the token collection of the SOAP envelope's security context object, the tokens will be persisted in the SOAP message when it is sent across the wire so that it can be verified on the receiving side. By adding the *MessageSignature* object, the computed digital fingerprint will be computed on the message and included with the message

Encryption is performed by adding an *EncryptedData* object to the SOAP context. The encryption is performed with the recipient's public key such that only the holder of the corresponding private key will be able to decrypt the message. This will place into the body element of the soap message a cipher value of the message content, rather than plain text.

```
. . .
envelope.Context.Security.Tokens.Add( sendToken );
envelope.Context.Security.Elements.Add( new MessageSignature(sendToken)
);

enveloper.Context.Security.Elements.Add( new EncryptedData(recToken) );
. . .
```

Figure 2 – Adding a digital signature to a message sent using web service involves adding the private key security token to the security token collection of the soap envelope context as well as adding a *MessageSignature* object to the security element collection of the soap envelope context. Encrypting of the message involves the addition of a *EncryptedData* object to the security element collection of the soap envelope context. By default, encryption is done on the body of the message and adding other element is configurable.

By comparison, the contents of messages sent with and without security content differ significantly. Messages not using security contain a *<soap:Header>* element, containing a *<wsa:Action>* element, a *<was:MessageID>* element, a *<wsa:To>* element containing the recipient address, and a *<wsa:Security>* element with *<was:Created>* and *<wsa:Expires>* timestamp information. The <soap:Body> element of the message contains a <x:message> element, which includes a <user> element and the <msg> element containing the chat message in plain text.

```
<soap:Envelope …">
    <soap:Header>
      <wsa:Action>urn:chat:message</wsa:Action>
      <wsa:MessageID>uuid:c36cd307-1c79-44df-827f-dbe5d4c30ba7</wsa:MessageID>
      <wsa:To>soap.tcp://bos-grobinson02:6666/lauren</wsa:To>
      <wsse:Security>
        <wsu:Timestamp wsu:Id="Timestamp-b31a6455-4a58-43f4-8560-b6ec8c550913">
          <wsu:Created>2004-12-31T18:01:51Z</wsu:Created>
          <wsu:Expires>2004-12-31T18:06:51Z</wsu:Expires>
        </wsu:Timestamp>
      </wsse:Security>
    </soap:Header>
    <soap:Body>
      <x:message xmlns:x="urn:chat">
        <user>greg</user>
        <msg>hello world</msg>
      </x:message>
    </soap:Body>
  </soap:Envelope>
```

Figure 3 – A message sent using web services a number of interesting feature: a header element with action, message id, to elements and a security element containing creation and expiration timestamp information. The body of the message is in plain text showing the intended recipient and message to be displayed.

The contents of a message with security have additional elements added. In addition to the element described above in the header element, a *<wsse:BinarySecurityToken>* element, a *<xenc:EncryptedKey>* element and a *<Signature>* element have now been added. The *<wsse:BinarySecurityToken>* is a representation of the *X.509* certificate used in signing the message, whose binary string representation is included as the value. The *<Signature>* element contains a *<SignatureValue>* element which holds the computed digest value used as the signature for the message. The *<xenc:EncryptedKey>* element contains a reference to the key used in encryption. The body of the SOAP message no longer contains plain text, as now a *<xenc:EncryptedData>* element with the encrypted string of the body message has been added as well as a *<xenc:CipherValue>* element to verify the integrity of the encrypted data.

```
<soap:Envelope ... /">
   <soap:Header>
   <wsa:Action wsu:Id="Id-1409e277-f549-4570-a135-52e5c0177a85">urn:chat:message</wsa:Action>
     <wsa:MessageID wsu:Id="..">[value omitted] </wsa:MessageID>
    <wsa:To wsu:Id="Id-f880bea5-3123-426c-aeee-15f006f72a4f">
                              soap.tcp://bos-grobinson02:6666/lauren</wsa:To>
    <wsse:Security soap:mustUnderstand="1">
     <wsu:Timestamp wsu:Id="Timestamp-8ec9d2fa-a7fa-4d34-bbf5-7e4d1e795b7a">
        <wsu:Created>2004-12-31T18:04:44Z</wsu:Created>
       <wsu:Expires>2004-12-31T18:09:44Z</wsu:Expires>
     </wsu:Timestamp>
     <wsse:BinarySecurityToken              …
          wsu:Id="SecurityToken-0209ebb7-d2e0-40f0-9914-5295d4335c52">[value omitted for length]
     </wsse:BinarySecurityToken>
     <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">  …
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <wsse:SecurityTokenReference>
          <wsse:KeyIdentifier>[value omitted for length]</wsse:KeyIdentifier>
```

```
        </wsse:SecurityTokenReference>
      </KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>[cipher value omitted for length]
        </xenc:CipherValue>
      </xenc:CipherData> …
    </xenc:EncryptedKey>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo> …
        <Reference URI="#Id-1409e277-f549-4570-a135-52e5c0177a85"> …
          <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
          <DigestValue>+6WBUNhaA1sG+nfayAfuGOiYxJ8=</DigestValue>
        </Reference>
      </SignedInfo>
      <SignatureValue>[cipher value omitted for length]</SignatureValue>
        <KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#SecurityToken-0209ebb7-d2e0-40f0-9914-5295d4335c52"
              ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
                        1.0#X509v3" />
          </wsse:SecurityTokenReference>
        </KeyInfo>
    </Signature>
  </wsse:Security>
 </soap:Header>
 <soap:Body wsu:Id="Id-4b6736bc-fb8f-4b8d-b4cc-3c3a9ead6796">
  <xenc:EncryptedData Id="EncryptedContent-b0a0cd6d-b414-4dad-ba24-2e5cb9f7bb05"
                      Type="http://www.w3.org/2001/04/xmlenc#Content"
                      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
    <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
    <xenc:CipherData>
      <xenc:CipherValue>[cipher value omitted for length]`</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
 </soap:Body>
</soap:Envelope>
```

Figure 4 - A message sent using web services contains a number of elements not found in message sent using unsecure web services. The x.509 certificate is included in the *<wsse:BinarySecurityToken>*, signature data is contained in the *<SignatureVlaue>* element and the body of the message now includes *<xenc:EncryptedDat>* and *<xenc:CipherData>* element with the encrypted message an integrity check information. Some elements have been highlights to make for easy reading. Some values have been omitted to save space.

Verifying the messages on the receiving side involves a two step process. The first is to verify that *MessageSignature* and *EncryptedData* objects have been added to the security token collection of the envelope's context, just as they were added to implement security. This verification step determines that the necessary parts are included in the message. The digital signature itself is automatically verified by the framework and the message is decryption as long as the corresponding private key of the public encrypting key is supplied. The second step is to verify the e-mail of the sender to be sure it is the one known by the receiver. Since any user with a chat client can send a message, this step is necessary to validate that the user is who they claim to be.

Implementation of secure messaging capability can be performed in relatively few lines of code, not including helper methods to get references to the appropriate tokens. What's more, instead of having to write code to implement security, the v2.0 implementation of WSE ships with a 'Configuration Editor' that allows a programmer to define policy through a configuration file, rather than having to explicitly write code. The same code example can easily be

implemented with a policy configuration files and can be used on any platform implementing WS-Security.

COGNITIVE WALKTHROUGH

Now that the basics of security in web services have been discussed, we will now examine a simple user interface (UI) example which uses the underlying WS-Security implementation. Because of the few lines of code needed to implement security, a simple UI configuration can be implemented with a few lines of code that is independent of the underlying code managing the secure messaging. Adding secure messaging to the application is a matter of calling a helper method in a singleton utility class whose sole responsibility is to add and verify security of the messaging.

We first examine the main chat interface. On this screen is a 'Configuration' menu item, some text about the web service listening URL, a user configuration section, and areas used to send messages and view the log of the current conversation. The initial screen is designed in such a way that makes it clear to the user the purpose of each element.

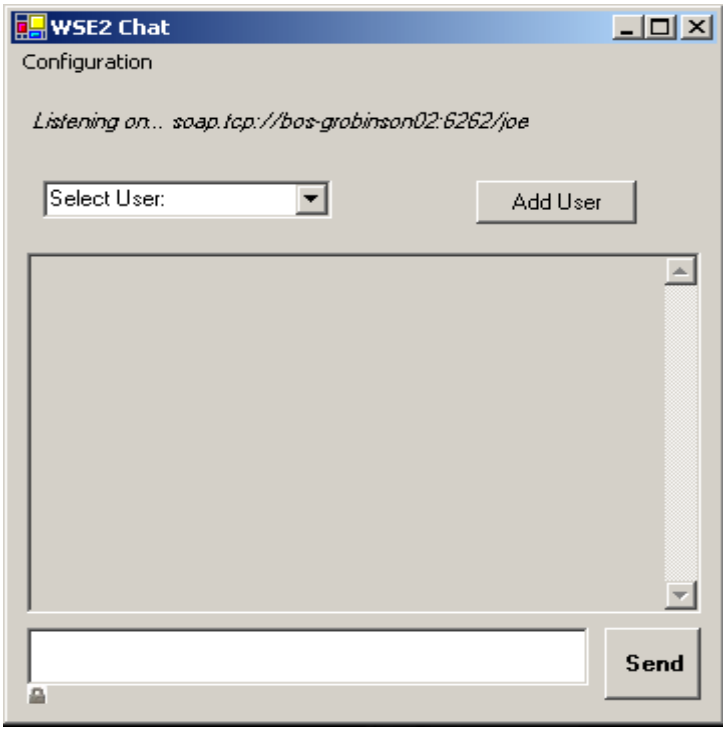

Figure 5 – The main chat interface. The functionality is clear to the user: a configuration menu, information about the connection, a user list, a button to add new users, a message log, a message box with a lock icon indicating secure chat and finally a button to send messages to a user.

Clicking on the 'configuration' menu item, a new window popup is presented where the user can change options for the username and port number values, as well as options in the 'Securities Settings' area. In this area is a check box for using security during chat as well as configuration for the sender's private key and a text blurb about the use of the private key id.

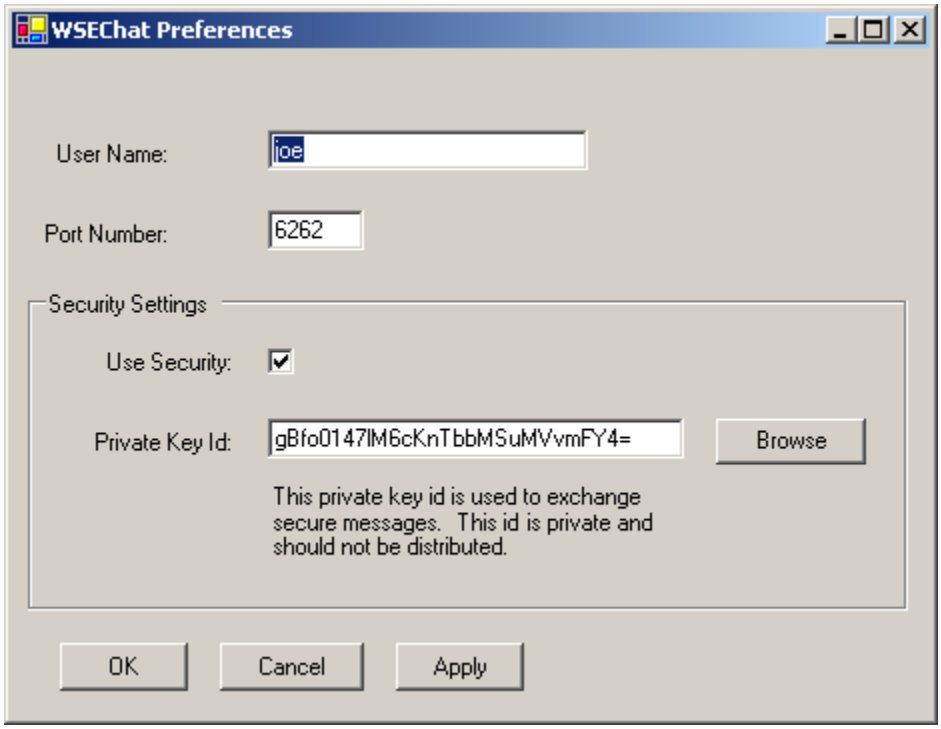

Figure 6 – This window is the configuration window used to change a user's preferences. They are allowed to alter the user name and port number as well as some security settings. The security option are whether or not to use security and the private key id needed to sign messages.

The checkbox enabling security in the chat application is checked by default, thus requiring no action on the part of the user (Yee, K.P., 2002). Corresponding to this checkbox option is an icon on the main chat page indicating the configuration. When selected, a small lock icon is displayed to the bottom left of the message box and when disabled, thus not communicating over a secure channel, an unlocked icon is shown. Since the user types from left to right, this is a logical spot to indicate the status of the security. The meaning of this icon is logical and it would be hard for a user not to notice the icon.

A user will experience problems if they were to disable of security for the application because nothing informs the user of the consequences of such an action. When security is disabled, secure chat can no longer take place between two parties due to the verification scheme built into the application. An enhancement to the user's experience that could be added to the UI would be a popup prompting the user of the consequences of the action. The user must never want to subvert the security of a system designed for secure chat.

Figure 7 – Icons displayed on the message box of the chat interface indicating the security state of the application. The lock on the left indicates that security measures are being used. The unlocked icon on the right indicates security measures have been removed from messaging.

The label for the key id and the accompanying message is perhaps not sufficient for novice users. Instead of showing the key id which has little meaning to users, it would be better to extract information from the certificate, perhaps the issuer and/or the e-mail address, to display to the user. The information as it is displayed now involves the user in too much of the PKI concepts and should be more abstracted (Sasses, M.A., 2004).

The user must configure certificate in two places in the application: specifying their own private key and specifying public keys for users added to the buddy list. To configure a certificate, users click the 'Browse' button resulting in a popup window from which the user chooses the proper certificate. Two potential problems for the users may result from the present design. First, there is no indication to the user about the store location browsed in the dialogue. Users should be notified to avoid any confusion. Second, it is assumed that the user will always want to keys from the default, hard-coded store location; the user is not allowed to change that location if the certificate exists in another location. Perhaps a better design here would be to default the user to one location and place a drop down with other locations on the dialogue window. This would allow novice users choose defaults while allowing more advanced users the freedom to configure as they wish. This would provide flexibility if the user does not store certificates in the same locations as the application expects to find the specific certificates.

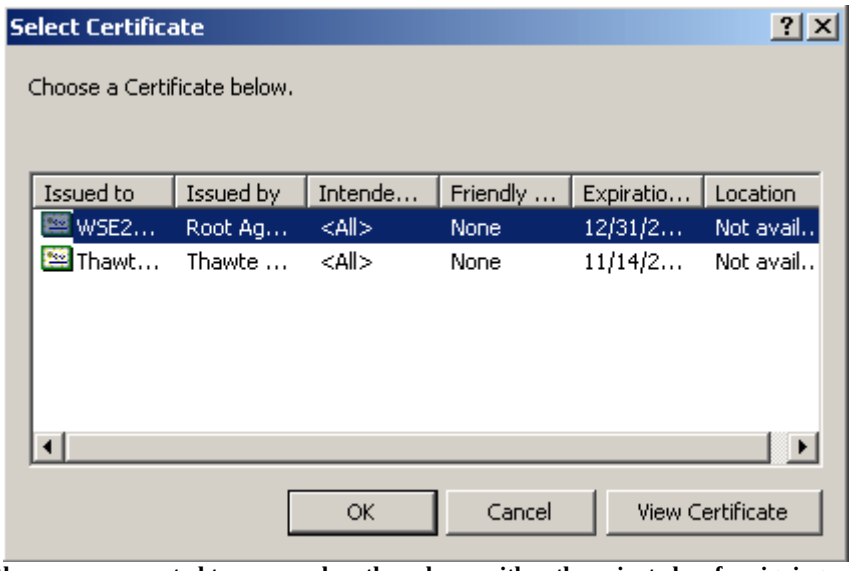

Figure 8 – This is the popup presented to a user when they choose either the private key for signing or the public key for other users used when encrypting messages.

In order to chat with other users, it is necessary to add other users' configuration data to the application by clicking the 'Add User' button. By adding the user name, URL, port number and public key of the remote user and clicking 'OK', the user is added to the user dropdown on the main page. A possible area of confusion to the user would be no indication of the user's state in the UI, and thus no indication that a message will or will not be received. Perhaps a better implementation here would be to include some type of status for each user in the list. Because the model used here does not involve a central server, a protocol should be developed to determine the status of users in the buddy list. This would clearly indicate to the user who is available to receive messages. Once added to the list, communication occurs by choosing them from the list, typing a message in the box and clicking the 'Send' button. A log of the exchanges maintained in the UI such that users can easily see the history of communications.

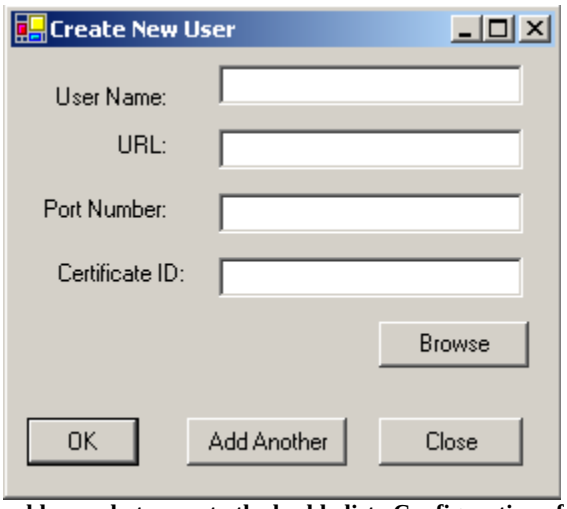

Figure 9 – This is the interface to add new chat users to the buddy list. Configuration of the user name, url, port number and public key certificate information is set for each user in the user list.

Some suggestions for future work that would draw the attention of users and thus possible adoption of users to a secure chat application would be:

- Most importantly, fix the usability issues mentioned in this paper.
- Multiple user chat. This would be simply configuring the UI and sending the message to multiple recipients. All that is required for security is the inclusion of multiple security tokens.
- The ability to interface with existing chap applications such as Trillian, where one client communicate through several chat protocols, as well as existing protocols such as Yahoo, MSN, AOL, ICQ, etc. Trillian is a third party chat application on the market with secure chat capabilities. Unfortunately, not much literature exists on the implementation. Secure chat would be for users of this protocol but they would still be able to communicate with users who have not yet embraced secure chat. This will contribute significantly to adoption.

CONCLUSION

Smetters and Grinter (2002) argued that standardization of underlying software security patterns and API's could assist developers less concerned with security than with their own business domain specialties to create usable, secure software. The WS-Security standard is definitely a step in the right direction. We have demonstrated here that implementation PKI in a TCP transport software system can be accomplished in very few lines of code. As vendors continue to build on this model, increasingly sophisticated API's for WS-Security are being created. In 2006, Microsoft will release Indigo which is an updated API for distributed programming that will use WS-Security to formalize bi-directional web services over the more easily implemented HTTP transport. This honing of the WS-Security API should provide an even more easily programmed communications and security environment.

Despite all of this progress, one critical element of usable software security that has gone unaddressed is the problem of public and/or secret (in symmetric cryptography) key exchange. It is even listed as a "Non-Goal" in the WS-Security standard (Nadalin et al., 2004). It would serve the field well for the standard to be augmented or complemented by another 'WS-*' standard to address and standardize protocols for key exchange. Doing so would complete the challenge

presented by Smetters and Grinter (2002) and would allow developers to create secure software without diverting their skill sets away the business domain and user interface problems they are tasked to solve.

References

1. Smetters, D. K., Grinter, R. E. (2002). Moving from the Design of Usable Security Technologies to the Design of Useful Secure Applications. *Proceedings of the 2002 workshop on New security paradigms*, Virginia Beach, Virginia, September 23 – 26, 2002, pp. 82 – 89, New York, NY: ACM Press.
2. Josang, A., Saderud, G. (2003). Security in Mobile Communications: Challenges and Opportunities. *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, Adelaide, Australia, pp. 43 – 48, Darlinghurst, Australia, Australia: Australian Computer Society, Inc.
3. Yee, K. P. (2002). User Interaction Design for Secure Systems. *Lecture Notes In Computer Science*, 2513, pp. 278 – 290, London, UK: Springer-Verlag.
4. Henry, D. (1999). Who's Got the Key? *Proceedings of the 27th annual ACM SIGUCCS conference on User services*: *Mile high expectations*, Denver, Colorado, 1999, pp. 106 – 110, New York, NY: ACM Press.
5. Shigetomi, R., Otsuka, A., Imai, H. (2003). Anonymous Authentication Scheme for XML Security Standard with Refreshable Tokens. *Proceedings of the 2003 ACM workshop on XML security*, Fairfax, Virginia, October 31, 2003, pp. 86 – 93, New York, NY: ACM Press.
6. Flechais, I., Sasse, M.A., Hailes, S.M. (2003) Bringing Security Home: A process for developing secure and usable systems. *Proceedings of the 2003 workshop on New security paradigms*, Ascona, Switzerland, 2003, 49 – 57, New York, NY: ACM Press.
7. Skonnard, A. (September, 2003). Introducing the Web Services Enhancements 2.0 Messaging API. *MSDN Magazine*, Volume 18, Number 9. Retrieved January 04, 2005 from http://msdn.microsoft.com/msdnmag/issues/03/09/XMLFiles/default.aspx.
8. Bull, L., Stanski, P., Squire, D.M. (2003). Content Extraction Signatures using XML Digital signatures and Custom Transforms On-Demand. *Proceedings of the twelfth international conference on World Wide Web*, Budpest, Hungary, 2003, pp. 170 – 177, New York, NY: ACM Press.
9. Januscewski, K. (March, 2004). Writing Asynchronous, Bidirectional, Stateful, Reliable Web Services with Indigo. Retrieved on January 4, 2005 from the MSDN Library at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnlong/html/indigoattrprog.asp.
10. Whitten, A. & Tygar, J.D. (1998). Usability of Security: A Case Study. Technical Report CMU-CS-98-155, Carnegie Mellon University, December, 1998. Retrieved on January 4, 2005 from http://citeseer.ist.psu.edu/cache/papers/cs/3321/http:zSzzSzreports-archive.adm.cs.cmu.eduzSzanonzSz1998zSzCMU-CS-98-155.pdf/whitten98usability.pdf.
11. Damiani, E., di Vimercati, S., Samarati, P. (2002). Towards Securing XML Web Services. *Proceedings of the 2002 ACM workshop on XML security*, Fairfax, Virginia, 2002, pp. 90 – 96, New York, New York: ACM Press.
13. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (Eds.) Web Services Architecture. W3C Working Group Note 11 February 2004. Retrieved on January 4, 2005 from http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

14. http://www.research.ibm.com/journal/sj/403/benantar.html
15. Nadalin, A., Kaler, C., Hallam-Baker, P., Monzillo, R. (Eds.) (2004). *Web Services Security: SOAP Message Security 1.0*. OASIS Standard 200401, March, 2004. Retrieved on January 4, 2005 from http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.
16. http://www.w3schools.com/soap/default.asp SOAP tutorial
17. http://www.microsoft.com/downloads/details.aspx?FamilyId=89457A1C-D31E-4A9B-92B7-645C1C208A2F&displaylang=en Hands-on Lab 33, Messaging
18. Daemen, J., Rijmen, V. (2003) AES Proposal: Rijndael, Amended version, September 2003. Retrieved January 4, 2004 from http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.
19. American Institute of Certified Public Accountants, Chartered Accountants of Canada. (2000). *AICPA / CICA WebTrust Program for Certification Authorities Version 1.0*. Retrieved on January 4, 2005 from http://www.cpawebtrust.org/CertAuth_fin.htm.k
20. Garfinkel, S. and Spafford, G. (2002). *Web Security, Privacy & Commerce*. (2nd ed.) Sebastopol, CA: O'Reilly & Associates.
21. Chappell, David. MSDN magazine, April 2003 from http://msdn.microsoft.com/msdnmag/issues/03/04/WS-Security/ .
22. Bartel, M., Boyer J., Fox B., LaMacchia B., and Simon E. (February 12, 2002) XML-Signature Syntx and Processing. (Online), retrieved Jan 2, 2005 from http://www.w3.org/TR/xmldsig-core/
23. Imamura T., Dillaway B., and Simon E. (December 10, 2002) XML Encryption Syntax and Processing. Retrieved January 2, 2005 from http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/
24. Microsoft Corporation. (n.d.) Web Services Enhancements 2.0 Hands On Lab – Messaging. Retrieved November 29, 2004 from http://www.microsoft.com/downloads/details.aspx?FamilyId=89457A1C-D31E-4A9B-92B7-645C1C208A2F&displaylang=en
25. Sasse, M. Angela. (April 6. 2004) Usability and trust in information systems. Cyber Trust & Crime Prevention Project.
26. Smith, D., (August, 2004) WS-Security Drilldown in Web Services Enhancements 2.0. MSDN Libray. Retrieved November 29, 2004 from http://msdn.microsoft.com/webservices/building/wse/default.aspx?pull=/library/en-us/dnwse/html/wssecauthwse.asp#wssecauthwse_topic5