

# An Advisor for Web Services Security Policies

Karthikeyan Bhargavan    Cédric Fournet    Andrew D. Gordon    Greg O'Shea

Microsoft Research, Cambridge  
{karthb, fourn, adg, gregos}@microsoft.com

## ABSTRACT

We identify common security vulnerabilities found during security reviews of web services with policy-driven security. We describe the design of an advisor for web services security configurations, the first tool both to identify such vulnerabilities automatically and to offer remedial advice. We report on its implementation as a plugin for Microsoft Web Services Enhancements (WSE).

## Categories and Subject Descriptors

D.4.6 [Security and Protection]: Verification

## General Terms

Security, Languages, Verification

## Keywords

Web Services, XML Security, WS-Security, Policy-Driven Security

## 1. INTRODUCTION

*SOAP Web Services and XML Rewriting Attacks.* Programmatic access to websites or to other remote servers can be achieved by exchange of XML messages. Websites offering such interfaces are called *web services*. SOAP [28] is a widely implemented standard for the messages sent to and from web services. A SOAP message consists of a body—the payload—plus headers containing routing or sequencing data, for example.

SOAP messages may be secured at the transport layer, using mechanisms such as TLS/SSL. Alternatively, for more flexible end-to-end protection, WS-Security [20] defines a SOAP security header that can provide confidentiality and integrity properties via a wide range of cryptographic mechanisms.

Web services may be vulnerable to the same classes of attack—such as script injections or buffer overruns—as other websites [23, 29]. Their use of XML may allow DOS attacks on XML parsers, such as DTD bombing [22]. Moreover, due to the flexibility of SOAP-level security mechanisms, web services may be vulnerable

to a distinct class based on the malicious interception, manipulation, and transmission of SOAP messages, a class we refer to as *XML rewriting attacks*. Used correctly, WS-Security can prevent XML rewriting attacks; this paper presents a new tool to help detect incorrect uses of WS-Security in SOAP processors.

*Security Policies and their Failure Modes.* Web services and their clients are typically written in strongly typed, compiled languages like Java or C#, and rely on SOAP libraries to help construct and process messages and headers. To allow parameters to be adjusted after deployment and without recompilation, clients and servers load configuration files, typically in XML, at runtime. In some systems, part of this configuration data constitutes a formal *security policy*, in the sense that it governs how WS-Security is applied to incoming and outgoing messages. The WS-\* family includes specifications, chiefly WS-SecurityPolicy [11], providing an XML syntax for declarative security policies. A policy is essentially a logical predicate on SOAP messages, determining which message parts must be present, signed, or encrypted. Web Services Enhancements (WSE) [19] is a SOAP library, implementing WS-Security and other specifications, that uses WS-SecurityPolicy as part of its configuration data.

The separation of application code from security policy afforded by WS-SecurityPolicy is good, as it allows much of the security critical part of the system sources to be easily identified for auditing and review. Still, WS-SecurityPolicy is essentially a domain-specific language for selecting cryptographic communications protocols, a famously tricky domain. Security reviews of policy samples for pre-release versions of WSE revealed vulnerabilities to a range of XML rewriting attacks, including replay, man-in-the-middle, and dictionary attacks. Hence, given the general difficulty of uncovering security vulnerabilities by testing, there is a need for domain-specific tools to check for vulnerabilities in web services security policies.

*Formal Tools for Web Services Security.* The threat model we consider for web services, “the network is the attacker”, originates with Needham and Schroeder’s original work on authentication protocols [21]; it is a conservative but realistic threat model for protocols over an open network. Dolev and Yao [12] were the first to formalize this threat model; by now, there are several automatic tools to check security properties of cryptographic protocols against this model. Tool chains based on TulaFale and ProVerif [6, 5, 2, 4] and also Casper and FDR [18, 26] are being applied to check detailed formal models of WS-Security protocols, and to clarify their semantics. These studies have revealed design and implementation flaws in web services protocols. Still, as the formal models are handwritten, critical details could be missing, so proofs show-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SWS’05, November 11, 2005, Fairfax, Virginia, USA.  
Copyright 2005 ACM 1-59593-234-8/05/0011 ...\$5.00.

ing the absence of attacks do not directly apply to the corresponding implementations.

To narrow the gap between model and implementation, in previous work [3] we built tools to extract TulaFale models from WSE configuration files. To the best of our knowledge, ours is the only tool chain that formally checks security properties of implementation files driving web services. It has been invaluable for checking properties of example policies distributed with WSE. Still, a limitation shared with other automatic tools is that the diagnostic messages are low-level, so that non-experts find them difficult to interpret and to act upon.

*This Paper: WSE Policy Advisor.* We describe a rule-based tool for detecting typical errors in WSE configuration and policy files. As shown in Figure 1, the WSE Policy Advisor generates a security report by running queries that check for over thirty syntactic conditions corresponding to errors found during security reviews. These reviews involved testing, modelling, and formal analyses of policy-based clients and services. Diagnostic messages for each condition explain the risk and suggest appropriate remedial action, making the tool appropriate for non-expert users. The tool offers no formal guarantees; we recommend its use as part of the systematic threat modelling [24] of a WSE installation. Our experience of security reviews suggests the tool is likely to find errors that otherwise might be overlooked.

The main contributions of this paper are to describe a range of failure modes found empirically during security reviews, and to describe the architecture of the first tool both to detect such errors and to suggest corrective action.

The tool is available at <http://research.microsoft.com/downloads>. A description of the tool accompanies an implementation of the WS-I Basic Security Profile sample application [17].

*Organization of the Paper.* Section 2 reviews WS-Security and illustrates its risks in the context of a typical SOAP request. Section 3 describes security policies. Section 4 presents the design and implementation of our advisor tool. Section 5 surveys the main failure modes detected by the tool. Section 6 discusses related work and concludes.

## 2. AN XML REWRITING ATTACK ON A WEB SERVICE

In this section, we set up a concrete running example, recall some features of WS-Security, and describe an error that leads to a typical XML rewriting attack.

Consider a simple SOAP-based server that responds to requests for the latest prices of a list of stocks. The server charges a subscriber's account for each quote and does not wish to respond to non-subscribers. Hence, it requires that each request be authenticated by a message signature generated using an X.509 certificate belonging to one of its subscribers. Furthermore, it requires that each request include a unique message identifier to be cached to detect message replay. So a typical request is a SOAP envelope with the following structure (eliding some headers, all namespaces, and abbreviating long strings for clarity):

```
<Envelope>
  <Header>
    ...
    <MessageID Id="Id-1">uuid:21c81...</MessageID>
    <Security mustUnderstand="1">
      <BinarySecurityToken ValueType="...#X509v3"
        Id="Id-2">
        MIIBxDCCAW6g...
```

```
</BinarySecurityToken>
      <Signature (detailed below)
    </Security>
  </Header>
  <Body Id="Id-3">
    <StockQuoteRequest>...</StockQuoteRequest>
  </Body>
</Envelope>
```

This message consists of a body, containing a list of stock symbols, plus a header, containing a message identifier and security elements. The `<MessageID>` element is defined by WS-Addressing [7] as a means of uniquely identifying requests and correlating them with responses. The `<Security>` element is defined by WS-Security, and here contains an X.509 certificate encoded as text within a `<BinarySecurityToken>` element, and a `Signature` element that signs the message body and the message identifier using the key associated with the certificate.

The `Signature` has the following structure, as defined in XML-Signature [13]:

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="...xml-exc-c14n#" />
    <SignatureMethod Algorithm="...#rsa-sha1" />
    <Reference URI="#Id-1">
      <Transforms>
        <Transform Algorithm="...xml-exc-c14n#" />
      </Transforms>
      <DigestMethod Algorithm="...#sha1" />
      <DigestValue>d5AOd...</DigestValue>
    </Reference>
    <Reference URI="#Id-3">
      <Transforms>
        <Transform Algorithm="...xml-exc-c14n#" />
      </Transforms>
      <DigestMethod Algorithm="...#sha1" />
      <DigestValue>zSzZT...</DigestValue>
    </Reference></SignedInfo>
    <SignatureValue>e4EyW...</SignatureValue>
  <KeyInfo>
    <SecurityTokenReference>
      <Reference URI="#Id-2"
        ValueType="...#X509v3" />
    </SecurityTokenReference></KeyInfo>
</Signature>
```

This element represents a joint signature of the body and message identifier of the envelope. It has three child elements. The first, `<SignedInfo>`, specifies both the parts of the message that are signed and the algorithms used in the computation. Each message part is specified by a `<Reference>` element, whose `URI` attribute is a fragment URI referring to the `Id` attribute of the message part, and whose `<DigestValue>` element is a cryptographic digest of the message part. In this case, the `<SignedInfo>` element has two `<Reference>` elements, referring to the message identifier and body, and each digest is obtained by canonicalizing the message part with the exclusive canonicalization algorithm ("xml-exc-c14n") and then hashing with the SHA-1 algorithm. The second element of the signature, `<SignatureValue>`, contains the outcome of first canonicalizing the `<SignedInfo>` element, and then signing with the RSA-SHA-1 algorithm. Finally, the `<KeyInfo>` element indicates the X.509 certificate containing the signature's verification key.

The signature above is meant to authenticate the client as well as to protect the message identifier and body against tampering by an active adversary. However, there is an XML rewriting attack where an adversary can prevent the message identifier from being

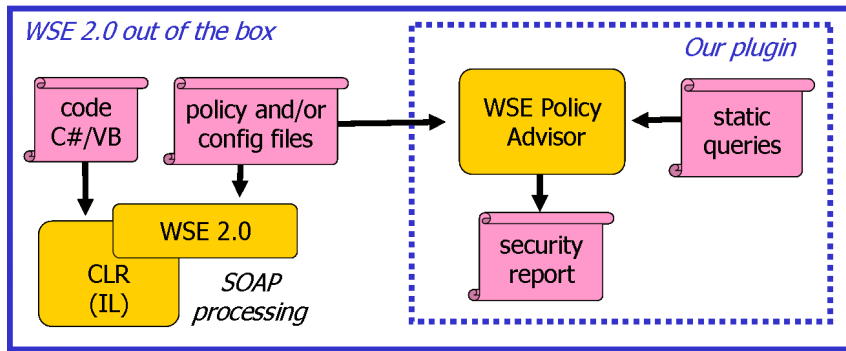


Figure 1: WSE Policy Advisor

processed or cached at the server. The attack stems from two factors. First, the digital signature references message parts by their `Id` attributes but says nothing of their location in the message. Second, the message identifier is optional.<sup>1</sup> Hence, the attacker can rewrite the message to obtain the following:

```
<Envelope>
  <Header>
    ...
    <Bogus>
      <MessageID Id="Id-1">uuid:21c81...</MessageID>
    </Bogus>
    <Security mustUnderstand="1">
      <BinarySecurityToken ValueType="...#X509v3"
        Id="Id-2">
        MIIBxDCCAW6g...</BinarySecurityToken>
      <Signature>
    </Security>
  </Header>
  <Body Id="Id-3">
    <StockQuoteRequest>...</StockQuoteRequest>
  </Body>
</Envelope>
```

Here, the `<MessageID>` element is moved into a new, bogus header element; everything else, including the certificate and signature, remains unchanged. The `<Bogus>` element and its contents are ignored by the recipient (correctly, since this header is unknown), but the signature remains acceptable because the element at reference URI "Id-1" still exists in the message and still has the same value. Assuming that the replay detection mechanism is only triggered by the presence of the optional `<MessageID>` element, the recipient accepts the message as authentic. Thus, the attacker has bypassed the replay detection checks and, by replaying the message (or some variants of the message), may cause the same request to be processed several times, making the client subscriber pay several times for the same query and forcing the server to do redundant work.

### 3. WEB SERVICES SECURITY POLICIES

The specifications WS-Policy [8], WS-PolicyAssertion [9], and WS-SecurityPolicy [11] define a declarative XML format to express how web services implementations construct and check SOAP messages. In this format, a *policy* is a propositional formula with

<sup>1</sup>According to WS-Addressing, a message identifier must be present in a request if a reply is expected. However, it is often difficult to tell whether an envelope belongs to a one-shot or to a request-reply pattern. Hence, implementations, such as WSE, treat message identifiers as optional.

disjunctions and conjunctions built from any set of *base assertions* that define predicates on SOAP messages. WSE is one of the first SOAP libraries to support policies. In principle, it is possible to write complex, deeply-nested policies with both standard and custom base assertions. (A detailed discussion of the expressiveness and limitations of this policy language appears in an earlier work [3].) In practice, however, the policies used with WSE typically consist of a single conjunction built from a set of five base assertions. Our tool focuses on three of these assertions: the message predicate assertion from WS-PolicyAssertion, and the integrity and confidentiality assertions from WS-SecurityPolicy.

- A *message predicate* assertion lists the message parts that must be present in an envelope. For instance, if an application supports WS-Addressing, this assertion can require that in addition to the body, the addressing headers, `<To>` and `<Action>`, are included in every request message. We then say that these parts are *mandatory*. In some cases, such checks may also be present at other layers. For instance, any WSE application will refuse to accept a message without a SOAP body, so including the body in a message predicate assertion may seem superfluous. Still, it is useful to document and check for the presence of the mandatory message parts at this layer, to avoid further processing, and to provide uniform error messages.
- An *integrity* assertion requires that the envelope include a digital signature; it lists the message parts to be jointly signed, if present in the envelope, and describes the security token to be used to generate the signature. We say that the message parts listed in an integrity assertion are *signed-if-present*.
- A *confidentiality* assertion lists the message parts that must be encrypted if present in the envelope, and describes the security token to be used for this encryption. We say that the message parts listed in a confidentiality assertion are *encrypted-if-present*.

As an example, here is a policy that can be used with WSE to generate and check the request messages from the previous section:

```
<Policy Id="PolicyReq">
  <MessagePredicate>
    Body() Header(To) Header(Action)
  </MessagePredicate>
  <Integrity>
    <TokenInfo>
      <SecurityToken>
        <TokenType>...#X509v3</TokenType>
        <TokenIssuer>CN=Root Agency</TokenIssuer>
```

```

    </SecurityToken>
  </TokenInfo>
  <MessageParts>
    Body() Header(To) Header(Action)
    Header(MessageID) Timestamp()
  </MessageParts>
</Integrity>
</Policy>

```

This policy (**PolicyReq**) is the conjunction of a message predicate assertion and an integrity assertion. The first assertion requires the presence of the three mandatory message parts: the body and the **<To>** and **<Action>** headers. The second assertion requires a signature, using an X.509 token issued by ‘CN=Root Agency’, that covers the body, the **<To>** and **<Action>** headers, the optional **<MessageID>** header, and the WS-Security **<Timestamp>**.

Different applications have different security requirements. Some applications may require the presence of a header that is not mandatory according to any standard. They must then take care to include this optional or custom header in a message predicate assertion. In the example policy above, the service is relying on the presence of the **<MessageID>** header for replay protection but the header is not included in the **<MessagePredicate>** assertion. This oversight enables the XML rewriting attack described in Section 2. Including **Header(MessageID)** in the message predicate assertion avoids this attack.

Some applications may require the secrecy of the body contents, to avoid information leaks to eavesdroppers. We stipulate encryption by adding a confidentiality assertion, requiring use of an X.509 public key certificate issued by a particular authority:

```

<Policy>
  ...
  <Confidentiality>
    <TokenInfo>
      <SecurityToken>
        <TokenType>...#X509v3</TokenType>
        <TokenIssuer>CN=Root Agency</TokenIssuer>
      </SecurityToken>
    </TokenInfo>
    <MessageParts> Body() </MessageParts>
  </Confidentiality>
</Policy>

```

Having written such a policy, we can then configure the client or service application to enforce it on all request (or response) messages. This configuration is not part of the WS-Policy specification and is implemented in a custom way by WSE. Each WSE policy file contains a set of *mappings* and a set of WS-SecurityPolicy policies. Each mapping associates a SOAP message to a policy, based on the service endpoint of the message, and whether the message is a request to that endpoint, or a response or fault from that endpoint. For instance, the policy file for the example stock web service has a mapping section as follows:

```

<mappings>
  <endpoint uri="http://.../StockService.asmx">
    <operation requestAction=".../StockQuoteRequest">
      <request policy="#PolicyReq" />
      <response policy="#PolicyResp" />
      <fault policy="#PolicyFault" />
    </operation>
  </endpoint>
</mappings>

```

It identifies an endpoint with a service URI and an operation URI; these are the values that appear within the **<To>** and **<Action>** headers of all requests sent to the service. It maps request messages sent to the endpoint to the first policy in this section (**PolicyReq**)

and maps response and fault messages to other policies (**PolicyResp** and **PolicyFault**) not shown here.

## 4. WSE POLICY ADVISOR

As illustrated by the previous sections, subtle mistakes in policies can lead to exposure to XML rewriting attacks. Policies written by WSE users exhibit a range of such errors, suggesting that the threats and security mechanisms are often not fully understood.

In response, we propose a tool, WSE Policy Advisor, for use by developers, testers, or operators of WSE installations. Our tool provides both a commentary on the positive security guarantees provided by a collection of policies, and advice on potential security vulnerabilities. This section describes the architecture and usage of the tool, and discusses its limitations and validation.

*Input Files and Usage.* WSE can be configured as part of the SOAP configuration on a web service or on a client. In the case of a service, WSE is configured by a **web.config** file and in the case of a client, by an **app.config** file. In either case, the configuration file may reference a policy file. WSE provides a policy editor tool, that can be run directly or from within the Visual Studio development environment, for interactive editing of configuration and policy files.

The WSE Policy Advisor plugs into the WSE policy editor and can be used to check the configuration and policy files being edited. Figure 1 shows the overall architecture. The advisor evaluates a set of queries against these files and generates a security report with commentary, warnings, and advice.

We anticipate three typical usages: (1) early in the design process, as a way of providing immediate feedback as the designer experiments with different WSE settings; (2) as part of the debugging process during development; and (3) as part of security evaluation and review, for instance before deploying amended configurations in production.

*Static Queries.* WSE Policy Advisor evaluates queries either against a single policy file, or against the combination of a configuration file and its associated policy file. Each query is triggered by a syntactic condition (a test that may or may not be satisfied by all or part of the configuration data), and then outputs a risk (a textual report indicating what sort of security vulnerability may exist) and a remedial action (a textual report suggesting how to modify the configuration data to eliminate or reduce the vulnerability).

The triggering conditions divide into four categories:

1. Likely errors in configuration file settings. (For example, acceptance of X.509 certificates used for test purposes, or failure to switch on replay detection.)
2. Lack of conformance to a conservative XML schema for policy files.
3. Likely errors in the mappings associating SOAP messages to policies. (For example, inclusion of a policy for responses but not for faults.)
4. Likely errors in particular policies. (For example, failure to sign or to check for the presence of WS-Addressing headers.)

For instance, to check for the policy error discussed in Section 3, a query in the fourth category checks that **Header(MessageID)** is included in a message predicate assertion whenever it is included in an integrity assertion.

In total, the advisor includes 36 queries (listed in Appendix A), each of which checks for a condition that may lead to a security

vulnerability. Section 5 explains in detail the rationale for these queries and the risks they seek to avoid.

*The Security Report.* WSE Policy Advisor generates a security report as a structured document containing both negative indications, generated by evaluating queries, and positive restatements of the behaviour of the policy file, to help a human user understand its implications and to spot irregularities that may be errors. The report includes a summary of the mapping section of the policy file, showing which policy applies to requests, responses, and faults associated with particular endpoints. Moreover, for each policy, a table summarizes for typical elements found in SOAP messages, whether those elements are mandatory, signed-if-present, or encrypted-if-present.

For example, Figure 2 shows the security report produced by analyzing a configuration file linked to the policy file from Section 3, after removing the response and fault policies for simplicity. This report is in HTML and can be viewed using a standard browser, or within the custom browser of the advisor window. The window divides into three panes. The pane at the lower left-hand-side can be used to navigate around the report: clicking on a branch in the tree updates the two other panes. The pane at the top displays the detailed report. The pane at the lower right-hand-side displays snippets of XML, relevant to the report, that are extracted from the configuration and policy files.

In the figure, the top pane shows a part of the report concerning the policy file shown in the lower right-hand-side pane. It starts with a table summarizing the policy. The first column shows all the headers that must be present, the second shows all that are signed-if-present, and the third shows all that are encrypted-if-present. The table guarantees that all message parts with a ‘yes’ in the first two columns must be present and included in a digital signature, and that all message parts with a ‘yes’ in the first and third columns (none in this case) must be present and encrypted. The report continues with a warning:

```
This policy accepts messages without a <wse:Timestamp>  
or <MessageID> element.
```

This arises from the fifth row of the table: `Header(MessageID)` is signed-if-present, but not mandatory. The warning is followed by a detailed explanation of the security risk, here a description of the attack from Section 2, and advice on possible corrections to the policy that can avoid it. A second warning (not shown above) is that the policy has no confidentiality element and so does not protect the secrecy of the body or headers.

Although the advisor essentially performs syntactic queries on XML configuration files, the security interpretation of the query and the effectiveness of remedial advice are carefully validated, as explained below—this interpretation and validation account for most of the effort of designing new queries.

*Limitations and Experimental Validation.* WSE Policy Advisor reports common errors in simple policies. These errors and their indicated risks can be easily demonstrated on sample WSE applications. We have tested each warning generated by the advisor and exhibited the corresponding attack on a WSE application. We have run the advisor against a test suite including all policy-based samples distributed with WSE, a selection of policies generated by WSE, and a collection of synthetic policies and configuration files for covering the results of all queries.

On the other hand, the advisor does not provide strong security guarantees. Even in the absence of any warnings in the security report, there may still be attacks beyond the limited threat model

considered by the tool. For instance, the web service may still be vulnerable to XPath injection attacks. Moreover, the tool may provide poor advice on complex policies, for instance those with policy assertions that it does not process.

*Validation by Formal Methods.* In earlier work [3], we described tools for automatically generating and formally verifying security policies using a theorem-prover. Although these verification tools operate on a more limited and a slightly different policy language, they are capable of providing strong security guarantees as formal theorems against a realistic threat model. When they fail to prove a security property, they generate an attack trace as a counter-example. These guarantees or attacks can be hard to read and interpret even for experts and require detailed knowledge of the underlying model. In contrast, the analysis performed by the WSE Policy Advisor is more limited than formal verification, and offers no strong guarantees, but its reports are easier to understand.

To benefit from both approaches, we use the formal analyses of policies to inform the queries and positive guarantees provided by the advisor. Most of the policy-level queries implemented by the advisor correspond to attacks uncovered during formal analyses of sample policies. On the other hand, the table that summarizes the integrity and confidentiality guarantees of a policy corresponds to authenticity and secrecy theorems proved by the verification tool: if any header is both present and signed according to the table then, in the model, it is authenticated between client and server. An analogous result holds for encryption.

The policy generation tool in our earlier work [3] also influences the advisor. The remedial advice provided by many of the advisor queries closely resembles the policies generated by the tool. This advice is validated by our security theorems for abstract models of generated policies.

In principle, since the advisor and our formal verification tools consume a similar XML format, one could automate this validation process, formally verifying that all queries correspond to attacks, and all remedies correspond to theorems. This could be made part of the production of new releases of the advisor. We leave the idea as further work.

*Potential Extensions.* There are several features we considered but did not include in the advisor.

- Since the WSE policy language is itself extensible, the advisor should also be extensible. Users should be able to extend the set of queries by, for instance, writing macros in some XML query language.
- For simplicity, the advisor is a pure diagnostics tool; it does not modify the configuration. It could be useful, although delicate, to integrate the advisor in a configuration management tool, with automated support to implement the remedial actions after obtaining permission from the user.
- It would be useful to provide APIs to run the queries automatically (and possibly remotely), as a regular checkup or whenever new queries are defined. The advisor could also partially generate a threat model [24] for a WSE installation, along the lines suggested by Udell [27].
- To extend the scope and the precision of our security queries, the advisor could take advantage of other available information on deployed web services, such as for instance the local certificate store, or logs of SOAP messages.

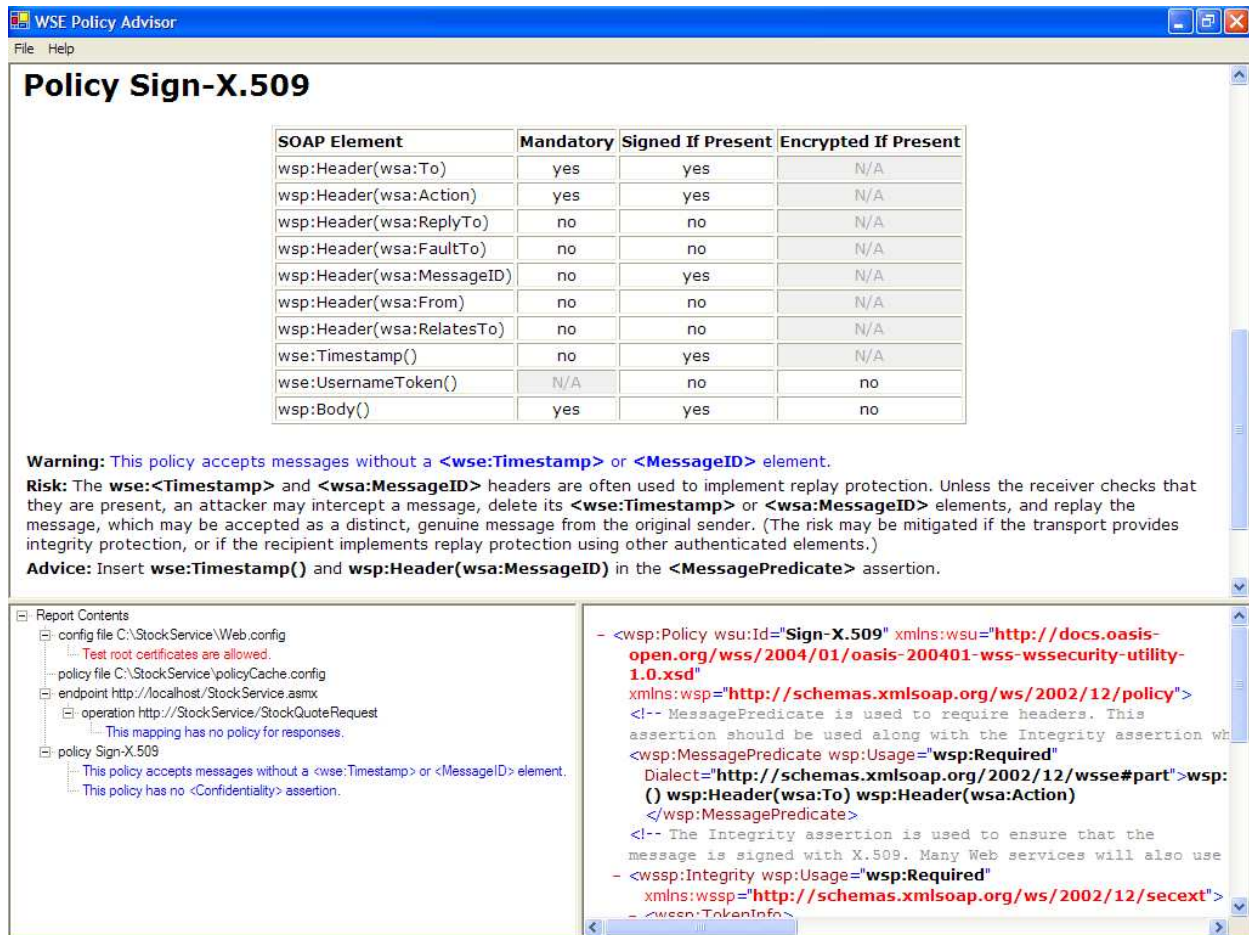


Figure 2: WSE Policy Advisor's Report on Example Policy

## 5. FAILURE MODES QUERIED BY WSE POLICY ADVISOR

This section summarizes the various sorts of failure modes detected by the advisor.

*Absence of Integrity or Confidentiality Assertions.* WS-Security defines a SOAP `<Security>` header that may contain username tokens, X.509 certificate tokens, message signatures, encrypted keys, references to encrypted data, and other items. It is up to a security policy to select appropriate security elements and use them correctly. For instance, if a policy contains no `<Integrity>` element the message is not authenticated. The message body, for instance, can be rewritten by an active attacker. Similarly, if the policy contains no `<Confidentiality>` element, none of the message parts is secret. Any eavesdropper can read the (possibly confidential) message body. Therefore, the advisor includes queries to check for the presence of these assertions.

*Password Vulnerabilities.* Username tokens allow users to authenticate themselves to a server via a public username and a secret password shared with the server. A `<UsernameToken>` element contains the username and may also contain the password, either in the clear or hashed with a nonce, also contained in the username token. Even if the password is not present in the username token, it may be used together with the nonce in the computation

of the key used to construct the XML signature on the SOAP message. Passwords are typically weak secrets with little entropy and should be protected not only from direct leaks to the attacker, as when the password is sent in the clear, but also from indirect leaks, such as when a hash of the password is sent, that enable dictionary attacks on the password. In the end, all policies that allow username tokens in the clear are unsafe: if the password is absent, the username is unauthenticated; if the password is present in the clear, it is directly compromised; if it is present as a digest or as the basis of a signature, it is vulnerable to a dictionary attack.

Consider the following policy that explicitly rejects username tokens that contain a password, whether in the clear or digested:

```

<Policy>
  <Integrity>
    <TokenInfo>
      <SecurityToken>
        <TokenType> UsernameToken </TokenType>
        <wssp:Claims>
          <wssp:UsePassword wsp:Usage="wsp:Rejected"/>
        </wssp:Claims>
      </SecurityToken>
    </TokenInfo>
    <MessageParts> ... </MessageParts>
  </Integrity>
</Policy>

```

This policy protects against direct leaks of the password. However, since the policy requires a message signature based on the

password, the signature can be subjected to a dictionary attack. The advisor warns about this attack, and suggests the inclusion of a confidentiality assertion to encrypt the username token:

```
<Policy>
  <Confidentiality>
    <MessageParts> ... UsernameToken() ... </MessageParts>
  </Confidentiality>
</Policy>
```

*Certificate-Based Authentication.* In addition to username tokens, WS-Security defines the usage of security tokens containing X.509 certificates for signing and encrypting message parts. An important aspect of X.509 certificate validation is to check that the certificate has been issued by a certification authority trusted by the recipient. Otherwise, an attacker can generate any self-signed certificate and get it accepted by the recipient, hence breaking the authenticity and secrecy properties of the message. To avoid such attacks, another query checks for a `<TokenIssuer>` assertion, such as the one below:

```
<Policy>
  <Integrity>
    <TokenInfo>
      <SecurityToken>
        <TokenType>...#X509v3</TokenType>
        <TokenIssuer> TrustedAuthority </TokenIssuer>
      </SecurityToken>
    </TokenInfo>
  <MessageParts> ... </MessageParts>
</Integrity>
</Policy>
```

*Routing Headers.* WS-Addressing defines SOAP headers that are used to route messages to the intended endpoint. For instance, each WSE web method is identified by a service URI, indicating the location of the service class, and an operation URI, indicating the method to call. To comply with WS-Addressing, each request to a web service must have `<To>` and `<Action>` headers containing these URIs for the intended endpoint. If these headers are modified, the message can be redirected to a different web service or method. To avoid such redirection attacks, the policy covering these messages must include an integrity assertion that requires that these headers be signed.

While the `<To>` and `<Action>` headers completely specify the intended recipient of request messages, the routing of response and fault messages is more complicated. Since the web client is typically anonymous and does not advertise a service URI, the web service must be told what URI to put in its `<To>` header. Hence, request messages carry a `<ReplyTo>` header and also, optionally, a `<FaultTo>` header indicating the URIs to which the response or fault message should be sent. If either of these headers can be modified by an attacker, the response or fault message will be redirected to a different endpoint. So, these headers must also be included in the integrity assertion. The advisor includes queries to check all these headers are included within integrity assertions in request policies.

*Message Identifiers.* Since a web services client may make several concurrent requests to the same or different web services, correlating responses with their requests is important. With WSE, this correlation is done on the basis of request message identifiers: each response message includes a `<RelatesTo>` header that echoes the contents of the `<MessageID>` header in the corresponding request. If this `<RelatesTo>` header is not signed, the attacker

can modify it and fool a client into accepting it as a response to an unrelated request. Hence, the policy for response messages must include `Header(RelatesTo)` in its `<Integrity>` assertion. A query checks this for all response policies.

Message identifiers are also useful, along with message timestamps, in detecting and avoiding replays of messages. If a web service, or client, caches all the message identifiers received within a time window and rejects all messages that either have a duplicate message identifier or a stale timestamp, then it avoids message replay attacks. Even without caching message identifiers, timestamps provide a cheap way of avoiding replays outside a time window. However, such mechanisms fail if the `<MessageID>` header or the `<Timestamp>` security header may be tampered with. Hence, the advisor includes a query to check both these headers are included within the integrity assertions of all message policies.

*Schema Compliance.* If a policy is treated as a propositional formula, the semantics of a missing, incorrectly-named, or schema non-compliant policy should be the unsatisfiable proposition: it should be satisfied by no message. However, this behaviour is not specified by WS-Policy, so some pre-release versions of WSE took the reverse approach and accepted some or all messages. To be safe, the advisor includes queries that check that all linked policies are present and schema compliant.

*MessagePredicate Assertion.* As noted in Section 3, a common misconception regarding integrity assertions is that they require the listed message parts both to be signed and present, when in fact they only require these parts to be signed-if-present. An analogous misconception arises for confidentiality assertions. If the intention is to require a header to be both present and signed, the policy must also include a message predicate assertion that requires the presence of the header. Otherwise it enables attacks as in Section 2. Although the message predicate assertion typically only includes optional headers such as `<MessageID>` and `<FaultTo>`, it is good practice to include all headers that are used for message routing or security. Queries check for the presence of a message predicate assertion containing all security-critical headers, which depend on whether the message is a request, response, or fault.

*Security Mapping.* Even the strongest policies are useless if they are not applied to the right messages. If a web service or web method is mapped to no request policy, WSE will drop all messages sent to it. However, if it has an empty request policy (`<request policy=""/>`) attached to it, then all messages sent to it will be sent through unchecked. For safety, all messages must have a minimally strong security policy attached to them. We consider empty policies inherently unsafe. In particular, even faults should have policies attached to them. Otherwise there is a denial-of-service attack on a client where the attacker sends an arbitrary fault message causing the client to close down its web session. For all endpoints mentioned in the policy file, queries check that every request, response, or fault message is mapped to some policy.

*Global Settings.* The WSE configuration file contains entries to customize the behaviour of the inbuilt `UsernameTokenManager`, that checks user passwords, and `X509TokenManager`, that checks the validity of X.509 certificates.

WSE's username token manager has a replay detection feature that can be configured by users. This feature is quite useful as it prevents replays of hashed passwords and password-based signatures without relying on other message replay detection mecha-

nisms. However, if this feature is not enabled, then an attacker can capture and use a username token with a password hash and include it with any number of messages to a server that relies on hashed passwords for (weak) authentication. An advisor query checks that this feature is enabled.

WSE uses the operating system's certificate manager to handle and store X.509 certificates. For specific web services, however, users can choose to disable trust-chain verification, which is unsafe as it accepts any self-signed X.509 certificate. Moreover, during testing, WSE users often enable the test-root authority that is used by the sample certificates provided with the toolkit. This should be disabled before the web service is deployed. Hence, a query checks that the WSE configuration file contains the following element:

```
<x509 allowTestRoot="false"
      allowRevocationUrlRetrieval="true"
      verifyTrust="true"/>
```

## 6. CONCLUSIONS AND RELATED WORK

The problem of XML rewriting attacks on web services is not new; several studies [1, 2, 4, 5, 16, 18, 26] develop formal analyses to help find such vulnerabilities and to rule them out. Still, although these tools can give strong guarantees of correctness within a formal model, there is typically a gap between the model and the implementation. Our previous study [3] of generating and analyzing web services security policies is the only prior work we are aware of that checks actual implementation files for vulnerabilities to XML rewriting attacks. WSE Policy Advisor does the same, but makes a different tradeoff between formal correctness and usability; it offers no formal guarantees, but instead detects typical error patterns, and suggests specific remedial action. Since it looks for particular error conditions, the resulting advice is more specific and actionable than the formal error conditions reported by the theorem prover used in our prior work [3].

This paper contributes a detailed description of the tool's architecture and the typical error patterns it detects. Most of the queries amount to checking conformance to fairly well known prudent practices for cryptographic protocols, and may appear obvious once stated, at least to experts. Still, packaging this advice in a tool is pragmatically more effective than appeal to standard principles in the literature. On the basis of positive feedback from WSE users, we conclude that in spite of the absence of any formal guarantees, WSE Policy Advisor helps users understand and improve their policies.

More generally, the use of XML configuration files is on the rise. Fowler [15] notes this is a consequence of the ease with which parameters can be edited without needing recompilation. Nonetheless, unlike compiled languages, XML-based domain-specific languages enjoy little or no static checking, and so errors are only detected at runtime, if at all. WSE Policy Advisor exemplifies the general idea of a static checker for a domain-specific language; notice that its checks include but go beyond conformance to an XML schema. As XML configuration files proliferate, we expect more instances of this idea.

An alternative to detection of typical failure modes is to re-design to reduce the possibility of error. As implemented in WSE 2, version 1.0 [11] of WS-SecurityPolicy expresses policy in terms of individual headers on individual messages. More recently, version 1.1 [10] expresses policy in terms of higher-level message patterns, as does the policy language being implemented in WSE 3. These design changes eliminate at least some of the possibilities of mis-configuration detected by our tool (which targets WSE 2 configurations).

Another approach is to start from an abstract description of security requirements and to generate lower-level policies; some research tools [3, 25] and WSE itself follow this approach. Static checkers such as our advisor may still find problems with generated policies, as they are often edited by hand after generation.

As well as static checkers, dynamic tools for penetration testing of running web services are starting to appear. For example, WS-Digger [14] attempts attacks such as SQL and XPATH injection, and cross site scripting. We are not aware of any dynamic tools to automate the discovery of XML rewriting attacks.

*Acknowledgements.* We thank all members of the WSE team—and in particular, Keith Ballinger, Mark Fussell, Sidd Shenoy, and Hervey Wilson—for their support of this work. Vittorio Bertocci provided invaluable feedback on an early version of the tool.

## 7. REFERENCES

- [1] M. Backes, B. Pfitzmann, S. Mödersheim, and L. Vigano. Symbolic and cryptographic analysis of the secure WS-ReliableMessaging scenario. Unpublished draft, 2005.
- [2] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *2004 ACM Workshop on Secure Web Services (SWS)*, pages 11–22, October 2004.
- [3] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, October 2004.
- [4] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Computer Science*, 340(1):102–153, June 2005. See also Microsoft Research Technical Report MSR-TR-2003-83.
- [5] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004. Tool available from <http://Securing.WS>.
- [6] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society Press, 2001.
- [7] D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, August 2004. W3C Member Submission, at <http://www.w3.org/Submission/ws-addressing/>.
- [8] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web services policy framework (WS-Policy), May 2003. Version 1.1.
- [9] D. Box, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, P. Patrick, C. von Riegen, and J. Shewchuk. Web services policy assertions language (WS-PolicyAssertions), May 2003. Version 1.1.
- [10] G. Della-Libera, M. Gudgin, P. Hallam-Baker, M. Hondo, H. Granqvist, C. Kaler, H. Maruyama, M. McIntosh, A. Nadalin, N. Nagaratnam, R. Philpott, H. Prafullchandra, J. Shewchuk, D. Walter, and R. Zolfonoon. Web services security policy language (WS-SecurityPolicy), July 2005. Version 1.1.
- [11] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. Web services security policy language (WS-SecurityPolicy), December 2002. Version 1.0.

- [12] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
- [13] D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation, at <http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/>.
- [14] Foundstone. WSDigger, July 2005. A t [www.foundstone.com/resources/proddesc/wsdigger.htm](http://www.foundstone.com/resources/proddesc/wsdigger.htm).
- [15] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. At <http://www.martinowler.com/articles/languageWorkbench.html>.
- [16] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *Proceedings of the 2002 ACM workshop on XML Security*, pages 18–29. ACM Press, 2002.
- [17] J. Hogg, H. de Lahitte, D. Gonzalez, P. Cibraro, P. Coupland, M. Bhao, and P. Slater. *Microsoft WS-I Basic Security Profile 1.0 Sample Application*. Microsoft Corporation, June 2005. Preview release for the .NET Framework version 1.1.
- [18] E. Kleiner and A. W. Roscoe. Web services security: A preliminary study using Casper and FDR. In *Proceedings of Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.
- [19] Microsoft Corporation. *Web Services Enhancements (WSE) 2.0*, 2004. At <http://msdn.microsoft.com/webservices/building/wse/default.aspx>.
- [20] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, March 2004. OASIS Standard 200401, at <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>.
- [21] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [22] M. O’Neill. Mapping security to a services oriented architecture, March 2005. CASSIS’05 presentation, at <http://www-sop.inria.fr/everest/events/cassis05/Transp/oneill.ppt>.
- [23] J. Scambray and M. Shema. *Hacking Web Applications Exposed*. McGraw-Hill/Osborne, 2002.
- [24] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.
- [25] M. Tatsubori, T. Imamura, and Y. Nakamura. Best practice patterns and tool support for configuring secure web services messaging. In *International Conference on Web Services (ICWS’04)*, pages 244–251, 2004.
- [26] L. Tobarra, D. Cazorla, F. Cuartero, and G. Diaz. Application of formal methods to the analysis of web services security. In *2nd International Workshop on Web Services and Formal Methods (WS-FM 2005)*, pages 215–229, Sep 2005.
- [27] J. Udell. Threat modeling, 2004. At <http://weblog.infoworld.com/udell/2004/05/25.html>.
- [28] W3C. *SOAP Version 1.2*, 2003. W3C Recommendation, at <http://www.w3.org/TR/soap12>.
- [29] A. Wiesmann, M. Curphey, A. van der Stock, and R. Stirbei, editors. *A Guide to Building Secure Web Applications and Web Services*. OWASP, 2.0 Black Hat edition, 2005. At <http://www.owasp.org>.

## APPENDIX

### A. CONDITIONS REPORTED

1. The supplied file is not valid XML.
2. This configuration file does not have an associated policy file.
3. This configuration file specifies a policy file, but the specified file does not exist.
4. Test root certificates are allowed.
5. Replay detection is not enabled for a SecurityTokenManager of type `UsernameToken`.
6. Replay detection is enabled for a SecurityTokenManager that is not of type `UsernameToken`.
7. This policy file is being analyzed independently of any configuration file.
8. This policy file does not conform to the schema used by Policy Advisor.
9. This mapping has no policy for requests.
10. This mapping does not authenticate requests.
11. This mapping has no policy for responses.
12. This mapping specifies a response policy, but it cannot be found.
13. This mapping does not authenticate responses.
14. This mapping has no policy for faults (although it has a policy for responses).
15. This mapping does not authenticate faults (although it authenticates responses).
16. This mapping accepts requests with unauthenticated `<ReplyTo>` headers.
17. This mapping accepts requests without a `<ReplyTo>` header.
18. This mapping accepts requests with unauthenticated `<FaultTo>` headers.
19. This mapping accepts requests without a `<FaultTo>` header.
20. This mapping accepts responses with unauthenticated `<RelatesTo>` headers.
21. This mapping accepts responses without a `<RelatesTo>` header.
22. This mapping accepts faults with unauthenticated `<RelatesTo>` headers.
23. This mapping accepts faults without a `<RelatesTo>` header.
24. This mapping applies to a WS-Trust RST/RSTR handshake, which WSE treats specially.
25. This policy is not used.
26. This policy has no `<Integrity>` assertion.
27. This policy has no `<Confidentiality>` assertion.
28. This policy accepts messages with unauthenticated `<To>` or `<Action>` headers.
29. This policy accepts messages without a `<To>` or `<Action>` header.
30. This policy accepts messages with unauthenticated `<Timestamp>` or `<MessageID>` elements.
31. This policy accepts messages without a `<Timestamp>` or `<MessageID>` element.
32. This policy accepts messages with an unauthenticated SOAP Body.
33. This policy accepts certificates from any issuer.
34. This policy uses an unencrypted `<UsernameToken>` that leaks passwords in the clear.
35. This policy uses an unencrypted `<UsernameToken>` that leaks password digests.
36. This policy uses an unencrypted `<UsernameToken>` for signing messages.