

**Proceedings of the OWASP Europe
2006 Conference, Leuven, Belgium,
refereed papers track**

Frank Piessens (editor)

Report CW 448, May 30-31, 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

**Proceedings of the OWASP Europe
2006 Conference, Leuven, Belgium,
refereed papers track**

Frank Piessens (editor)

Report CW448, May 30-31, 2006

Department of Computer Science, K.U.Leuven

Abstract

The Open Web Application Security Project (OWASP) is dedicated to finding and fighting the causes of insecure software. The OWASP AppSec conferences bring together application security experts, researchers and practitioners from all over the world. Industry and academia can meet to discuss open problems and new solutions in application security. This report bundles the papers accepted for presentation at the refereed papers track of the OWASP Europe 2006 conference.

Preface

The Open Web Application Security Project (OWASP, <http://www.owasp.org>) is dedicated to finding and fighting the causes of insecure software. OWASP has dozens of projects and over 50 chapters worldwide focused on application security. Their high quality tools and documentation are used everywhere, including the freely available book-length "Guide to Secure Web Applications and Services", the leading web application penetration testing tool called "Web-Scarab", and an advanced web application security training application called "WebGoat". The OWASP Foundation, a not-for-profit charitable organization, ensures the ongoing availability and support for this work.

The OWASP AppSec conferences (<http://www.owasp.org/conferences.html>) bring together application security experts, researchers and practitioners from all over the world. Industry and academia can meet to discuss open problems and new solutions in application security. The conferences offer tutorials, keynotes, and invited presentations.

For the first time, the 2006 OWASP AppSec Europe conference has featured a refereed papers track. Original papers pertaining to all aspects of web application security were solicited. Each submitted paper was reviewed by three members of the programme committee. Four papers were accepted for presentation at the conference. This report bundles the four accepted papers.

Programme Committee:

Konstantin Beznosov, University of British Columbia, Canada
Sebastien Deleersnyder, Aszure and OWASP Belgian Chapter, Belgium
Andreas Fuchsberger, Royal Holloway, University of London, UK
André Mariën, Ubizen, Belgium
Mattia Monga, Milan University, Italy
Johan Peeters, secappdev.org, Belgium
Frank Piessens, Katholieke Universiteit Leuven, Belgium (chair)
Erik Poll, Radboud Universiteit Nijmegen, The Netherlands
Maarten Rits, SAP Research, France
Chris Vanden Berghe, IBM Research, Switzerland

RequestRodeo: Client Side Protection against Session Riding*

Martin Johns and Justus Winter

Security in Distributed Systems (SVS)
University of Hamburg, Dept of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
(johns|4winter)@informatik.uni-hamburg.de

Abstract. The term Session Riding denotes a class of attacks on web applications that exploit implicit authentication processes. There are four distinct methods of implicit authentication found in today’s web applications: Cookies, http authentication, IP address based access control and client side SSL authentication. As many web applications fail to protect their users against Session Riding attacks we introduce RequestRodeo, a client side solution to counter this threat. With the exception of client side SSL, RequestRodeo implements protection against the exploitation of implicit authentication mechanisms. This protection is achieved by removing authentication information from suspicious requests.

1 Introduction

Session Riding (also known as “Cross Site Request Forgery”) attacks are public at least since 2001 [16]. However this class of web application vulnerabilities is rather obscure compared to attack vectors like Cross Site Scripting or SQL Injection. Session Riding is neither included in the OWASP’s top 10 list of web application threats [15] nor in the Web Application Security Consortium’s threat classification [3]. As the trend towards web applications continues and an increasing number of local programs and appliances like firewalls rely on web based frontends, the attack surface for Session Riding grows continuously. Session Riding is an attack that targets the user rather than the web application. As long as web applications do not take measures to protect their users against this threat, it is important to investigate possibilities to implement client side mechanisms. In this paper we present RequestRodeo, which is, to the best of our knowledge, the first client-side solution for protection against Session Riding attacks.

* This work was supported by the German Ministry of Economics (BMWi) as part of the project “secologic”, www.secologic.org.

2 Technical Background

2.1 Implicit Authentication

With “implicit authentication” we denote processes which cause the web browser to automatically include authentication information in http requests. There are several widely supported methods of implicit authentication:

- **Http authentication:** Http authentication [8] enables the web server to request authentication credentials from the browser in order to restrict access to certain webpages. There are three methods frequently used: Basic, digest and NTLM (a proprietary extension by Microsoft [9]). In all these cases the initial authentication process undergoes the same basic steps (for brevity reasons only a simplified version of the process is given):
 1. The browser sends an http request for a URL for which authentication is required.
 2. The web server answers with the status code “401 Unauthorized” causing the web browser to demand the credentials from the user e.g. by prompting for username and password.
 3. After the user has supplied the demanded information, the web browser repeats the http request for the restricted resource. This request’s header contains the user’s credentials in encoded form via the **Authorization** field.
 4. The server validates whether the user is authorized. Depending on the outcome, the server either answers with the requested page or again with a 401 status code.

The browser remembers the credentials for a certain time. If the client requests further restricted resources that lie in the same authentication realm, the browser includes the credentials automatically in the request.

- **Cookies:** Web browser cookie technology [14] provides persistent data storage on the client side. A cookie is a data set consisting at least of the cookie’s name, value and domain. It is sent by the web server as part of an http response message using the **Set-Cookie** header field. The cookie’s domain value is used to determine in which http requests the cookie is included. Whenever the web browser accesses a webpage that lies in the domain of the cookie (the domain value of the cookie is a valid domain suffix of the page’s URL), the cookie is automatically included in the http request using the **Cookie** field. Cookies are often used as authentication tokens by today’s web applications. After a successful login procedure the server sends a cookie to the client. Every following http request that contains this cookie is automatically regarded as authenticated.
- **Client side SSL authentication:** The Secure Sockets Layer (SSL) and its successor the Transport Layer Security (TLS) [4] protocols enable cryptographically authenticated communication between the web browser and the web server. To authenticate the communication partners X.509 certificates and a digital signature scheme are used.

All these methods have in common, that after a successful initial authentication, the web browser includes the authentication tokens (either the cookie, the http authentication credentials or the SSL signature) automatically in further requests without user interaction.

IP address based authentication: A special case of implicit authentication is often found in intranets. Instead of actively requesting user authentication, the web application passively uses the request's source IP address as authentication token, only allowing certain IP (or MAC) addresses. Some intranet servers do not employ any authentication at all because they are positioned behind the company's firewall. In this case every web browser that is behind that firewall is authorized automatically.

2.2 Cross Site Request Forgery

“Session Riding” or “Cross Site Request Forgery” (CSRF) [18] is an attack technique that exploits implicit authentication. The attack is executed by causing the victim's web browsers to create hidden http requests to restricted resources. In the case that a successful request for such a resource causes the web application to commit further persistent actions (e.g. like altering database fields, sending email or changing the applications state), these actions are done using the victim's authentication tokens.

Example: a (rather careless) site for online banking provides a HTML form to place credit transfers. This form uses the GET method and has the action URL `http://bank.com/transfer.ext`. The form is only accessible by properly authenticated users, employing one of the techniques described above. If an attacker is able to trick a victim's browser to request the URL `http://bank.com/transfer.ext?amount=10&ano=007`, while the victim's browser maintains an authenticated state for the banking site, the owner of the account with the number 007 might gain €10. To execute the attack the attacker manufactures a harmless appearing webpage. In this webpage the attacker includes HTML or Javascript elements, that cause the victims web browser to request the malicious URL. This can be done for example with a hidden image: ``. If the attacker successfully lures the victim to visit the malicious website, the attack can succeed.

All access control methods described in Section 2.1 are vulnerable against CSRF as long as no countermeasures against this attack method have been implemented by the web application.

CSRF attacks are not necessarily limited to submitting a single fraudulent request. Workflows that require a series of http requests (i.e. web forms that span over more than one webpage) might be vulnerable as well, as long as certain conditions are fulfilled: The content and identifiers of every step of the workflow's web forms are known prior to the attack and the workflow does not employ a separate mechanism to track the workflow's progress (i.e. a request id) but uses the implicit communicated session identifier. If these conditions are satisfied an attacker can create in most cases a series of hidden iframes that host malicious

web forms. These forms are automatically submitted sequentially via JavaScript using the iframe's `onLoad`-events, thus simulating a user that is filling the forms in their proper order.

Cross Site Scripting (XSS) [5] and CSRF attacks are frequently confused as they are clearly related. Both attacks are aimed at the user and often require the victim to access a malicious webpage. Also the potential consequences of the two attack vectors can be similar: The attacker is able to submit certain actions to the vulnerable web application using the victim's identity. The causes of the two attack classes are different though. A web application that is vulnerable to XSS fails to properly sanitize user provided data before including this data on a webpage, thus allowing an attacker to include malicious JavaScript in the web application. This JavaScript consequently is executed by the victim's browser and initiates the malicious requests. XSS attacks have more capabilities beyond the creation of http request and are therefore more powerful than CSRF attacks: A rogue JavaScript has almost unlimited power over the webpage it is embedded in and is able to communicate with the attacker. A XSS can e.g. obtain and leak sensitive information.

3 Securing Web Applications against CSRF

This paper mainly focuses on client side protection against CSRF. However it is important to discuss the server side as well in order to understand why CSRF vulnerabilities are still an issue.

3.1 How NOT to do it

There exist misconceptions about possibilities to protect web applications against CSRF attacks:

- **Accepting only http POST requests:** A frequent assumption is, that a web application which only accepts form data from http POST request is protected against CSRF, as the popular attack method of using IMG tags only creates http GET requests. This is not true: Hidden POST requests can be created e.g. by using HTML forms in invisible iframes, which are automatically submitted via JavaScript.
- **Referrer checking:** An http request's referrer [7] indicates the URL of the webpage that contained the HTML link or form that was responsible for the request's creation. The referrer is communicated via an http header field. To protect against CSRF web application check if a request's referrer matches the web applications domain. If this is not the case, the request is usually rejected. Some users prohibit their web browsers to send referrer information because of privacy concerns. For this reason web applications have to accept requests, that do not carry referrer information. Otherwise they would exclude a certain percentage of potential users from their services. It is possible for an attacker to reliably create referrerless requests (see below). For this

reason any web application that accepts requests without referrers cannot rely on referrer checking as protection against CSRF.

In the course of preparing this paper, we conducted an investigation on the different possibilities to create http requests without referrers in a victim’s browser. We found three different methods to create hidden request that do not produce referrers. Depending on the web browser the victim uses, one or more of these methods are applicable by the attacker.

1. Page refresh via meta tag: This method employs the “HTTP-EQUIV = Refresh” meta tag. The tag specifies a URL and a timeout value. If such a tag is found in the “head” section of an HTML document, the browser loads the URL after the given time. Example:

```
<META HTTP-EQUIV=Refresh CONTENT="0; URL=http://path_to_victim">
```

On some web browsers the http GET request, which is generated to retrieve the specified URL, does not include a referrer. It is not possible to create POST request this way.

2. Dynamically filled frame: To generate hidden POST requests, the attacker can use an HTML form with proper default values and submit it automatically with JavaScript. To hide the form’s submission the form is created in an invisible frame. As long as the `src` attribute of the frame has not been assigned a value, the referring domain value stays empty. Therefore the form cannot be loaded as part of a predefined webpage. It has to be generated dynamically. The creation of the form elements is done via calls to the frames DOM tree [12].
3. Pop under window: The term “pop under” window denotes the method of opening a second browser window that immediately sends itself to the background. On sufficiently fast computers users often fail to notice the opening of such an unwanted window. Such a window can be used to host an HTML form that is submitted either automatically or by tricking the victim to click something. The form can be generated by calls to the DOM tree or by loading a prefabricated webpage. Depending on the victim’s browser one of these methods may not produce a referrer (see below for details).

To examine the effectiveness of the described methods, we tested them with common web browsers. See table 1 for the results of our investigation. The only web browser that was resistant to our attempts was Opera.

Method/Browser	IE 5	IE 6*	IE 7**	FF 1.07	FF 1.5	O 8	S 1.2
META Refresh				X	X		
Dynamic filled frame	X	X	X	X	X		X
Pop up window (regular)	X	X	X				
Pop up window (dynamically filled)				X	X		

IE: Internet Explorer; FF: Firefox; S: Safari; O: Opera; *: IE 6 XPSP 2; **: IE 7 (Beta 2)

Table 1. Generating referrerless requests (“X” denotes a working method)

3.2 How to do it

Using random form tokens: To prevent CSRF attacks, a web application has to make sure that incoming form data originated from a valid HTML form. “Valid” in this context denotes the fact that the submitted HTML form was generated by the actual web application in the first place. It also has to be ensured that the HTML form was generated especially for the submitting client. To enforce these requirements, hidden form elements with random values can be employed. These values are used as one time tokens: The triplet consisting of the form’s action URL, the ID of the client (e.g the session ID) and the random form token are stored by the web application. Whenever form data is submitted, the web application checks if this data contains a known form token which was stored for the submitting client. If no such token can be found, the form data has been generated by a foreign form and consequently the request will be denied. See [18] for a similar approach.

Using explicit authentication: There are methods to communicate authentication tokens explicitly: Authentication tokens can be included into the web application’s URLs or transported via hidden fields in HTML forms. These techniques are resistant to CSRF attacks.

3.3 Reasons for the Existence of CSRF Vulnerabilities

In today’s web applications CSRF problems can be found frequently. There are several reasons for this. Compared to vulnerability classes like Cross Site Scripting (XSS) [5] or SQL Injection, CSRF is rather obscure. While in many cases the consequences of CSRF attacks can be as severe as XSS exploits, web application developers are often unaware or dismissive when it comes to this vulnerability class. Furthermore, most web application frameworks lack a central mechanism for protection against CSRF, opposed to e.g. XSS for which numerous filtering functions are provided. In addition, automatic approaches like “taint checker” [11] for detecting SQL Injection and XSS problems do not exist for CSRF.

4 Client Side Protection against CSRF

We propose a client side solution to enable security conscious users to protect themselves against CSRF attacks. Our solution works as a local proxy on the user’s computer.

4.1 Concept

As described in Section 2.2 the fundamental mechanism that is responsible for CSRF attacks to be possible is the automatic inclusion of authentication data in any http request that matches the authentication data’s scope. Our solution is to partly disable the automatism that causes the sending of the authentication data.

The proxy identifies http requests which qualify as potential CSRF attacks and strips them from all possible authentication credentials. We chose to implement our solution in form of a proxy instead of integrating it directly into web browser technology because this approach enables CSRF protection for all common web browsers.

Identification of suspicious requests: The proxy resides between the client’s web browser and the web application’s server. Every http request and response is routed through the proxy. Because of the fact that the browser and the proxy are separate entities, the proxy is unable to identify how an http request was initiated. To decide if an http request is legitimate or suspicious of CSRF, we introduce a classification:

Definition 1 (entitled). *An http request is classified as **entitled** only if:*

- *It was initiated because of the interaction with a web page (i.e. clicking on a link, submitting a form or through JavaScript) and*
- *the URLs of the originating page and the requested page satisfy the “same-origin policy” [17]. This means that the protocol, port and domain of the two URLs have to match.*

Only requests that were identified to be entitled are permitted to carry implicit authentication information.

To determine if a request can be classified as *entitled*, the proxy intercepts every http response and augments the response’s HTML content. Every HTML form, link and other means of initiating http requests is extended with a random URL token. Furthermore the tuple consisting of the token and the response’s URL is stored by the proxy for future reference. From now on, this token allows the proxy to identify outgoing http requests with prior http responses. Every request is examined whether it contains a URL token. If such a token can be found, the proxy compares the token value to the values which have been used for augmenting prior http responses. This way the proxy is able to determine the URL of the originating HTML page. By comparing it with the request’s URL, the proxy can decide if the criteria defined in definition 1 are met. If this is not the case, all implicit authentication information is removed from the request.

Removal of authentication credentials: As discussed in Section 2.1 there are two different methods of implicit authentication used by today’s web applications that include credentials in the http header: Http authentication and cookies. If the proxy encounters an http request, that cannot be classified as *entitled*, the request is examined if its header contains `Cookie` or `Authorization` fields. If such header fields are found, the proxy triggers a reauthentication process. This is done either by removing the `Cookie` header field or by ignoring the `Authorization` field and requesting a reauthentication before passing the request on to the server. Following the triggered reauthentication process, all further requests will be *entitled* as they originated from a page that belongs to the web application (beginning with the webpage that executed the reauthentication).

Prevention of IP address based attacks: To protect resources, that filter access based on the request’s IP address, the proxy verifies that the destination URL of every request, which has not been classified as *entitled*, is reachable from the outside. To achieve this, the proxy has to employ an entity that resides outside the client’s intranet (see below for details). If the request’s destination URL is not “world reachable”, the proxy drops the request and replies with a confirmation dialog, to ensure that the request was intended by the user and not part of an CSRF attack.

Client side SSL authentication: Our solution is not yet able to prevent CSRF attacks that exploit client side SSL authentication.

4.2 Implementation

We implemented a proof of concept of our approach using the Python programming language with the Twisted [6] framework. Free Python interpreters exist for all major operating systems. Thus, using our solution should be possible in most scenarios. We call our implementation “RequestRodeo”. In the next paragraphs we discuss special issues that had to be addressed in order to enforce the solution outlined in Section 4.1.

Augmenting the response’s HTML content: The process of adding the random tokens to a webpage’s URLs is straight forward: The proxy intercepts the server’s http response and scans the HTML content for URLs. Every URL receives an additional GET parameter called `_rrt` (for “RequestRodeoToken”). Furthermore JavaScript code that may initiate http requests is altered: The proxy appends a JavaScript function called `addToken()` to the webpage’s script code. This function assumes that its parameter is a URL and adds the GET token to this URL. Example: The JavaScript code

```
document.location = someVariable;
```

is transformed to

```
document.location = addToken(someVariable);
```

This alteration of URLs that are processed by JavaScript is done dynamically because such URLs are often assembled on script execution and are therefore hard to identify reliably otherwise.

Removal of header located authentication credentials: The following aspects had to be taken into consideration:

- Cookies: If a `Cookie` header field is found in a suspicious request, it is deleted before the request is passed to the server. To ensure compatibility with common web applications, our solution somewhat relaxes the requirements of Definition 1: The proxy respects a cookie’s domain value. A cookie is therefore only discarded if its domain does not match the domain of the referring page. Otherwise e.g. a cookie that was set by `login.example.org` with the domain value “example.org” would be deleted from requests for `order.example.org`.
- Http authentication: Simply removing the authorization data from every request that has not been classified as *entitled* is not sufficient. The proxy

cannot distinguish between a request that was automatically supplied with an **Authorization** header and a request, that reacts to a 401 status code. As the web browser, after the user has entered his credentials, simply resends the http request that has triggered the 401 response, the resulting request is still not *entitled* because its URL has not changed. Therefore the proxy has to uniquely mark the request's URL before passing it on to the server. This way the proxy can identify single requests reliably. It is therefore able to determine if an **Authorization** header was sent because of a "401 Unauthorized" message or if it was included in the message automatically without user interaction.

Whenever the proxy receives a request, that was not classified as *entitled* and contains an **Authorization** header, the following steps are executed (see figure 1):

1. The proxy sends a "302 temporary moved" response message. As target URL of this response the proxy sets the original request's URL with an added unique token.
2. The client receives the "temporary moved" response and consequently requests the URL that was provided by the response.
3. The URL token enables the proxy to identify the request. The proxy ignores the **Authorization** header and immediately replies with a "401 Unauthorized" message, causing the client browser to prompt the user for username and password. Furthermore the proxy assigns the status *entitled* to the URL/token combination.
4. After receiving the authentication information from the user, the client resends the request with the freshly entered credentials.
5. As the request now has the status *entitled*, the proxy passes it on to the server.

An analog process has to be performed, whenever a not *entitled* request triggers a "401 Unauthorized" response from the server. The details are left out for brevity.

Usage of an outside entity to prevent IP based attacks: As described earlier, in intranet scenarios the possession of a local IP address is often considered to be sufficient authentication. For this reason, the proxy has to make sure that the target of a request, that was not classified as *entitled*, is accessible from any host on the internet. We introduce a *reflection service* that is installed on a host outside of the cooperate intranet. The URL of every questionable request is submitted to this service. The reflection service poses a HEAD request for the URL. If this request succeeds, it is save to assume that no IP based authentication mechanism is in place. The outcome of this test is communicated back to the proxy. If the reflection service was not able to access the URL, the proxy withholds the http request and replies with a confirmation dialog instead. Only if the user explicitly confirms his intend of requesting the protected content, the request is passed on to the server.

For performance reasons, the proxy keeps track of all IP addresses that were checked this way. Every IP address is therefore only checked once, as long as

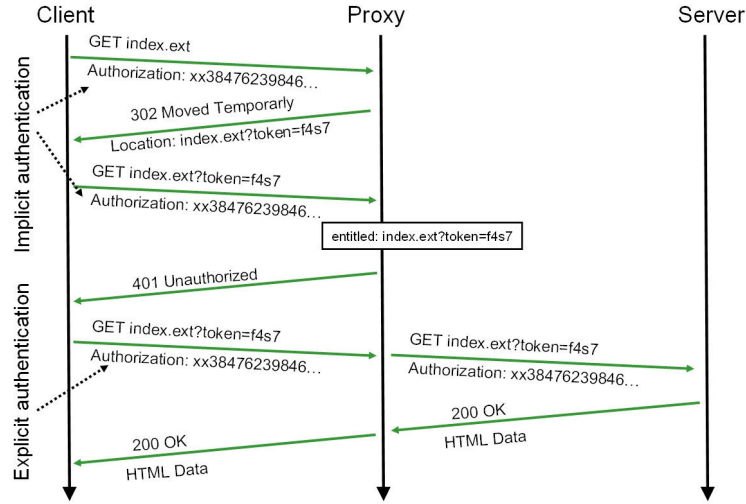


Fig. 1. Intercepting implicit http authentication

the user's IP address does not change. The reflection server will be distributed together with the proxy to enable privacy conscious users to set up their own service.

Notification: Whenever the proxy removes implicit authentication credentials, an unobtrusive notification element is added to the http response's HTML content in order to notify the user about the proxy's action. In our prototype this is done via a small floating sign.

5 Discussion

As described above our solution identifies http requests that pose potential CSRF attacks. For these requests the implicit authentication processes are disabled. With the exception of intercepting requests for intranet resources, the http request themselves are not prevented, only the authentication information is removed. For this reason our solution interferes as little as possible with the usage of web based applications. For example if a web application provides additionally to the restricted resources also public content, this public content can be referenced by outside webpages without the interference of the proxy. The requests for these public items may initially contain authentication credentials, which are subsequently removed by the proxy. But this removal does not influence the server's response, as no authentication was required in the first place.

With the single exception of local attacks (see below), the in Section 2.2 described CSRF attacks are prevented reliably, as all http requests originating from an attacker's website or from outside the web browser (e.g. from an email application) are identified as not being *entitled*

5.1 Limitations

Our solution cannot protect from “local” CSRF attacks. With local CSRF attacks we denote attacks that have their origin on the attacked web application. If e.g. an application allows its users to post images to one of the application’s webpages, a malicious user may be able to use the image’s URL to launch a CSRF attack. Our proxy would consider the image request as *entitled* as the image is referenced by a webpage that belongs to the application.

As mentioned above, our solution is furthermore not able to prevent CSRF attacks on client side SSL authentication. This issue could be solved, if the proxy instead of the browser would handle the client side SSL authentication.

Some webpages use JavaScript to create parts of the page’s HTML code locally. As Javascript is a highly dynamic language, our current implementation may fail in some cases to correctly classify all included URLs as *entitled*. For this reason, we are considering the implementation of strict referrer checking as a second line of defense in certain cases.

We designed our solution to interfere as little as possible with a user’s browsing. The most notably inconvenience that occurs by using the proxy is the absence of auto login: Some web applications allow the setting of a long lived authentication cookie. As long as such a cookie exists, the user is not required to authenticate. In almost every case, the first request for a web application’s resource is not *entitled*, as it is caused either by entering the URL manually, selecting a bookmark or via a web page that does not belong to the application’s domain. For this reason the proxy removes the authentication cookie from the request, thus preventing the automatic login process.

5.2 Future Work

As noted above, our solution does not yet protect against attacks on client side SSL authentication. An enhancement of our solution in this direction is therefore desirable.

Another future direction of our approach could be the integration of the protection directly into the web browser. This step would make the process of augmenting the HTML code unnecessary, as the web browser has internal means to decide if a request is *entitled*. Furthermore, such an integration would also enable protection against attacks on client side SSL authentication, as no interception of encrypted communication would be necessary. As noted above, we decided to implement our solution at first in form of a local web proxy to enable a broad usage with every available web browser.

5.3 Related Work

To date, little attention has been paid to CSRF attacks. The work conceptually closest to ours is NOXES [13], a client side proxy for protection against Cross Site Scripting attacks. NOXES prevents the communication of sensitive data (like session identifiers) to third parties by disallowing dynamically generated http

requests. Therefore, a malicious JavaScript is not able to leak information that was gathered on runtime, as only http requests that were statically embedded in the HTML code are allowed to pass the proxy.

Google’s “Safe Browsing Toolbar” [10] is an extension for the Firefox web browser. This extension intends to provide client side protection against “phishing” attacks. The protection is done by sending information about every visited webpage to a central entity that compares these information to a database of known phishing attacks. If the submitted information matches one of the stored signatures, the web browser displays a warning.

Furthermore, from a technological point of view, related approaches can be found in the domain of personal web firewalls like WebCleaner [1]. These firewalls are also implemented as a client side web proxy that intercepts all http communication.

6 Conclusion

In this paper we presented RequestRodeo, a client side solution against CSRF attacks. Our solution works as a local http proxy on the user’s computer. RequestRodeo identifies http requests that are suspicious to be CSRF attacks. This is done by marking all incoming and outgoing URLs. Only requests for which the origin and the target match, are allowed to carry authentication credentials that were added by automatic mechanisms. From suspicious requests all authentication information is removed, thus preventing the potential attack. Furthermore, special measures have been implemented to protect local resources in the client’s intranet. For this reason we introduced the concept of a reflection server, which assures that the requested resources are not protected by external mechanisms like firewalls, before allowing http requests that have a non local origin to access these resources. By implementing the described countermeasures RequestRodeo protects users of web applications reliably against almost all CSRF attack vectors that are currently known.

References

1. Webcleaner - a filtering http proxy. [application], <<http://webcleaner.sourceforge.net>>, (03/20/06).
2. Jesse Burns. Cross site reference forgery - an introduction to a common web application weakness . Whitepaper, <https://www.isecpartners.com/documents/XSRF_Paper.pdf>, 2005.
3. Web Application Security Consortium. Threat classification. whitepaper, <<http://www.webappsec.org/projects/threat/v1/WASC-TC-v1.0.pdf>>, 2004.
4. T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, <<http://www.ietf.org/rfc/rfc2246.txt>>, January 1999.
5. David Endler. The evolution of cross-site scripting attacks. Whitepaper, iDefense Inc., 20. May 2002.
6. Abe Fettig. *Twisted Network Programming Essentials*. O’Reilly, first edition edition, October 2005.

7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, <<http://www.w3.org/Protocols/rfc2616/rfc2616.html>>, June 1999.
8. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. RFC 2617, <<http://www.ietf.org/rfc/rfc2617.txt>>, June 1999.
9. Eric Glass. The ntlm authentication protocol. [online], <<http://davenport.sourceforge.net/ntlm.html>>, (03/13/06), 2003.
10. Google. Safe browsing for firefox. [application], <<http://www.google.com/tools/firefox/safebrowsing/>>, (03/20/06), 2006.
11. Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.
12. Philippe Le Hégaré, Ray Whitmer, and Lauren Wood. Document object model (dom). W3C recommendation, <<http://www.w3.org/DOM/>>, January 2005.
13. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks, security. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.
14. D. Kristol and L. Montulli. Http state management mechanism. RFC 2965, <<http://www.ietf.org/rfc/rfc2965.txt>>, October 2000.
15. OWASP. Top ten most critical web application security vulnerabilities. whitepaper, <<http://www.owasp.org/documentation/topten.html>>, January 2004.
16. John Percival. Cross-site request forgeries. [online], <<http://www.tux.org/peterw/csrf.txt>> (03/09/06), June 2001.
17. Jesse Ruderman. The same origin policy. [online], <<http://www.mozilla.org/projects/security/components/same-origin.html>> (01/10/06), August 2001.
18. Thomas Schreiber. Session riding - a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH, <http://www.securenet.de/papers/Session_Riding.pdf>, December 2004.

An Inline Approach for Secure SOAP Requests and Early Validation¹

Mohammad Ashiqur Rahaman, Maarten Rits and Andreas Schaad
SAP Research
805, Avenue du Docteur Maurice Donat
Font de l'Orme, 06250 Mougins
+33 (0)4 92 28 62 00
{mohammad.ashiqur.rahaman,maarten.rits,andreas.schaad}@sap.com

Abstract. Regarding the current status of message level security in Web Services, various standards like WS-Security along with WS-Policy play a central role. Although such standards are suitable for ensuring end-to-end message level security as opposed to point-to-point security, certain attacks such as XML rewriting may still occur. In addition the generation and validation of the key security mechanisms (e.g. signature) are always processor intensive tasks. Based on some real world scenarios we propose a scheme to include SOAP Structure information in outgoing SOAP messages and validate this information before policy driven validation in the receiving end. This allows us to detect some XML rewriting attacks early in the validation process, with an improved performance. We report on this efficient technique and provide a performance evaluation. We also provide insights into the WS-Security, WS-Policy and related standards' features and weaknesses.

Keywords: Web Services, WS-Security, XML rewriting attack.

1 Introduction

Web service specifications (WS*) have been designed with the aim of being composable to provide a rich set of tools for secure, reliable, and/or transacted web services. Due to the flexibility of SOAP-level security [1] mechanisms, web services may be vulnerable to a distinct class of attacks based on the malicious interception, manipulation, and transmission of SOAP messages, which are referred to as XML rewriting attacks [2]. Although WS-Security, WS-Policy and other related standards theoretically can prevent XML rewriting attacks in practice, incorrect use of these standards may make web services vulnerable to XML rewriting attacks.

All WS* security related specifications, however, introduce new headers in SOAP messages. So concerns about the operational performance of Web services security are legitimate because added XML security elements not only make use of more network bandwidth but also demand additional CPU cycles at both the sender side and at the receiver side. Therefore it is desirable to examine the performance issue of Web services security.

The main achievements of this paper are that we explore XML rewriting attacks [2] against web services. We propose measures detecting these attacks build on the idea of adding additional SOAP structure information. We further evaluate the performance of the proposed solution against the existing state of the art.

The paper is organized as follows. Section 2 discusses related work. Section 3 reviews the state of the art and its limitations. Section 4 illustrates attacker and rewriting attack sce-

¹This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

narios. Section 5 presents our efficient solution to prevent these attacks. In Section 6 we describe a case study in which we applied our approach. Section 7 evaluates the performance of the approach.

2 Related Work

Security protocols, described using web service specifications (WS*), are getting more complex day by day. Researchers are applying formal methods to verify and secure the protocols' and specifications' goals. As new vulnerabilities are exposed, these specifications continue to evolve. Microsoft's SAMOA [3] project is among the pioneer efforts where web services specifications are analyzed using rigorous formal techniques.

One of the focus areas of the SAMOA project is to identify common security vulnerabilities during security reviews of web services with policy-driven security [2]. The authors 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. While their previous work [4] is successful to generate and analyze web services security policies to be aware for vulnerabilities to XML rewriting attacks, this tool is able to bridge the gap between formal analyses and implementation quite efficiently.

The mentioned previous work [4] describes a formal semantics for WS-SecurityPolicy, and proposes an abstract link language [6] for specifying the security goals of web services and their clients. They present the architecture and implementation of fully automatic tools to compile policy files from link specifications, and to verify by invoking a theorem prover [7] whether a set of policy files run by any number of senders and receivers correctly implements the goals of a link specification, in spite of active attackers. Note that policy-driven web services implementations are prone to the usual subtle vulnerabilities associated with cryptographic protocols; these tools help prevent such vulnerabilities, as policies can be verified when first compiled from link specifications, and also can be re-verified against their original goals after any modifications during deployment.

The assumptions with all these formalizations are that the attacker can compose messages, replay them, or decipher them only if it knows the right key, but cannot otherwise decrypt the encrypted messages. A severe limitation is that these formalizations do not model insider attacks: principals with shared keys are assumed well-behaved.

Some guidelines about performance evaluation of web services are specified in [8]. They address about the performance overheads of web services: the XML size, the calculation of the message size, the choice of the XML parser, the costs of the serialization and deserialization, the costs of connection establishment and the overheads at the network level.

The performance of web service security in terms of its relationship in both the implemented cryptography in Java and the properties of input documents like the size and the complexity of structure are investigated in [9]. Different cryptography algorithms and various kinds of key references in XML are compared as well. They also categorize the web service security activities consisting of at least cryptography operations and XML processing. Cryptography operations are byte based, structure neutral and computation intensive. They conclude that XML processing, in particular XML canonicalization (page 2, Section 3.5) is the area where WS security performance should focus on.

In our later evaluation we take such performance issues into account as well.

3 State of the Art

In this section we describe different web services standards' insights and limitations related to web services security that are deployed widely in web services technologies.

3.1 WS-Security

The WS-Security [1] specification defines an end to end security framework that provides support for intermediary security processing. Message integrity is provided by using XML Signature [10] in conjunction with security tokens to ensure that messages are transmitted without modifications. Message confidentiality is granted by using XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. WS-Security seeks to encapsulate the security interactions described above within a set of security Headers.

3.2 WS-Policy

WS-Policy is essentially a logical predicate on SOAP messages over base assertions, determining which message parts must be present, signed, or encrypted. A standard policy framework would make it possible for developers to express the policies of services in a machine-readable way. Web services infrastructure could be enhanced to understand certain policies and enforce them at runtime. The policy framework currently expressed by WS-Policy [11] requires the definition of policy “Assertions” (predicates) for each domain to which policy is to be applied. Three examples of specifications defining such Assertions have been published to date:

- *WS-PolicyAssertions* [12], defining some general-purpose Assertions,
- *WS-SecurityPolicy* [5], defining policy Assertions for WS-Security and other specifications that might cover the same message security space, and
- *WS-ReliabilityPolicy* [14], defining policy Assertions for WS-Reliability [4] and other specifications that might cover the same reliable messaging space.

3.3 WS-SecurityPolicy

WS-SecurityPolicy1.0 [5], built on the WS-Policy [11] and WSPolicyAssertion [12], is a declarative XML format for programming how web services implementations construct and check WS-Security headers. It expresses policy in terms of individual headers on individual messages. It defines two base assertions for integrity and confidentiality. The more recent version WS-SecurityPolicy1.1 [15] expresses policy in terms of higher-level message patterns.

3.4 XML-Signature

XML Digital Signature specification [10] specifies how to describe, attach and validate a digital signature using XML. The structure of a digital signature as currently defined within the specification is shown in Fig 1.

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Fig. 1. Structure of XML Digital Signature

The Signature element is the primary construct of the XML digital signature specification. The signature is generated from a hash over the canonical form of the manifest, which can reference multiple XML documents. Canonicalization means to put a structure in a

standard format that is generally used. The <SignedInfo> element is the manifest that is actually signed. This data is sequentially processed through several steps on the way to becoming signed. A concrete example using XML Signature is given in Fig 2.

3.5 XML Canonicalization

XML canonical form states that XML documents though being logically equivalent within an application context may differ in physical representations based on XML permissible syntactic changes. These changes, for example, can be attribute ordering, entity references or character encoding. Basically this is a process of applying standard algorithms to generate a physical representation of an XML document. In XML security, there is a standard mechanism to produce an identical input to the digestion procedure prior to both signature generation and signature verification. Given its necessity, the speed of canonicalization will have an impact on the overall performance of SOAP security.

3.6 Limitations of the State of the Art

Considering the state of the art, various web service specifications [1],[10],[15],[17] and articles [22], we observe the following limitations to their applicability in secure web services:

1. It is not realistic to capture all security needs within a simple declarative syntax (e.g. WS-Policy, WS-SecurityPolicy).
2. In order to handle each Assertion, a policy processor must incorporate a domain-specific code module that understands the interpretation of that Assertion as defined in the domain-specific specification.
3. The interpretation is subject to human error, so without strict conformance tests, different implementations of the processor for each assertion may not be consistent.
4. Policy files need to be validated on application startup because if a policy file is compromised then malicious SOAP messages could be transported.
5. Enforcing the policy is totally domain dependent and it is a must. The strongest policy may be useless if it is not applied to the right message.
6. To enforce the policy for intermediaries is yet to be standardized.
7. Lack of standardization to retrieve policy for sender or receiver.
8. The digital signature references message parts by their Id attributes but says nothing of their location in the message. So an attacker can rewrite the message part placing it in a new header keeping the reference valid.
9. The message identifier is optional according to WS-Addressing [16] but must be present in a request if a reply is expected.

All these limitations may directly affect the security and the performance of the web services.

4 ATTACKER SCENARIOS

The presence of a hostile opponent who can observe all the network traffic and is able to send any fraudulent messages meeting the protocol requirements must always be assumed. So SOAP messages are vulnerable to a distinct class of attacks by an attacker placed in between any two legitimate SOAP nodes (Sender, Intermediaries, and Ultimate Receiver). The attacker intercepts the message, manipulates it and may transmit it. These kinds of attacks are called XML rewriting attacks [2].

4.1 Attack Patterns

We observe that XML rewriting attacks follow two patterns in general. The patterns give us indication about the security loophole. The general patterns are as follows:

1. SOAP Extensibility Exploitation: The attacker generates new SOAP elements and adds those into the message, keeping it well formed. Consequently malicious data may be transported.

2. Copy & Paste: The attacker copies parts of a message into other parts of that message, or into completely new messages which may be generated using the previous pattern.

The attacker is able to forge message parts and tricks the recipient in a way that it is impossible to detect any tampering if the standards (e.g. WS-Security, WS-Policy, WS-SecurityPolicy) are not used carefully. The important observation here is that not the cryptographic technique used as part of the standard has been broken but that the SOAP message structure has been exploited by an attacker. From a general perspective XML rewriting attacks exploit a known weakness of XML signatures that is described in section 3.6, item 8.

4.2 XML Rewriting Attacks

In this section, we see two concrete examples and show two errors that lead to typical XML rewriting attacks.

First Scenario: Consider, one service consumer of a Stock Quote service requests for some particular Stock (Fig 2). Each request causes the consumer to pay. We assume that one SOAP node (Ultimate receiver) is supposed to process the SOAP header or Body. The **<Security>** header block without a specified actor can be consumed by anyone, but must not be removed prior to the final destination. An attacker can now observe and manipulate the message on the SOAP path. He can move an element (e.g. **Message ID**) into a new, false header (e.g. **Bogus**) (Fig 3); everything else, including the certificate and signature, remains same. The **<Bogus>** element and its contents are ignored by the recipient since this header is unknown, but the signature is still acceptable because the element at reference URI **"Id-1"** remains in the message and still has the same value. This may cause the consumer to pay several times for the same request and forces the service to do redundant work.

Second Scenario: A customer submits an order that contains an **orderID** header (Fig 4) through his mobile device. He signs the **orderID** header and the body of the request (the contents of the order). When this is received by the order processing sub-system (an intermediary), it may insert **shippingID** into the header. The order sub-system would then sign, at a minimum, the **orderID** and the **shippingID**, and possibly the body as well. Then when this order is processed and shipped by the shipping department, a **shippedInfo** header might be appended. The shipping department would sign, at a minimum, the **shippedInfo** and the **shippingID** and possibly the body and forward the message to the billing department for processing. The billing department can verify the signatures and determine a valid chain of trust for the order, as well as who did what. An attacker with access to any SOAP node can copy & paste the **orderID** in a bogus header which causes the Order Processing System to process the same request several times. The attacker can copy & paste the body of the message under a new fake header and may insert arbitrary order information to be processed subsequently.

4.3 State of the Art Approach Against Attack

Methodical usage of WS-Policy [11], WS-PolicyAssertion [12], and WS-SecurityPolicy [13] resists these attacks. Microsoft has performed some experiments on its Web Services Enhancement [21] tool using another tool called WSE Policy Advisor [2]. The later tool uses three assertions: the message predicate assertion from WS-PolicyAssertion, and the integrity and confidentiality assertions from WS-SecurityPolicy.

- A **<MessagePredicate>** assertion lists the message parts that are mandatory.

- An <Integrity> assertion lists the message parts to be jointly signed. The listed message parts must be signed if present.
- A <Confidentiality> assertion lists the message parts that must be encrypted if present in the envelope.

The flawed policy that is used to generate and check the request messages for Fig 2 is shown in Fig 5. In the Fig 5, 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 allows the attacker to introduce a new header enabling the replay attack.

The policy file in Fig 6 could be used in the second scenario and it does not include <To>/<Action> header in the <MessagePredicate> which enables attacker to route the message to another shop.

To check these policy errors WSE Policy Advisor includes 36 static queries [2]. Those queries enable us to detect the flaw. In this case the query is: “The Header (MessageId, Header (To) and Header (Action) are included in a message predicate assertion whenever the same header is included in an integrity assertion”. Note that after writing the policy the client and the service need to enforce it on all request or response messages. The enforcement is totally domain dependent. WSE does this using some mappings which are included in policy file.

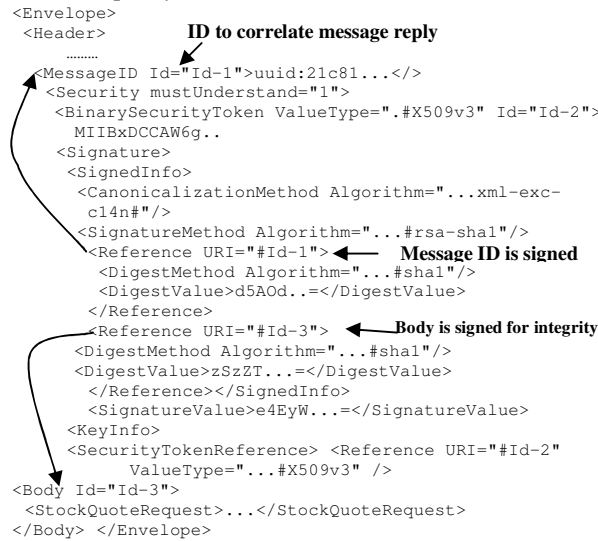


Fig. 2. SOAP message sent by the requester (Excerpt)

5 INLINE APPROACH

Considering the specifications of WS-Security, WS-Policy, WS-SecurePolicy and above discussion, we observe that a large number of SOAP extensions is possible. SOAP header (<Security>) information never considers the SOAP message structure which is essentially the major attack point. In the context of our paper we recognize the dynamic structure of SOAP messages by referring to them using the term SOAP Account. We show that including SOAP Account information in SOAP we can detect these XML rewriting attacks early in the validation process. This SOAP structure (SOAP Account) information can be inte-

grated easily while adding the headers. We need an inline approach to incorporate SOAP Account information into the message. The objectives of the proposed technique are:

1. To be able to pass SOAP Account information about the exchanged SOAP messages in a domain independent fashion.
2. SOAP messages should be protected while they are processed by a node such that they can be updated legitimately by the intermediaries if they are required to do so. Any tampering with pre-existing message parts by the compromising node must be detected early by the recipient before committing its resources to validate and to process the request.

```

<Envelope>
  <Header>..... Bogus header has been added by the atta-
  <Bogus>
  <MessageID Id="Id-1">uuid:21c81...</> </Bogus>
  <Security mustUnderstand="1">
    <BinarySecurityToken ValueType="...#X509v3"
      Id="Id-2">MIIBxDCCA...
    <Signature>
      <SignedInfo>
        <CanonicalizationMethod Algorithm="...xml-exc-
          c14n#" />
        <SignatureMethod Algorithm="...#rsa-sha1" />
        <Reference URI="#Id-1">
          <DigestMethod Algorithm="...#sha1" />
          <DigestValue>d5A0d...=</DigestValue>
        </Reference>
        <Reference URI="#Id-3">
          <DigestMethod Algorithm="...#sha1" />
          <DigestValue>zSzZT...=</DigestValue>
        </Reference></SignedInfo>
        <SignatureValue>e4EyW...=</SignatureValue>
      <KeyInfo>
        <SecurityTokenReference> <Reference URI="#Id-2"
          ValueType="...#X509v3" />
      </KeyInfo>
    </BinarySecurityToken>
  </Security>
  <Body Id="Id-3">
    <StockQuoteRequest>...</StockQuoteRequest>
  </Body></Envelope>

```

Fig. 3. SOAP message after attack (Excerpt)

The general objective is to protect the integrity features of a SOAP message while in transit from malicious attackers. While securing the requester and the service using WS-policy is well understood, securing the intermediaries using the same is not. A SOAP message may choose its next hop dynamically and its' required message parts may be secured dynamically based on the requirements of the intermediaries. We assume that the sender and the ultimate recipient of the message are always trusted.

5.1 Motivation of Using SOAP Account Information

After carefully observing the rewriting attacks, the following conclusions are obvious:

- All attacks are some kind of modification of a SOAP message (either deleting some parts and adding afterwards, or adding some completely new element in a SOAP message essentially in the Header portion or in Body).
- When some unexpected modification occurs in the form of manipulation of underlying XML elements, the intended predecessor or successor relationship of the SOAP element is lost consequently.
- The number of predecessor, successor, and sibling elements of a SOAP element where the unexpected modification occurs is changed and thus the expected hierarchy of the element is modified as well.

At the time of sending SOAP message, we can always keep an account of the SOAP elements by including SOAP Account into the message: (Fig 7)

- Number of child elements of root (Envelope).
- Number of header elements.
- Number of references for signing element.
- Predecessor, Successor, and sibling relationship of the signed object.

These SOAP Account information are computed while we are creating the message itself in the sending application. We do not incur any considerable overheads for the computation. Since this information is computed while creating the same message we call it the *inline approach* in this paper.

5.2 Proposed Technique

On the sender side, the protocol stack generates SOAP envelopes that satisfy its policy and then we add SOAP Structure/Account information (Fig 7) into the outgoing SOAP message. The sender must sign the SOAP Account information.

Conversely, on the receiver side, a SOAP envelope is accepted as valid, and passed to the application; if its SOAP Account/Structure Information is validated at first and then policy is satisfied for this envelope. Validating SOAP Account information before validating policy, we can detect rewriting attacks in the first phase and thus without doing processing intensive policy driven validation which may not detect attacks at all if not used carefully (Fig 5, Fig 6).

5.3 Using SOAP Account Information in SOAP Message

Fig 7 depicts the SOAP Structure/Account Information that we consider in our implementation of Java SOAP Account module. We capture the SOAP structure that is computable using any SOAP processor. Using this information we are able to detect the attacks in the scenarios. We have left one extension element to include any future extension. In later sections we give a concrete example how can we add and validate SOAP Account information.

Before sending any SOAP message we calculate the SOAP Account information and capture it in SOAP Header Elements. We add this information in a SOAP element either in Header or in Body and then sign it (Preferably in Header). The same arguments are applicable for intermediaries as well. This calculation is done by the `AddSoapAccount` module in our implementation. To understand it rigorously we take the second scenario from Fig 4 and use the following notation:

Encryption of a plaintext m into a cipher text is written as $C = \{m\}_K$, where K is the key being used.

A *digital signature* written as encryption $\{m\}_{S^{-1}}$, with a private signing key S^{-1} . When A sends some message m to B we write $A \rightarrow B : m$. We write $A \rightarrow B : \{m\}_{S^{-1}}$ when m is sent with a signature. Concatenation of m_1 and m_2 is written as $m_1 + m_2$. We define a *signed object pattern (SOP)* which manifests the signed elements in a message one intends to sign.

In Fig 8, A 's *signed object pattern (SOP_A)* is {SOAP Account + OrderID + Body},

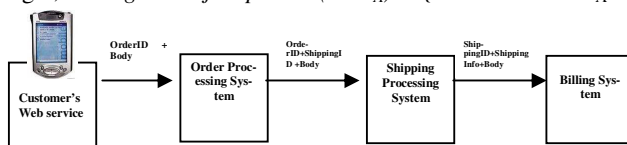


Fig. 4. SOAP processing with multiple intermediaries

where SOAP Account_A refers to the SOAP Account of A. A signs its *signed object pattern* before sending it to B:

$A \rightarrow B : R, \{ \text{SOAP Account}_A + \text{OrderID} + \text{Body} \}_A^{-1}$; Here R is the rest part of the message. B processes the order and adds new header ShippingID and B's *signed object pattern* (SOP_B) is $\{ I_1 + \text{SOAP Account}_B + \text{ShippingID} + \text{Body} \}$, where SOAP Account_B refers to the SOAP Account of B.

```
<Policy Id="FlawedPolicy1">
  <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>
```

Fig. 5. FlawedPolicy1 (Excerpt)

```
<Policy Id="FlawedPolicy2">
  <MessagePredicate>
    Body () Header (MessageID)
  </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>
```

Fig. 6. FlawedPolicy2 (Excerpt)

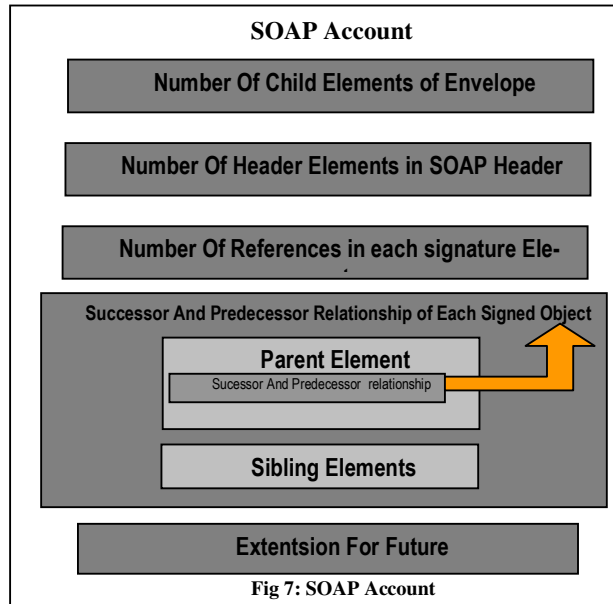


Fig 7: SOAP Account

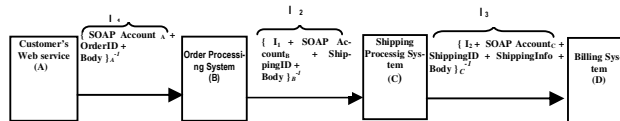


Fig 8: SOAP Processing with multiple intermediaries

$B \rightarrow C : R, \{ I_1 + \text{SOAP Account}_B + \text{ShippingID} + \text{Body} \}_B^{-1}$; Here I_1 is the signature of A's signed object pattern. C processes the order and adds new header ShippingInfo and C's signed object pattern(SOP_C) is $\{ I_2 + \text{SOAP Account}_C + \text{ShippingID} + \text{ShippingInfo} + \text{Body} \}$, where SOAP Account_C refers to the SOAP Account of C.

$C \rightarrow D : R, \{ I_2 + \text{SOAP Account}_C + \text{ShippingID} + \text{ShippingInfo} + \text{Body} \}_C^{-1}$; Here I_2 is the signature of B's signed object pattern.

Finally, D receives: $R, \{ \{ \{ SOP_A \}_A^{-1} + \{ SOP_B \} \}_B^{-1} + \{ SOP_C \} \}_C^{-1}$. D will validate SOAP Account_C using `CheckSoapAccount` module in our implementation. D uses SOAP Account_C as C is the outermost signature, it needs to validate. Having a nested signature, D can validate each signature subsequently using each public certificate respectively. Note that all SOAP Account information is also well protected by a signature which makes it impossible to change by any malicious host. Now the virtue of SOAP Account will be manifested directly. If any kind of XML rewriting attacks appears in the message in the form of the mentioned scenarios, it will be caught immediately by `CheckSoapAccount`. This is straightforward as each attack in a received SOAP message essentially invalidates the SOAP Account information that is bundled in the same SOAP message.

A key advantage of our approach compared to the policy-driven approach is that the deletion of headers and elements can be detected without restricting the flexible XML format. Deletion is a stronger form of a rewriting attack. In order to prevent this with the policy approach, every header and element should be declared as mandatory, which introduces first of all a performance penalty in the validation phase and more important it reduces strongly the flexibility of the XML message format. Only message elements that were defined in advance by the partners can be added. With the inline approach the different intermediaries still have some flexibility to add additional information, which can be detected for deletion/rewriting by the subsequent parties.

5.4 Where is the Efficiency?

One can argue about achieving efficiency in detecting XML rewriting attacks, considering the added modules (e.g. `AddSoapAccount`, `CheckSoapAccount`) at both ends. In Fig 8 we assume every SOAP node (A,B,C,D) has WS-security infrastructure implementation at least for signature generation (maybe for WS-Policy as well) and is supposed to process some headers. Every intermediary receives the message and it checks it in the `CheckSoapAccount` module before committing its resources for WS-Security and WS-Policy infrastructure. We show how we can detect the attacks in the added module whereas in the current approach the attacks may be undetected all the way to the ultimate end and even might be undetected in the end as well, unless it has a well designed policy. We show a concrete example describing this fact in the following section. We do a performance evaluation supporting this claim in section 7.

Note that the most popular XML packages comply with W3C Document Object Model (DOM), which provides a set of interfaces for creating, accessing and modifying both the structure and the content of the document. Security related XML processing includes parsing, validation, transformation and document tree traversal. As we mentioned in section 2 cryptographic processing (e.g. signing and verification, encryption and decryption, and among different algorithms) incurs negligible computing time where some researchers find that XML canonicalization is disproportionately time consuming. We can consciously ignore this XML canonicalization while validating SOAP Account information.

6 Case Study

6.1 Concrete Example

We consider a scenario where only one SOAP Account is attached with the requester's SOAP message and no intermediaries are supposed to update it. A customer, Alice, requests 1000 euros to be transferred from her account to the supplier (Bob's) account (Fig 9). Some malicious attacker intercepts this message and updates it stating to transfer 5000 euros instead of 1000 euros (Fig 10).

Fig 9 depicts the outgoing message after adding SOAP Account information. The policy file for this message would be Fig 11.

Observe that in spite of using `Body()` in `<MessageParts>` and `<MessagePredicate>` in Fig 11 the attack in Fig 10 is possible. This is due to the fact that `MessagePredicate` only makes statements about mandatory parts of the message and the XML signature does not say anything about the location of the message parts to be signed as stated in section 3.6.

6.2 Adding SOAP Account Information into SOAP Message

Using the `AddSOAPAccount` module we can calculate any accounting information about the outgoing message in Fig 9. The SOAP Account of Fig 9 is as follows:

- Number of children of Envelope is 2
- Number of Header is 2
- Number of Signed Elements is 3
- Immediate Predecessor of the 1st Signed Element is "Envelope"
- Sibling Elements of the 1st Signed Element is "Header"

The Extension element of the SOAP Account (Fig 7) makes it easy to add any additional common required accounting information between sender and receiver. There should be an agreement about the SOAP Account information they require beforehand. This information is added into a header named "SoapAccount". Before sending the message, SOAP Account must be signed by the sender.

Generation of Soap Account information neither depends on any enforcement infrastructure nor does it incur considerable execution time. It is rather efficient in terms of execution time as a SOAP Account can be computed inline while generating the SOAP message. We can easily attach this information using existing SOAP message libraries which makes it robust.

6.3 Simulating the Attacker

To simulate the attacker in these scenarios we design a Java class `Attacker` which represents the malicious host. After receiving the message from the legitimate sender it updates the message following the attack patterns described in section 4.1 and sends the updated message to the next hop. But this attempt to attack is detected by the legitimate receiver of the message.

6.4 SOAP Account Validation

The Soap Account validation should be done as soon as the message is received, before doing any policy validation. The receiver calculates the SOAP Account information of the received SOAP message (Fig 10) using `CheckSoapAccount` module as follows:

- Number of children of Envelope is 2
- Number of Header is 3.
- Number of Signed Elements is 3

- Immediate Predecessor of the 1st Signed Element is “BogusHeader”
 - Sibling Elements of the 1st Signed Element is “SoapAccount”, ”Security”
- On the other hand the obtained SOAP Account information as provided in the received SOAP message, (Fig 10) is as follows:
- Number of children of Envelope is 2
 - Number of Header is 2.
 - Number of Signed Elements is 3
 - Immediate Predecessor of the 1st Signed Element is “Envelope”
 - Sibling Elements of the 1st Signed Element is “Header”

```

<Envelope>
<Header>
  <Security>
    <UsernameToken Id=3>
      <Username>Alice</>
      <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
      <Created>2003-02-04T16:49:45Z</>
    <Signature>
      <SignedInfo>
        <Reference URI= #1>
          <DigestValue>Ego0...</>
        <Reference URI= #2>
          <DigestValue>Qser99...</>
        <Reference URI= #3>
          <DigestValue>OUytt0...</>
        <SignatureValue>
          vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI=#3/>
          <SoapAccount id=2>
            <NoChildOfEnvelope>2</>
          <NoOfHeader>
            </SoapAccount>
          </Body Id=1>
        <TransferFunds>
          <beneficiary>Bob</>
          <amount>1000</>
        </Body>
      </Envelope>
    </Header>
  </Envelope>

```

Message to bank's web service says: "Transfer 1000 euro to Bob, signed Alice"

Verifying signature using key derived from Alice's secret password

Fig. 9. A SOAP request before an attack (Excerpt)

```

<Envelope>
<Header>
  <Security>
    <UsernameToken Id=3>
      <Username>Alice</>
      <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
      <Created>2003-02-04T16:49:45Z</>
    <Signature>
      <SignedInfo>
        <Reference URI= #1>
          <DigestValue>Ego0...</>
        <Reference URI= #2>
          <DigestValue>Qser99...</>
        <Reference URI= #3>
          <DigestValue>OUytt0...</>
        <SignatureValue>
          vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI=#3/>
          <SoapAccount id=2>
            <NoChildOfEnvelope>2</>
          <NoOfHeader>
            </SoapAccount>
          <BogusHeader>
            <Body Id=1>
              <TransferFunds>
                <beneficiary>Bob</>
                <amount>1000</>
              </Body>
            </BogusHeader>
          </Body>
        <TransferFunds>
          <beneficiary>Bob</>
          <amount>5000</>
        </Body>
      </Envelope>
    </Header>
  </Envelope>

```

Attacker has intercepted the message

This reference is not valid anymore because No of header is not 2. After attack it is 3

Attacker has added a BogusHeader & included the Body

Amount has been changed to 5000 by the attacker

Fig. 10. SOAP request after an attempt to attack (Excerpt)

```

<Policy Id="FlawedPolicy3">
  <MessagePredicate>
    Body() Header(To) Header(Action)
  </MessagePredicate>
  <Integrity>
    <TokenInfo>
      <SecurityToken>
        <TokenType>...
        #UserNameToken
      </TokenType>
    </SecurityToken>
  </TokenInfo>
  <MessageParts>
    Body() Header(To) Header(Action)
    UsernameToken() Header(SoapAccount)
  </MessageParts>
</Integrity>
</Policy>

```

Body is included in MessagePredicate & Integrity assertions

Fig. 11. Flawedpolicy3 (Excerpt)

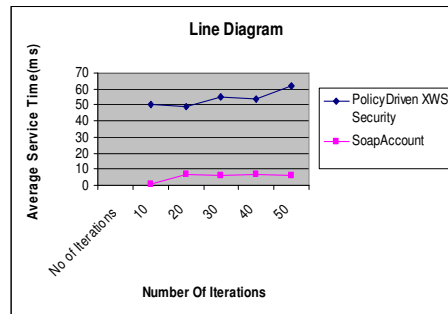


Fig. 12. Performance Diagram

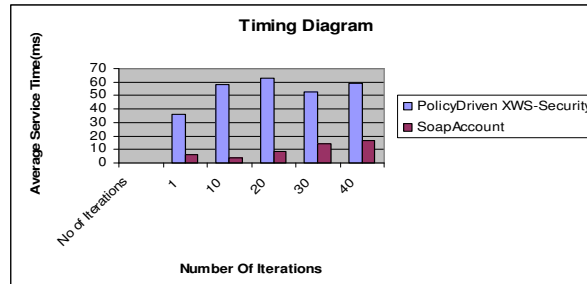


Fig. 13. Performance Diagram

If any mismatch happens the receiver can conclude that the SOAP message is not acceptable. In our proposed scenario, there is a clear mismatch. In addition, if an attacker changes the SOAP Account information meeting its updated SOAP message’s account information, then this message will be invalidated in the receiving end while validating the signature of the signed SOAP Account by the initial sender. Again, no substantial execution time is required here as we can validate the SOAP Account information inline while reading the message. The performance evaluation in the next section describes this more in detail.

7 PERFORMANCE EVALUATION

Performance evaluation of Web services can help implementers understand the behavior of the services and gives an indication to the feasibility of the deployment. The most commonly used approach to obtain performance results of a given Web service is performance testing, which means to run tests to determine the performance of the service under specific application and workload conditions. To be more specific the total execution time of a process is a measure of its efficiency.

We use XWS-Security framework as a comparable message level security infrastructure that has already wide deployments. XWS-Security framework has its own way of enforcing policy. XWS Configuration file is a domain dependent way of enforcing policy in XWS-Security Framework [13] in Java. This is essentially a XML file. Instead of using Policy directly this framework uses this XML file which has its own syntax and semantics for attaching and using the security features (e.g. attaching signature, referencing a key certificate).

Since we focus on the integrity aspect of a message which requires only signature infrastructure, we take the advantage of an already implemented signature infrastructure in the XWS-Security framework. In the process, we add Soap Account information in the outgoing message before incorporating WS-Policy in the message. This exception does not weaken our evaluation; rather it helps us to make a more solid claim. The XWS-Security framework which has both a signature and an encryption infrastructure, will incur more execution time in the sending side compared with the execution time of using the signature infrastructure alone. We show this in the following section.

7.1 Timing Analysis

We measure the execution time taken by a Web service invocation using two time frames: *Service Time (S)* and *Message Delay Time (M)*. Service Time is the time that the Web service takes to perform its task. In our case Service time is essentially the duration of detection of XML rewriting attacks. Message Delay Time is the time taken by the SOAP messages, in being sent/received by the invocation call. We simulate Message Delay using Random number to iterate a loop. It may be determined by the size of the SOAP message

being transmitted/received and the load on the network through which the message is being sent/received. We do not consider the size of the message, as the same message is transmitted each time in the simulation to get the clear measurement and the load on the network is out of scope here. To be more specific, Message Delay time increases by the increased SOAP size of augmented SOAP headers but it is not a WS-Security specific concern.

Total Invocation Time (T) for a Web service σ is given by the following formula.

$$T(\sigma) = M(\sigma) + S(\sigma)$$

Evaluating the above two components of T for a Web service invocation, help us to analyze the efficiency of a Web service. We perform tests to determine each of the above two components for a number of iterations for a policy driven solution versus our proposal. We generate the SOAP message in Fig 9 in the sender side and we simulate an attacker as a malicious intermediary which generates the rewriting attacks as in Fig 10. We send the same message to the receiver for a specific number of iterations, while the attacker generates the same attack same number of times. Fig 13 and Fig 14 show corresponding charts in line and timing diagram for 50 and 40 iterations using different timing resolution of Java profiling. It is clearly indicative that the proposed method shows better execution time in comparison to the XWS-Security policy driven framework.

7.2 Evaluation Environment

The sender (e.g. `AddSOAPAccount`) and receiver code (e.g. `CheckSoapAccount`) are written in Java and they are compiled and executed with Sun's `jdk1.5.0_06`, for windows. To be more specific we use XWS Security Framework of JWSDP 1.6 package for WS-Security features. The experiments are carried out using a PC with Mobile Intel(R) Pentium(R) 4, 2.80GHz Processor and 512 MB RAM, running on Microsoft Windows XP Professional.

7.3 Discussion

The data in Fig 12 are extracted using the `System.currentTimeMillis()` Java method which has a resolution of 15/16 ms. The result shows an impressive performance against policy driven validation. In average, service time using SOAP Account is 10 times faster than using a comparable policy based approach. Fig 13 shows another performance diagram obtained using a library called "hrtlib.jar" [19] instead of using `System.currentTimeMillis()`, which improves the accuracy of 15/16 ms to a fraction of 1 ms (e.g. 0.5 ms).

As SOAP supports a variety of message exchange patterns, such as request-response, one way message, RPC, and peer-to-peer interaction, XML rewriting attacks are possible in any patterns. So is the SOAP Account driven validation to detect these attacks.

8 Conclusion and Future Work

SOAP structure information has been ignored in detecting XML rewriting attacks so far. We have presented and discussed an inline approach to include SOAP structure information (SOAP Account) in the SOAP message and to validate the information by the receiver of the message. SOAP Account information can be used to detect the XML rewriting attacks immediately in the receiving end which might not be detected using the state of the art (e.g. WS-Security, WS-Policy, WS-SecurityPolicy) as it is showed in the section 3. This simple and elegant feature can be incorporated in WS-Security. In particular we can attach SOAP Account information into `<Security>` header in the WS-Security. We can even use it in any standalone web service which may be subject to XML rewriting attacks. It is not an at-

tempt to replace policy based validation; rather it is designed to be an alternative that can be used when performance is an issue in detecting XML rewriting attacks.

We have considered the SOAP structure information to be used in the context of securing single messages. Using WS-Security independently for each message to secure the integrity of a session of messages is rather inefficient. WS-SecureConversation [20] introduces security contexts, which can be used to secure sessions between two parties. How SOAP structure information can be used for securing a session is a future research topic. We have used only the XWS-Security Framework as a comparable message level security implementation and have drawn some comparisons of WSE implementation with our technique. It would be interesting to see how the performance scales regarding other implementation frameworks of message level security.

9 REFERENCES

- [1]<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- [2]K. Bhargavan, C. Fournet, A. Gordon, and G. O'Shea An Advisor for Web Services Security Policies, <http://research.microsoft.com/~adg/Publications/details.htm#ws05>
- [3]Microsoft Research; <http://research.microsoft.com/projects/Samoa/>
- [4]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.
- [5]T. Nadalin, ed., *Web Services Security Policy Language (WS-SecurityPolicy)*, Version 1.0, 18 December 2002, <http://www.verisign.com/wss/WSSecurityPolicy.pdf>
- [6]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)*, LNCS. Springer, 2004
- [7]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.
- [8]Ana C.C. Machado and Carlos A. G. Ferraz. Guidelines for Performance Evaluation of Web Services, *WebMedia'05*, December 5-7, 2005
- [9]Hongbin Liu, Shrideep Pallickara Geoffrey Fox, Performance of Web Services Security
- [10]XML-Signature Syntax and Processing <http://www.w3.org/TR/xmlsig-core/>
- [11]Bajaj, et al., *Web Services Policy Framework (WS-Policy)*, September 2004, <http://www.ibm.com/developerworks/library/specification/ws-polfram/>
- [12]T. Nadalin, ed., *WS-Policy Assertions*, 28 May 2003, <http://www.ibm.com/developerworks/library/ws-polas>
- [13]Java Web Services Tutorial <http://java.sun.com/webservices/docs/2.0/tutorial/doc/index.html>
- [14]Stefan Batres, ed., *Web Services Reliable Messaging Policy Assertion (WS-RM Policy)*, February 2005, <http://specs.xmlsoap.org/ws/2005/02/rm/WSRMPolicy.pdf>
- [15]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.
- [16]Web Services Addressing (WS-Addressing) W3C Member Submission 10 August 2004 <http://www.w3.org/Submission/ws-addressing/>
- [17]SOAP, <http://www.w3.org/TR/soap/>
- [18]<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-routing.asp>
- [19]Roubtsov, V. My kingdom for a good timer, <http://www.javaworld.com/javaworld/jwqa/2003-01/01-qa-0110-timing.html>
- [20]<http://specs.xmlsoap.org/ws/2005/02/sc/WSSecureConversation.pdf>
- [21]Microsoft Corporation. *Web Services Enhancements (WSE) 2.0 SPI*, July 2004. At <http://msdn.microsoft.com/webservices/building/wse/default>.
- [22]<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse/html/40c4b84a-a6e8-40db-810e-2521fdd8c09d.as>

Security Testing Web Applications throughout Automated Software Tests

Stephen de Vries

stephen.de.vries@corsaire.com

Corsaire Ltd. 3 Tannery House, Tannery Lane, Send, Surrey GU23 7EF
United Kingdom

Abstract. Testing software during the development phase has become an important part of the development lifecycle and is key to agile methodologies. Code quality and maintainability is increased by adopting an integrated testing strategy that stresses unit tests, integration tests and acceptance tests throughout the project. But these tests are typically only focused on the functional requirements of the application, and rarely include security tests. Implementing security in the unit testing cycle means investing more in developer awareness of security and how to test for security issues, and less in specialised external resources. This is a long-term investment that can vastly improve the overall quality of software, and reduce the number of vulnerabilities in web applications, and consequently, the associated risks.

1 Outline

The following sections are presented below:

- Section 2. An introduction to automated software testing;
- Section 3. A taxonomy and description of testing types;
- Section 4. An introduction to JUnit and examples of its use;
- Section 5. Testing compliance to a security standard using software tests;
- Section 6. Testing security in Unit Tests;
- Section 7. Testing security in Integration Tests;
- Section 8. Testing security in Acceptance Tests;
- Section 9. Conclusion; and
- Section 10. References.

2 Introduction

Software development methodologies generally make a clear distinction between functional testing and security testing. A security assessment of the application is usually performed towards the end of the project; either after, or in parallel with user acceptance testing, and is almost always performed by an external security testing team. This approach has a number of serious disadvantages:

- The cost of addressing issues identified in the testing phases after the bulk of development is complete is relatively high compared to fixing bugs identified during the development phase.
- Developer involvement in testing is minimal, which means that the people with the best understanding of the code are not involved with testing it.
- Developer (and overall project) buy-in into the security process is minimised since it is perceived as an external testing exercise performed by outside experts.

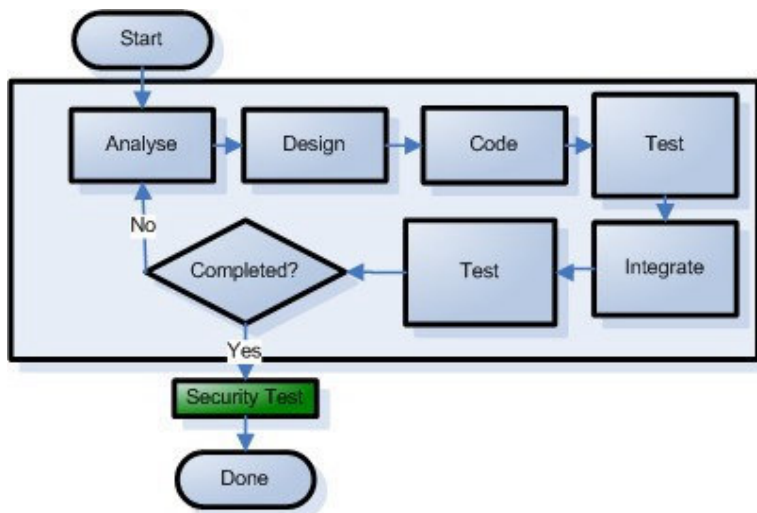


Fig. 1. Typical iterative software development cycle

Even in Agile methodologies that stress the importance of continuous and integrated testing, security is usually absent from the list of things to test. Developers tend to have their eyes fixed firmly on meeting the functional requirements without paying much attention to the security requirements. Security testing is again implemented at the end of the project, negating a lot of the benefits of an agile process.

2.1 Introducing Automated Software Testing

An automated software test is a software function used to verify that a particular module of source code is working as expected. Software tests can be written as:

- Unit Tests;
- Integration Tests; or
- Acceptance Tests.

Tests exist as distinct, self-contained source code entities that can be run against a given source code base at any time. Test cases should be written for all functions and methods so that their integrity can be tested at any point in the development process. It is important to know that a particular method functions as expected, and it is even more important to know that this method keeps functioning as expected after re-factoring and maintenance work, to prevent regressions.

Unit tests are used to test individual units of work, such as methods, functions or at most classes. These unit tests can be performed in complete isolation of both the rest of the application and also of each other. They excel at testing application and module states in exceptional conditions and not only the expected execution path. Security vulnerabilities are often introduced through software failures under precisely these exceptional conditions.

It is the thesis of this paper that security testing can, and should, be integrated into unit, integration and acceptance testing and that doing so will result in:

- A shorter security testing phase;
- More robust applications – because tests will be run on internal as well as external APIs; and
- Developer buy-in to the security process with its consequent advantages of better security in future applications.

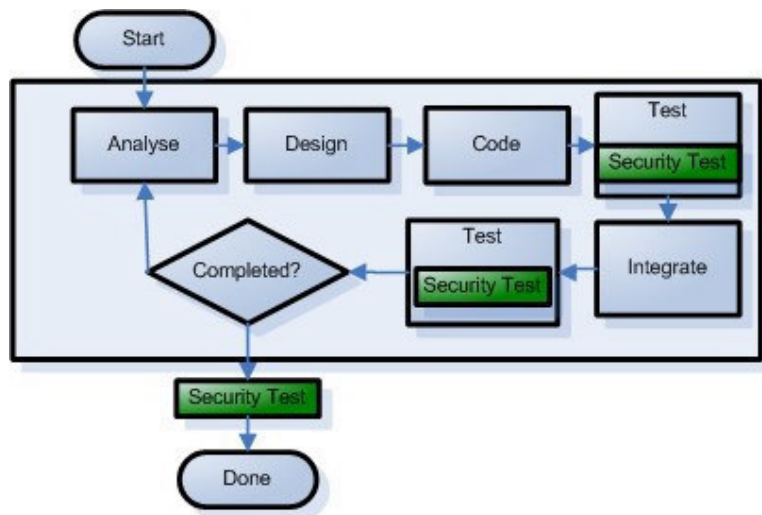


Fig. 2. Software development with integrated security tests

2.2 Use cases and Abuse cases¹

Software testing is usually aimed at testing only the functional aspects of an application. It is generally assumed that the application will be used normally, consequently it is only the normal conditions that are tested. This is precisely the kind of thinking that contributes to the proliferation of security vulnerabilities because the actions of a user with malicious intent was never considered when designing, building or testing the application.

In addition to testing the normal functional aspects of an application, it is essential that the abnormal abuse cases also be tested. Abuse cases can be derived from a formal risk analysis of the application and specific controls to mitigate the risks can be built into the application. This should be standard practice for secure development.

In addition to formal approaches, developers could also play an active role in identifying and mitigating abuse cases by always considering the abuse potential of even small pieces of code. Once a risk has been identified, it can be mitigated and the appropriate software test written to confirm its efficacy.

3 Taxonomy of Software Tests

Software tests can be divided into groups based on their granularity and which elements of the application are tested. This also brings us to the subject of “Test Coverage” which refers to how much of the code is tested. Where only QA testing is performed on the application, only those specific execution paths exposed by the external API will be tested. This is a form of shallow testing which could allow subtle and future bugs to go undetected. Security testing is likewise, typically performed only on the functional external API. In contrast, unit and integration testing operates at multiple layers and can allow virtually every method and every class in the application to be tested which results in a high degree of test coverage.

3.1 Unit Tests

Unit tests are performed on individual classes and methods to ensure that they properly satisfy their API contracts with other classes. At this level, unit tests must be tested as isolated units without any interaction or dependency on other classes or methods. Since applications are naturally dependent on other code techniques such as stubbing or using mock objects allow test developers to stub out the dependencies so that the subject class can be tested in isolation.

Unit tests are typically written by the developers themselves to verify the behaviour of their code. These tests provide an excellent control that the internals of a class behave as expected, but because of their limited scope they cannot test the integration between modules or classes.

¹See reference: McGraw, 2006

3.2 Integration Tests

Integration tests aim to test the integration of several classes as opposed to testing the classes in isolation. In J2EE environments, the web or EJB container provides a lot of important functionality and integration testing would therefore have to be conducted in the container, or by stubbing the relevant functions provided by the container. This class of tests could test interaction across the application tiers such as access to databases, EJBs and other resources.

Integration tests are also typically written by developers but are not executed as often as unit tests.

3.3 Acceptance Tests

Acceptance tests are at the far end of the spectrum and can be defined as the group of tests which ensure that the contract between the application API and the end user is properly satisfied. This group of tests is typically performed on the completed and deployed application and can be used to verify each use-case that the application must support. While it provides the least test coverage, it is essential in testing the complete integration of all the tiers of an application, including the services provided by application containers and web servers.

Acceptance tests are typically written by QA testers rather than by developers as the tests are far removed from the code and operate on the external API.

4 Introducing JUnit

JUnit is a Java framework for performing unit tests based on the original Smalltalk SUnit framework. Martin Fowler has said of JUnit: “Never in the field of software development was so much owed by so many to so few lines of code.”

JUnit itself is a very simple framework, but the impact it has on software development is where its true value lies. On its own, JUnit is used to perform unit tests, but integrated with other testing tools it can be used to perform integration and acceptance testing.

A simple example of using JUnit to test a method from a shopping cart class follows. Consider the following interface for a shopping cart that is implemented by the Cart class:

```
interface CartInterface {
    Iterator getAllCartItems(); //Returns all the items in the cart
    int getNumberOfItems(); //Returns the number of items in the cart
    boolean containsItemId(String itemId); //Checks whether an item is
already in the cart
    void addItem(Item item, boolean isInStock); //Adds an item
    Item removeItemId(String itemId); //Remove an item given its ID
    void incrementQuantityByItemId(String itemId); //Increment the quantity
of an item
    void setQuantityByItemId(String itemId, int quantity); //Set the
quantity of an item
    double getSubTotal(); //Calculate and return the subtotal
}
```

Below is the implementation detail of the addItem method that accepts an item and a Boolean flag as arguments and then adds the item to the cart. If the item is not in the cart, it is created and if it already exists a quantity counter is incremented.

```
public void addItem(Item item, boolean isInStock) {
    CartItem cartItem = (CartItem) itemMap.get(item.getItemId());
    if (cartItem == null) {
        cartItem = new CartItem();
        cartItem.setItem(item);
        cartItem.setQuantity(0);
        cartItem.setInStock(isInStock);
        itemMap.put(item.getItemId(), cartItem);
        itemList.getSource().add(cartItem);
    }
    cartItem.incrementQuantity();
}
```

If we were to design a unit test for this method, the following tests should be considered:

- Test that a new cart has 0 items in it.
- Test whether adding a single item results in that item being present in the cart.

- Test whether adding a single item results in the cart having a total of 1 items in it.
- Test whether adding two items results in both items being present in the cart.
- Test whether adding two items results in the cart having a total of 2 items in it.
- Test whether adding a null item results in an exception and nothing being set in the cart.

This can be implemented as a JUnit test case as follows:

```
public class CartTest extends TestCase {

    public CartTest(String testName) {
        super(testName);
    }

    protected void setUp() throws Exception {
        //Code here will be executed before every testXXX method
    }

    protected void tearDown() throws Exception {
        //Code here will be executed after every testXXX method
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(CartTest.class);
        return suite;
    }

    public void testNewCartHasZeroItems() {
        Cart instance = new Cart();
        assertEquals("0 items in new cart", instance.getNumberOfItems(), 0);
    }

    public void testAddSingleItem() {
        Cart instance = new Cart();
        boolean isInStock = true;

        Item item = new Item();
        item.setItemId("item01");
        instance.addItem(item, isInStock);
        boolean result = instance.containsItemId("item01");
        assertTrue("Add single item", result);
        assertEquals("1 item in cart", instance.getNumberOfItems(), 1);
    }

    public void testAddTwoItems() {
        Cart instance = new Cart();
        boolean isInStock = true;

        //Add a single item
        Item item = new Item();
        item.setItemId("item01");
        instance.addItem(item, isInStock);

        //Test adding a second item
        Item item2 = new Item();
        item2.setItemId("item02");
        instance.addItem(item2, isInStock);

        //Check whether item01 is in the cart
        boolean result = instance.containsItemId("item01");
        assertTrue("First item is in cart", result);

        //Check whether item02 is in the cart
        result = instance.containsItemId("item02");
        assertTrue("Second item is in cart", result);
        //Check that there are 2 items in the cart
        assertEquals("2 items in cart", instance.getNumberOfItems(), 2);
    }

    public void testAddNullItem() {
        Cart instance = new Cart();
        boolean isInStock = true;

        try {
            instance.addItem(null, isInStock);
            fail("Adding a null item did not throw an exception");
        } catch (RuntimeException expected) {
            assertTrue("null Item caught", true);
        }
    }
}
```

```

    }
    assertEquals("Null not in cart", instance.getNumberOfItems(), 0);
}
}

```

When this code is executed, JUnit will iterate through all the methods that start with the word “test”, then first execute the setUp() method, then the “test” method, followed by the tearDown() method as illustrated below.

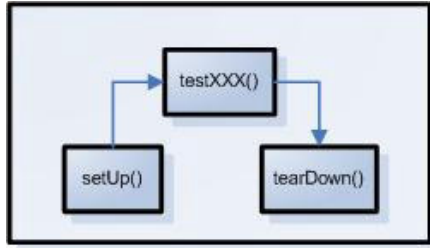


Fig. 3. JUnit’s execution of each testXXX method

JUnit can execute the test methods in any order. A closer look at the testAddTwoItems() method will illustrate how JUnit works. Firstly, a new shopping cart is created, then a new item is created and added to the cart. Similarly, a second item is created and added to the cart. Next the containsItemId method is called and the result stored in a variable. An “assertTrue” statement is made to ensure that the return value was true. The same method call and assert statement is performed for the second item. Finally another assert statement, this time “assertEquals”, checks that the number of items in the cart is exactly 2.

The “assert” statements make assertions about the code, should any assertion fail, it would mean that the particular test failed.

The testAddNullItem Method is an example of performing a simple test for exceptional conditions. It is important to know how the cart will behave if a null item is added to it. The test checks to ensure that an exception is thrown and that nothing was added to the cart.

JUnit is well supported in almost all Java IDEs as well as build tools such as Ant and Maven, this facilitates the execution of tests as a simple extension to the debug cycle rather than a distinct testing phase. Executing the above test case results in the following output:

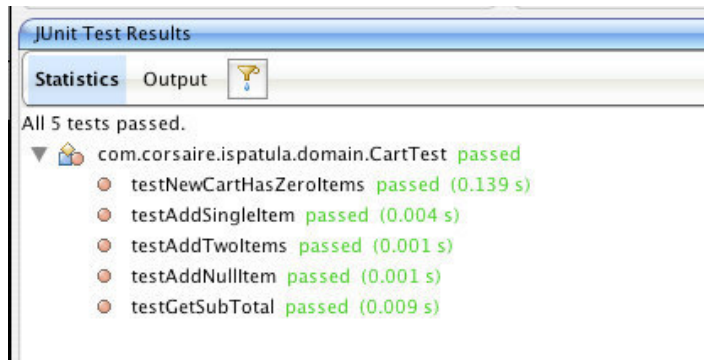


Fig. 4. Output of JUnit test case in the NetBeans IDE

The key to unit testing is that the writing and execution of tests is integrated into the development process, rather than being a distinct phase, and they can be executed at any time to ensure that code changes have not introduced regressions.

4.1 Advantages of using Unit Tests

Writing unit tests takes time and effort, but the benefits are substantial:

- They provide more test coverage of the code than QA testing which only tests the application from an external perspective.

- They allow re-factoring of the code and prevent regression. Since they are automated, it is very easy to run a test suite to ensure that all internal and external APIs function as expected after code or component changes.
- They allow teams of developers to work in parallel without having to wait for one team to complete required modules.
- They improve the design of the application by encouraging loosely coupled, pluggable components.
- They serve as living developer documentation to the code.
- They reduce the time spent debugging because component flaws are easily and quickly identified.
- They improve code quality because they encourage the developer to test for exceptional states that could cause the code to fail, instead of only concentrating on the expected execution path.

4.2 Unit testing frameworks for popular languages²

- Java – JUnit, (www.junit.org), TestNG (<http://beust.com/testng/>), JTiger (www.jtiger.org)
- Microsoft .NET - NUnit (www.nunit.org), .NETUnit (<http://sourceforge.net/projects/dotnetunit/>), ASPUnit (<http://aspunit.sourceforge.net/>), CSUnit (www.csunit.org) and MS Visual Studio Team Edition.
- PHP – PHPUnit (<http://pear.php.net/package/PHPUnit>), SimpleTest (www.simpletest.org)
- Coldfusion – CFUnit (<http://cfunit.sf.net>), cfcUnit (www.cfcunit.org)
- Perl – PerlUnit (<http://perlunit.sf.net>), Test::More (included with Perl)
- Python – PyUnit (<http://pyunit.sf.net>), doctest (included in standard library)
- Ruby – Test::Unit (included in the standard library)
- C – CUnit (<http://cunit.sf.net>), check (<http://check.sf.net>)
- C++ - CPPUNIT (<http://cppunit.sf.net>), cxxtest (<http://cxxtest.sf.net>)

5 Web Application Security Standards and the Coverage Offered by Unit Tests

The JUnit example in the previous chapter demonstrates the typical use in ensuring that the functional requirements of application components are met. In some cases this could include obvious security functions such as authentication and authorisation, but there are many more security requirements that are typically not included in the functional requirements. These non-functional security requirements should be included in unit tests so as to provide a fast, accurate and repeatable view of the security of the application at any point during the development process.

The security requirements of an application should be captured in an internal Standards document. Such a standard would be derived from the organisation wide security policy and from a risk assessment performed on the application. Depending on the requirements, a Security Standard could be derived for each web application, or an organisation wide Standard for all web applications could be adopted.

4.3 Example Web Application Security Standard

The matrix below presents an extract from an example security standard for a web application; and indicates which type of software test is able to verify each of the controls. A security standard such as this is essential in defining exactly how the application's security functionality should behave.

<i>Ref.</i>	<i>Category</i>	<i>Control Question</i>	<i>Unit</i>	<i>Integration</i>	<i>Acceptance</i>
AU-LO	Lockout	Is there an effective account lockout?		X	X
AU-S	Storage	Are authentication credentials stored securely?		X	
CO-AU	Authorisation	Does the application properly manage access to protected resources?		X	X
CO-PM	Manipulation	Does the application successfully enforce its access control model?		X	X
CO-LO	Logout/Log off	Is a logout function provided and effective?		X	X
SM-TR	Transport	Are Session IDs always passed and stored securely?			X
SM-CT	Cookie Transport	Where cookies are used, are specific secure directives used?		X	X

² A more list can be found at http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

<i>Ref.</i>	<i>Category</i>	<i>Control Question</i>	<i>Unit</i>	<i>Integration</i>	<i>Acceptance</i>
SM-E	Expiration	Are session expiration criteria reasonable and complete?		X	X
DV-I	Input Validation	Is all client-side input (including user, hidden elements, cookies etc.) adequately checked for type, length and reasonableness?	X	X	X
DV-SC	Special Characters	Are special characters handled securely?	X	X	X
DV-H	HTML Injection	Is HTML code as input handled securely?			X
DV-S	Active script injection	Is the application resilient to script commands as input?			X
DV-OS	OS Injection	Is access to underlying OS commands, scripts and files prevented?		X	X
DV-SQ	SQL Injection	Is the application resilient to SQL command insertion?		X	X
PT-L	Legacy data	Has all legacy data been removed from the server?			X
PT-E	Error Messages	Are all error messages generic to prevent information leakage?		X	X

It is clear that the vast majority of controls can be tested using functional, and to a lesser extent, integration testing techniques. Unit tests are only able to test a limited number of controls due to the fact that, in typical applications, a lot of security functionality is provided by other modules, the web server, or web container.

The next three sections will provide more detail on how to perform security tests in unit, integration and acceptance tests.

6 Testing Security in Unit Tests

Testing individual classes and methods provides a fine-grained approach to testing code functionality. Unit tests should be performed on individual classes and methods without a dependency on other classes. This limits the types of security tests that can be performed, but allows the tests to be executed very early in the development process.

6.1 Testing in Isolation

Unit tests should only test a single class and should not rely on helper or dependent classes. Since few classes exist in such a form of isolation it is usually necessary to create a “stub” or “mock” of the helper class that only does what is expected by the calling class and no more. Using this technique has the added benefit of allowing developers to complete modules in parallel without having to wait for dependent modules to be completed. To enable this form of testing it is important that the code is pluggable, this can be achieved by using the Inversion of Control (IoC) or Service Locator³ design patterns. Pluggable code using these patterns is a worthy goal in itself and the ease with which they allow tests to be performed is just one of their many advantages.

6.2 Vulnerability Testing Coverage

The number of security controls that can be verified through unit tests will depend largely on how security services are implemented in the application. A control concerning the ciphers used for the SSL session, for example, cannot be tested at the unit level since this is provided entirely by the application server or web server. Similarly, services such as meta-character encoding and access control could be implemented in the code, provided by a framework or by the application server. Usually only the former case can be tested in isolated unit tests.

6.3 Example: Testing Input Validation

Validation of user-supplied data can be performed in a number of different areas in a web application. To support modular application design, it is recommended that data validation be performed in the

³ Inversion of Control Containers and the Dependency Injection Pattern by Martin Fowler: <http://www.martinfowler.com/articles/injection.html>

domain object itself. The validation rules remain portable and will be executed even when the front end of the application is changed. An example of performing validation testing using the Spring framework and JUnit is provided below:

```
public class AccountValidatorTest extends TestCase {
    private Account acc = null;
    private AccountValidator validator = null;
    private BindException errors = null;

    public AccountValidatorTest(String testName) {
        super(testName);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(AccountValidatorTest.class);
        return suite;
    }

    public void setUp() {
        acc = new Account();
        validator = new AccountValidator();
        errors = new BindException(acc, "Account");
    }

    public void testValidPhoneNumbers() {
        //Test valid input
        String number = "232321";
        acc.setPhone(number);

        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
            errors.hasFieldErrors("phone"));

        number = "+23 232321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
            errors.hasFieldErrors("phone"));

        number = "(44) 32321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
            errors.hasFieldErrors("phone"));

        number = "+(23)232 - 321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
            errors.hasFieldErrors("phone"));
    }

    //Test invalid input
    public void testIllegalCharactersInPhoneNumber() {
        String number = "+(23)';[]232 - 321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertTrue(number+" did not cause a validation error.",
            errors.hasFieldErrors("phone"));
    }

    public void testAlphabeticInPhoneNumber() {
        String number = "12a12121";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertTrue(number+" did not cause a validation error.",
            errors.hasFieldErrors("phone"));
    }
}
}
```

When testing security functionality it is important that both valid input is accepted (a functional requirement), and also that invalid and potentially dangerous data is rejected. Testing boundary and unexpected conditions is essential for security tests.

6.4 Discussion

Implementing security tests at the Unit level is preferable to implementing the same tests at the integration or acceptance level because the tests are executed very early on in the development cycle. However, the number of security controls that can be tested as unit tests is limited by the fact that the majority of security issues facing web applications are simply not visible at the single class level.

7 Testing Security in Integration Tests

Integration tests aim to test the functionality of collaborating classes, including functionality provided by the Application server. Integration tests can be conducted using Mock objects or by running the tests within the container. In-container testing has the benefit of allowing developers to test the security services provided by the container such as access control and encryption. Compared to unit tests, many more security controls can be tested using integration tests.

7.1 Testing Strategies

There are primarily two ways to perform integration tests; using mock objects to provide a mock implementation of the application server API, or by running the tests in an application server (in-container testing). A number of projects⁴ ease the process of writing mock objects and provide mechanisms for mocking common API's such as the Java, Servlet and EJB APIs. Mock objects provide a general way to perform integration testing in any environment.

In-container testing requires specific tools for specific containers, consequently there are fewer options in this space. For J2EE testing popular choices are Apache Cactus (<http://jakarta.apache.org/cactus/>) and TESTARE (<http://www.thekirschners.com/software/testare/testare.html>).

7.2 Apache Cactus

Apache Cactus has become a standard testing tool for in-container testing of Java web applications. It allows testing of web and EJB applications and includes convenience plugins for Jetty, Ant, Maven and Eclipse. The disadvantage of using Cactus is that a container has to be started and stopped for the tests to run, for lightweight products such as Jetty this takes a few seconds, but for full blown J2EE containers this may be a lot longer.

7.3 Example: Testing Container Managed Access Control

Consider a web application that uses container managed security to restrict access to the `/admin/*` resource to only those users that are in the administrators role. To ensure that only those users can access the resource the following code performs three tests: first to verify that admin users can access the resource, then to verify that unauthenticated users cannot access the resource and lastly it checks that users from other roles cannot access the resource.

```
public class TestAccessControl extends ServletTestCase {
    public TestAccessControl(String theName) {
        super(theName);
    }

    public static Test suite() {
        return new TestSuite(TestAccessControl.class);
    }

    public void beginAdminAccessControl(WebRequest theRequest) {
        theRequest.setAuthentication(new BasicAuthentication("admin",
"admin"));
    }

    public void testAdminAccessControl() throws IOException,
javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
    }
}
```

⁴ See: <http://www.mockobjects.com> for more information

```

    }

    public void endAdminAccessControl(WebResponse theResponse) throws
    IOException {
        int position = theResponse.getText().indexOf("Welcome
    administrator");
        assertTrue("Administrator can view /admin", position > -1);
        assertTrue("false", false);
    }

    public void testUnauthenticatedAccessControl() throws IOException,
    javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
    }

    public void endUnauthenticatedAccessControl(WebResponse theResponse)
    throws IOException {
        assertTrue("Unauthenticated users must not be able to access
    /admin", theResponse.getStatusCode() == 401);
    }

    public void beginUnprivilegedUserAccessControl(WebRequest theRequest) {
        theRequest.setAuthentication(new BasicAuthentication("user",
    "password"));
    }

    public void testUnprivilegedUserAccessControl() throws IOException,
    javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
    }

    public void endUnprivilegedUserAccessControl(WebResponse theResponse)
    throws IOException {
        assertTrue("Normal users must not be able to access /admin",
    theResponse.getStatusCode() == 401);
    }
}

```

Cactus works by implementing tests in the whole HTTP request-response conversation. The life cycle of a single Cactus test is illustrated below.

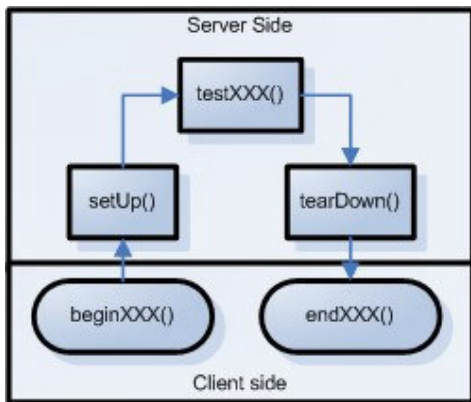


Fig. 5. Single Cactus test

For each testXXX method specified in the test case, the following actions are taken:

- Step 1. Execute beginXXX methods which setup client side data needed for the test, for example the beginUnprivilegedUserAccessControl method above sets the authentication credentials for HTTP BASIC authentication.
- Step 2. Execute the setUp() method on the server side if it exists. This method is used execute test code that is common to all server side tests. In the test case above, there is no such common code.
- Step 3. Execute the testXXX() method on the server side. This is where the core of the server side testing is done. Tests are JUnit tests and follow the familiar format.
- Step 4. Execute the tearDown() method which contains the common server side code to be executed when a test has completed. In the example above, none was required.

Step 5. Execute the endXXX methods on the client side. In this step the results returned from the server can be tested. The endUnauthenticatedAccessControl() method in the above test case makes an assertion to ensure that the HTTP status code for the response was 401 (Unauthorised access).

7.4 Discussion

The integration layer offers many opportunities to test for security vulnerabilities since the complete security feature set of the application is exposed and can be tested. Security tests that can be performed at this layer include Injection flaws, Authentication bypass and Access Control tests.

In container testing is a powerful form of integration testing that allows realistic tests to be run against the application because the testing is performed in a real application server. But this approach suffers from the overhead of starting and stopping the application server, as well as a limited number of testing frameworks. Although the number of security tests that can be performed at the integration layer is much more than at the unit testing layer, there are still some issues which can't be tested such as Cross Site Scripting and services provided by a web server (e.g.: SSL, URL filtering).

It is appropriate and common for developers to write integration tests, and in agile methodologies it is also recommended that integration tests be executed at least daily⁵ which will help identify issues early on. Skilled developers who have undergone training in security testing techniques and who understand software security issues will be well equipped to write security tests at this layer.

8 Security Testing in Acceptance Tests

Acceptance testing is at the far end of the testing spectrum and can be considered an automated form of QA testing. Acceptance tests are performed against the whole application as deployed in the application server. This allows complete testing of all application security functions, but does not offer as much test coverage as integration or unit tests. Writing security tests at the acceptance testing level is more appropriate for security or QA testers as opposed to developers.

8.1 Testing tools

There are a number of testing tools available; the Java based tools typically use the HTTP functions provided by the J2SE API or a custom HTTP client to perform the tests. They differ in how they handle the presentation tier and the degree of low-level access to HTTP they offer. Since they act as external HTTP clients, the language used by the client and that of the web application need not be the same. Popular tools in this space, include HttpUnit, jWebUnit, HtmlUnit, Canoo Webtest and the Ruby based WATIR project⁶.

Testing at the functional layer is more natural for a dedicated tester than a developer and since the tests require full end-to-end functionality they can reasonably only be run towards the end of the development process. Security testers are able to use the features of the functional tools to verify and reproduce the results of a manual security assessment. This could greatly reduce the time needed for retesting, since all the discovered vulnerabilities could be scripted during the initial assessment.

8.2 Example: Testing HTML injection with jWebUnit

The test case below checks to make sure that the search field of a web application is not susceptible to HTML injection.

```
public class XSSinSearchFieldTest extends WebTestCase {
    public XSSinSearchFieldTest(String name) {
        super(name);
    }

    public void setUp() throws Exception {
        getTestContext().setBaseUrl("http://localhost:8084/ispatula/");
    }
}
```

⁵ <http://www.martinfowler.com/articles/continuousIntegration.html>

⁶ For a more complete list see: <http://opensource-testing.org/functional.php>

```

    public void testHtmlInjection() throws Exception {
        beginAt("/index.html");
        assertLinkPresentWithText("Enter the Store");
        clickLinkWithText("Enter the Store");
        assertFormPresent("searchForm");
        setFormElement("query", "<a id=\"injection\"
href=\"http://www.google.com>Injection</a>");
        submit();
        //If the link is present, it means injection succeeded, therefore
our test should fail
        assertLinkNotPresent("injection");
    }
}

```

jWebUnit follows the familiar JUnit format for tests and extends this with HTML and HTTP aware functions. jWebUnit maintains an internal conversation state which keeps track of which page is being viewed and manipulated. In the test case above, the testHtmlInjection method performs the testing, most of the method calls are self explanatory. The setFormElement method is used to set the value of a form field, in this case an attempt is made to insert an HTML link into the “query” value. If the link is present after the form is submitted, then the test case should fail since the function is susceptible to HTML injection.

8.3 WATIR

The Web Application Testing in Ruby tool takes a different approach to the aforementioned tools in that it does not use its own HTTP client but instead drives an instance of Internet Explorer. This approach means that tests are representative of how real world web clients behave. But it has the disadvantage that low level tests (such as those testing at the HTTP level) have to be coded manually.

Ruby is a high level dynamic scripting language which can be understood by non-developers and programmers alike.

8.4 Example: Testing for SQL injection in a login form

```

require 'unittests/setup'
require 'watir'

$APP_HOME = 'http://localhost:8080/ispatula'
$USERNAME = 'corsaire1'
$PASSWORD = 'corsaire1'
$SQL_CONCAT_USERNAME = 'corsaire\'+\'1'

class SQL_Injection_Test < Test::Unit::TestCase
  include Watir

  def test_SQL_Blind_Injection()
    $ie.goto($APP_HOME)
    $ie.link(:url, /signonForm.do/).click
    $ie.text_field(:name, 'username').set($USERNAME+'\' OR 1=1--')
    $ie.form(:action, "/ispatula/shop/signon.do").submit
    assert($ie.contains_text('Signon failed'));
  end

  def test_SQL_Injection_String_Concat()
    $ie.goto($APP_HOME)
    $ie.link(:url, /signonForm.do/).click
    $ie.text_field(:name, 'username').set($SQL_CONCAT_USERNAME)
    $ie.text_field(:name, 'password').set($PASSWORD)
    $ie.form(:action, "/ispatula/shop/signon.do").submit
    assert($ie.contains_text('Signon failed'));
  end
end
end

```

As with jUnit, test methods are labeled by starting the method name with the word “test”. The \$ie object holds a reference to Internet Explorer and provides access to the entire IE DOM.

The test_SQL_Blind_Injection method first navigates to the logon form by calling \$ie.goto to access the application start page, then it finds a link in the page that matches the regular expression signonForm.do and clicks it. The text field with the name “username” is set to the value of: corsaire1’ OR 1=1– and the form is submitted. Next, an assertion is made to ensure that the page returned contains the text

“Signon failed”, if it does the test passes and we know that the form is not susceptible to this form of SQL injection.

The `test_SQL_Injection_String_Concat` method also tests for SQL injection, but uses a different technique, that of concatenating strings in an SQL statement. It first navigates to the correct page, then signs on with a username of: `corsaire+1`. If the application is vulnerable to SQL injection then that username will be concatenated to: `“corsaire1”` which is a valid username. Consequently, if the login is successful, then the application is vulnerable to SQL injection and the test case should fail.

8.5 Example: Testing Access Control

Consider an application that allows only administrative users to view all orders placed by accessing the URL: `/shop/listOrders.do`. Should regular users attempt to access this resource they should be redirected to the login page. The following test methods could be used to verify this behaviour:

```
def test_Access_Control_List_Orders_Unauthorised()
  #First check unauthenticated access
  logout
  $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
  assert('Please enter your username')
end

def test_Access_Control_List_Orders_Normal_User()
  #Check normal user access
  login('corsaire1','corsaire1')
  $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
  assert('Please enter your username')
end

def test_Access_Control_List_Orders_Admin()
  #Check administrator access
  login('admin','password')
  $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
  assert('Administrative functions available')
end
```

8.6 Example: Testing for XSS

The example application is susceptible to Cross Site Scripting in the search field, to test this, the script will insert Javascript that opens a new window. Watir will then attempt to attach to the new window, if the window does not exist then an exception will be thrown. Using the unit testing framework’s `“assert_raises”` function, it’s possible to check whether this exception is raised or not.

```
def test_XSS_In_Search
  $ie.goto('http://localhost:8080/ispatula/shop/index.do')
  $ie.text_field(:name,
'query').set('<script>window.open("http://localhost:8080/ispatula/help.html"
)</script>')
  $ie.form(:action, /Search.do/).submit
  assert_raises(Watir::Exception::NoMatchingWindowFoundException,
"Search field is susceptible to XSS") {
    ie2 = Watir::IE.attach(:url,
"http://localhost:8080/ispatula/help.html")
  }
end
```

8.7 Discussion

Security tests integrated into the acceptance test layer do not allow for as granular approach to testing as unit or integration tests, but they do allow for full testing of the external API of the application. Anything that can be tested during a black box security assessment of an application, can be tested for at the acceptance test layer.

External security testers that perform penetration tests and security assessments of applications can use acceptance testing tools to script common vulnerabilities. This is not as useful in detecting vulnerabilities, as it is in documenting and retesting vulnerabilities.

9 Conclusions

Unit testing of functional requirements of applications is already a well-established process in many development methodologies and is strongly emphasised by the Agile methods. If developers are trained in security then existing testing tools and techniques can be used to perform security testing. External security testers can also make use of testing tools to document and automate discovered vulnerabilities.

Security tests could be implemented:

- At the unit test layer by developers to ensure that the security requirements of the application are met, and that all failure and exceptional conditions are tested.
- At the integration test layer by developers and/or QA staff using either an in-container testing or mock object strategy to ensure that security services provided by components and the application server function as required and are free from common security vulnerabilities.
- At the acceptance test layer by security consultants and/or trained QA staff to ensure that the external API of the application is free from security issues. The creation of functional tests can accompany a manual security assessment of the application to provide an automated means of verifying all vulnerabilities discovered. Automated security retests could then be executed at any time with minimal overhead.

Implementing security tests in this manner provides a number of benefits:

- Developers are more aware of security issues and understand how to test for them in their code.
- Security issues are discovered early on when development and debugging is ongoing meaning that issues are addressed rapidly.
- Code is more robust since the tests can be executed at any point to confirm that changes have not adversely affected the security of the application.
- Code auditing is improved since the security standards or policies for web applications can be compared to the tests executed against the application.
- Overall security awareness is improved since it is an integral part of the process rather than an add-on.

Well-tested code that includes security tests results in an end product that is more robust, easier to maintain, naturally self-documenting and more secure.

10 References

MASSOL, Vincent; HUSTED, Ted: *JUnit in Action* (Manning Publications co.; Greenwich, CT, 2004)

MCGRAW, Gary: *Software Security* (Addison-Wesley; 2006)

FOWLER, Martin: "Inversion of Control Containers and the Dependency Injection Pattern"

(<http://www.martinfowler.com/articles/injection.html>)

- "Continuous Integration"

(<http://www.martinfowler.com/articles/continuousIntegration.html>)

jWebUnit (<http://jwebunit.sourceforge.net>)

WATIR (<http://wtr.rubyforge.org>)

"Open source testing tools, news and discussion" (<http://opensource-testing.org>)

Securing Web Services

Rix Groenboom
Rami Jaamour

Parasoft UK Ltd
Orchard Villa, Porters Park Drive,
Shenley, Radlett,
United Kingdom
{rixg, rjaamour}@parasoft.com

Introduction

Web services security constitutes the technological and managerial procedures applied to the system to ensure the confidentiality, integrity, and availability of information that is exchanged by the Web service. This article explores security issues specific to Web services and illustrates the engineering and testing practices required to ensure security throughout the Web services development life cycle.

Common Threats to Web Services and Web Sites

A Web service is an application that can be described, published, located, and invoked over the Web. A Web service is identified by a URI, whose public interfaces and bindings are defined and described using XML in a WSDL (Web Service Description Language) document. SOAP, a W3C specification, is the most common binding that is used to communicate messages between the service consumers (loosely known as clients) and the service provider (the server). SOAP determines how message data should be enveloped and formatted along with meta-data (headers).

There are many complexities specific to, and inherent in Web services that further complicate their security. Numerous threats can compromise the confidentiality, integrity, or availability of a Web service or the back end systems that a Web service may expose. Some of these threats are shared with conventional Web application systems (web sites), while others are specific to Web services. However, before delving into specific Web service security issues, it would be wise to first examine the general security threats that can occur in any Web application.

Typically, Web sites and Web services use common technologies in terms of the programming languages of the application. For example, both applications use data stores and application servers on the back end, and on the front end both typically use a Web server and are exposed over HTTP. Such architectural and technological similarities result in Web services inheriting many common Web site security threats.

SQL Injections

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach: if the hacker is able to break perimeter security and pass fixed inputs into the SQL statement, then these inputs can become part of the SQL statement. SQL injections can be generated by inserting spatial values or characters into SOAP requests, web form submissions or URL parameters. If the hacker knows his SQL, he can use this technique to gain access to privileged data, login to password-protected areas without a proper login, remove database tables, add new entries to the database, or even login to an application with admin privileges.

Capture and Replay Attacks

As Web messages are transmitted over the Internet, they are prone to man-in-the-middle attacks. A man-in-the-middle attack occurs when a malicious party gains access to some point between the peers in a message exchange. For example, a hacker may capture and replay a SOAP request to make a monetary transfer, or modify the request before it reaches its destination – ultimately causing severe losses for any of the peers in the message exchange.

Buffer Overflows

Native applications can suffer from unchecked input data sizes. If inputs are not validated, a buffer overflow attack can transpire remotely via SOAP requests or web form submissions. Buffer overflow attacks occur when a hacker manages to specify more data into one or more fields and write to the buffer beyond the size of the memory allocated to hold the data. Buffer overflows can result in application or system crashes, or when crafted carefully, they can even allow attackers to compromise the system and access unauthorized information or initiate unauthorized processes. The hacker can exploit this weakness so that the function returns to a hacker-designated function, or so that the function executes a hacker-designated procedure.

Denial of Service Attacks

Denial of Service (DoS) attacks are launched to compromise system availability. There are two ways to mount DoS attacks. First, attackers can consume web application resources to a point where other legitimate users can no longer access or use the application. This can be accomplished by sending a query for large amounts of data. The second approach can occur when attackers lock users out of their accounts or even cause the entire application to fail by overloading the service with a large number of requests. As we will see shortly, attackers could combine these two approaches with Web service specific attacks to maximize damage.

Improper Error Handling

Many application servers return details if an internal error occurred. Such details typically include a stack trace. These details are useful during development and debugging, but once the application is deployed, it is important that such details do not find their way to regular users, because the details may include information about the implementation and could expose vulnerabilities. For example, an error message about a bad SQL query indicates to a malicious user that his or her inputs are used to generate data base queries, thus possibly exposing a SQL Injection vulnerability.

As another example, a request that includes a wrong username or password should not be met with a response that indicates whether the username is valid or not; this would make it easier for an attacker to identify valid usernames, then use them to guess the passwords.

Threats Specific to Web Services

As we have seen, Web services are prone to all of the attacks that can be launched on a Web application. In addition, Web services are open to an additional class of vulnerabilities that exploit the idiosyncrasies of XML, WSDL (Web Service Description Language), and SOAP – the components that compose a Web service. The scope of these threats expand further when enterprises establish a SOA (Service Oriented Architecture) framework, which is an industry trend towards transforming organizations infrastructure into loosely coupled Web services that can be better governed, maintained and orchestrated to drive business process automation needs. SOA's reliance on Web services as the backbone to the system can introduce a more complex system thereby increasing security risks.

WSDL and Access Scanning

A WSDL document contains information pertaining to the Web service such as available operations, the content of the messages each of these operations accept and return, and the endpoints that are available to invoke the operations. Securing a Web service should rely on cryptographic measures to fully protect information rather than obscurity (security by obscurity should be avoided), because the information that a WSDL provides can expose certain architectural aspects about the Web service that could make it easier for an unauthorized user to mount an attack.

For example, attackers can determine valid request formats by examining the schemas and message descriptions in the WSDL that can contain operations such as `getItemByTitle`, `getItemById`, `placeOrder`, `getPendingOrders`, etc. Knowing this information, the attacker may be able to guess other possibly hidden operations such as `updateItem`, and perhaps use this information to place items on sale and then place orders using newly discounted prices. In fact, it is easy for a developer to introduce such vulnerabilities to a Web service since WSDLs are mostly generated automatically and contain a full description of the available operations. Developers may later comment out the operation section to hide it the service, but leave related information such as the message descriptions and schema types of these operations intact.

Broken Access Control

This vulnerability occurs when restrictions on what authenticated users are authorized to do are not properly enforced. Attackers can exploit these flaws to access other users' accounts, view sensitive files, or use unauthorized functions. For example, a Web service that requires requests to be digitally signed with a certain certificate should reject requests to all of the Web service operations that are supposed to enforce that requirement. When authentication and authorization tokens such as SAML (Security Assertion Markup Language) are exchanged, such tokens can contain complex authorization assertions that may not be enforced correctly in the implementation, which in turn exposes vulnerabilities.

External Entity Attacks

XML has the ability to build data dynamically by pointing to a URI where the actual data is located. An attacker may be able to replace the data that is being collected with malicious data. For example, in the following XML Entity, which can appear in a DTD (Document Type Definition) or an XML document, the external reference is declared with the syntax:

```
<!ENTITY name SYSTEM "Some URI">
```

The attacker could submit an XML request with such an entity using an arbitrary URI. This URI can either be pointed to local XML files on the Web service's file system to make the XML parser read large amounts of data, to steal confidential information, or launch DoS attacks on other servers by having the compromised system appear as the attacker by specifying the URLs of the other servers.

XML Bombs

DTDs may have recursive entity declarations that, when parsed, can quickly explode exponentially to a large number of XML elements. This consumes the XML parser resources causing a denial of service. For example:

```
<?xml version="1.0" ?>
<!DOCTYPE foobar [
  <!ENTITY x0 "Bang!">
  <!ENTITY x1 "&x0;&x0;">
  <!ENTITY x2 "&x1;&x1;">
  ...
]
```

```
<!ENTITY x99 "&x98;&x98;">
<!ENTITY x100 "&x99;&x99;">
]>
```

If it is processed, the DTD above explodes to a series of 2¹⁰⁰ “Bang!” elements and will cause a denial of service.

Large Payloads

Large payloads can be used to attack a Web service in two ways. First, a Web service can be clogged by sending a huge XML payload in the SOAP request, especially if the request is a well-formed SOAP request and it validates against the schema. Secondly, large payloads can also be induced by sending certain request queries that result in large responses.

For example, an attacker can submit a search request for available items containing a common keyword in the database. Even worse, if the query interface allows for it, the attacker might send a request to return all the available items in a large data base, which would consume resources and enable a DoS attack against the Web service.

Another attack style can be performed against stateful Web services by abusing the exposed operations to queue a large number of events or store a large number of items in the current session. This is executed by looping around certain operations then requesting the result from other operations in order to generate large responses. These bogus requests and responses will exhaust the service.

Malicious SOAP Attachments

Web services that accept attachments can be used as a vehicle for injecting a virus or malicious content to another system. The attachments that are delivered to a Web service can then be processed by other applications that have vulnerabilities. Or worse, if the attachment is an executable, or a package that contains an executable, then it is possible to infect that attachment with a virus to infect the machine that executes the file.

XPath Injections

XPath injections are similar to SQL injections in that they are both specific forms of code injection attacks. XPaths enable you to query XML documents for nodes that match certain criteria. For example, an XPath can be as simple as the following:

```
//*[local-name(.)="user"][attribute::username="somebody"]/@*[local-
name(.)="password"]
```

The above XPath returns the value of the password attribute for the username “somebody”. If such a query is constructed dynamically in the application code (with string concatenation) using invalidated inputs, then an attacker could inject XPath queries to retrieve unauthorized data.

Best Practices For Securing Web Services

Security has the inherent nature of spanning many different layers of a Web services system. Web services vulnerabilities can be present in the operating system, the network, the data base, the web server, the application server, the XML parser, the Web services implementation stack, the application code, the XML firewall, the Web service monitoring or management appliance, or just about any other component in your Web services system.

The key to effective Web services security is to know the threats as described before, understand the technical solutions for mitigating these threats, then establish and follow a defined engineering process that

takes security into consideration from the beginning and throughout the Web service life cycle. This process can be established in the following four steps:

1. Determine a suitable Web service security architecture.
2. Adhere to technology standards.
3. Establish an effective Web services testing process.
4. Create and maintain reusable, re-runnable tests.

By following these four steps, you can ensure complete Web service security.

Step 1: Determine A Suitable Web Services Security Architecture

Web services security architecture not only depends on the required security measures, but it also depends on the service scope and scale of deployment. For instance, security can either be enforced within the application server itself, or as a separate security appliance (such as an XML firewall) that can virtualize the service by sitting in the middle between the service and its consumers. Most Web service architects recommend decoupling the security layer from the application server in order to achieve better maintainability, flexibility and scalability. However, using a security appliance as an intermediary may not be necessary for simple end-to-end Web service deployments.

Another architectural decision to make is whether to implement the security on the transport layer or on the message layer. TLS (Transport Layer Security) is a mature technology, so both standards and tools have already been developed. It also provides a good transition path for engineers who are somewhat familiar with transport-level security but are new to Web services. On the other hand, TLS has inherent limitations that make it inappropriate for some situations. Fortunately, message layer security provides an alternative solution for situations where TLS's limitations are troublesome.

Transport Layer Security

The security of the transport that is being used for the Web service can be used to protect the Web service. For example, for HTTP one can enable Basic Authentication, Digest Authentication or SSL (Secure Socket Layer).

The main benefit of using TLS is that it builds on top of existing Web application experience to implement the security. Many developers know SSL and it is easy to enable it in common Web and application servers. SSL is particularly an ideal choice for end-to-end Web service integrations. SSL can enforce confidentiality, integrity, authentication and authorization, thus protecting the Web service from capture and replay attacks, WSDL access and scanning.

The drawback of SSL is that it is an all-or-nothing protocol. It does not have the granularity to secure certain parts of the message, nor can it use different certificates for different message parts. Besides, all intermediaries on the message path would need to have the proper certificates and keys to decrypt the entire message to process it then resend it over SSL again, which can be difficult or even impossible in some cases.

Message Layer Security

Currently, there is a lot of activity in the area of message level security, and it is fair to say that it is not nearly as mature as TLS. With that disclaimer, here are the message layer security technologies that may become the most important because they address some of the same concerns as TLS (privacy, authentication, message integrity) at the message level instead of the transport level:

- XML Signature provides a mechanism for digitally signing XML documents or portions of XML documents. The signature does not need to be in the document that is being signed, so you can also use XML Signature to sign non-XML documents.
- XML Encryption provides a mechanism for encrypting portions of XML documents. Encrypting a complete document is pretty easy; just treat it like a text document. There are some subtleties involved in encrypting portions of a document, however, and these are what XML Encryption addresses.
- WS-Security is perhaps most easily understood as a specification that defines a standard way of securing a single message by applying Username Tokens, XML Signatures, and XML Encryption to a SOAP envelope. The Username Token profile provides a simple way to describe authentication data, i.e., usernames and passwords.

When using one or more of the message layer security standards, it is important to use the combination that provides the required security protections. For example, Username Tokens alone cannot secure a message against capture and replay attacks unless they include a signed nonce and time stamp, and the SOAP Body is signed. The signature can be generated using the password in the token, or it can be independent of the password. However, if that password is not digested as recommended by the specification, then the token itself should be encrypted. As another example, XML encryption does not ensure the message authenticity or integrity, in which cases XML signature can be combined with encryption to ensure all three requirements.

Step 2: Adhere to Technology Standards

As in other security fields, adherence to standards is a necessary practice for Web services. There is a consensus among security professionals that publicly available, commonly used, well-analyzed cryptographic algorithms are the best choice, simply because they have already undergone a great deal of research and scrutiny as they were adopted by the industry. The same principle applies to Web services security.

For example, compliance with the WS-Security specification from OASIS will likely be safer than developing your own custom security implementation because it has been developed by experts in the field with threat protection in mind. Furthermore, you can reduce development time by using a readily-available implementation of the specification, and your service would be able to interoperate with other implementations of the same standard.

Another issue to consider with regards to adherence to standards is compliance with the Basic Security Profile (BSP) from WS-I. The BSP is intended to address interoperability, but in some cases it restricts the W3C and OASIS specifications in a manner that favors stronger security practices. Moreover, section 13 “Security Considerations” of the BSP lists a number of useful security considerations that should be taken into account when deploying secure Web services using WS-Security.

Step 3: Establish an Effective Web Services Testing Process

Understanding security threats is not enough. It is necessary to have a mature engineering process that makes security vulnerability detection and testing an indivisible part of the Web services development process so the threats are mitigated to the maximum extent. Thinking about security as early as possible throughout the Web service life cycle is key to achieve the best results in the most efficient manner.

A common pitfall that companies encounter is their attempt to use the same human and technological resources of Web QA and testing for Web services without implementing the proper training, processes and technology changes that can leverage such resources successfully. The same resources used for Web QA and testing cannot be used for Web services due to the following reasons:

- Web services testing requires a different skill set in XML, SOAP, WSDLs and other WS standards, let alone experience in security issues unique to Web services
- Web services testing can be better implemented with specialized tools rather than tools that are designed for traditional Web testing. The features of these tools need to support Web services standards and have the ability to design tests along these standards.
- Web services security testing requires the facilitation of tools and practices that can exercise the tests for exposing vulnerabilities that are general to Web applications and specific to Web services alike.

In order to detect and prevent security vulnerabilities in Web services, several engineering activities must be performed on multiple fronts. These activities can be summarized into three tiers.

Tier One Testing: Static Analysis

Knowledge of unsafe coding practices is crucial when developing secure software, but detecting such practices can be a tedious, time consuming process unless it is automated as much as possible. Static analysis tools are proving to be very effective in exposing dangerous method calls, insufficient validations, or poor code quality. Although manual code inspections can expose some of these problems, such problems can be subtle and difficult to find manually. Static analysis does not eliminate the need for code inspections completely, but it can significantly reduce the time and effort that is required to perform them since static

analysis tools can scan the entire source code to identify unsafe coding patterns, then the code reviewer can then analyze these instances to verify their severity. Without such automation, much more time would be spent in finding the unsafe coding patterns in the first place.

For example, in Java, using “PreparedStatement” is recommended over plain “Statement” to prevent SQL injections. A static analysis rule which searches for Statement.executeQuery() invoked with a dynamic string can pinpoint an engineer to this statement and provide a first line of defense against SQL injection problems. Other common insecure code patterns that can be found with static analysis include XPath Injections, uncaught exceptions that cause improper error handling, and some denial of service conditions caused by resource intensive operations.

Suspicious code patterns can also be identified with static analysis. Some security bugs result from programming negligence. However, more dangerous code may come from malicious programmers who hide Trojans, easter eggs, or time bombs in their code to provide discreet access at a later time. Such code often relies on random numbers or date/time checking to avoid detection, and it can change the normal security settings to allow surreptitious access. Static analysis rules which find all random objects and time date objects, called "triggers", and which find custom class loaders and security managers, can help a code reviewer identify and inspect suspicious code pattern.

In addition to detecting vulnerable or suspicious code, it is important to keep in mind that coding best practices play a role in producing secure code. There are also several different types of coding standards that can be enforced through static analysis which have general rather than specific security relevance and can improve the overall security posture of an application. For example, if code is found which has a synchronization problem, such a problem impacts security because synchronization problems tend to have unexpected effects. Indeed, coding practices should be considered during security testing.

Tier Two Testing: Penetration Testing

Not all security vulnerabilities can be found through static analysis, so penetration testing comes into the picture to expose such problems. Penetration testing dynamically exercises and scans the Web service deployed on a staging or production server.

Understanding the security threats allows the tester to design tests that can expose them with the help of good tools. For example, external entity attacks and XML bombs can be thrown at the service, to see if the service refuses to process XML processing instructions or DTDs by returning a SOAP Fault. WSDL access vulnerabilities can be detected by attempting to get a WSDL without the expected security channel if it is protected. For example, if the WSDL is protected with client-side SSL on port 443, then it should not be accessible on port 80; it is possible to forget an open connector in the Web server, which leaves multiple open channels. When it comes to thwarting WSDL scanning threats, then it is important to inspect the WSDL for redundant artifacts such as schemas or unused message definitions.

Capture and replay attacks can be simulated by sending multiple requests with the same message identifier that determines its uniqueness. For example, if you are using Username Tokens, you should test the service by sending multiple messages with the same nonce values and verify that the service rejects such requests properly. The service should implement a sufficient, but limited cache size for the recently accepted nonce values. Many WS-Security implementations do not take this into consideration by default, which makes them vulnerable to capture and replay attacks.

In order to test a Web service's vulnerability to DoS attacks caused by heavy loads, such DoS attacks should be simulated in fashion that is suitable to Web services. You cannot tell if a service can sustain a certain load scenario unless such scenario has been tested. However, it is important to execute such load tests in a manner that is effective.

Some test engineers have the tendency to perform load tests with the same static request to generate a load. Although this is a viable test scenario, is not sufficient because such DoS attacks could be detected by network security appliances. Therefore, Web service DoS attack simulations should be generated with dynamic request values that are semantically valid and that can exercise wider code coverage in the Web service's application logic in order to test the Web services to its limits. Such attacks are difficult to generate by manual coding, but they are possible with load testing tools that are specialized for Web services. In fact, the mere existence of such tools should alert Web service Engineers that such attacks can be done easily by a hacker if such tools fell into their hands. For example, to test a Web service that accepts Username Tokens with timestamps and nonces, it is important to apply a load on the service where the timestamps and nonces are generated dynamically for each request. Otherwise, errors such as the ones

caused by concurrency problems would go undetected. Another example would be load tests that send signed requests, where the hash and signature values should differ from one message to another.

Not only should Web service load tests generate dynamic requests, but such tests should also simulate real use case scenarios or usage patterns. For example, a use case scenario could be a Web service client retrieving an authorization token (such as a SAML assertion) from a security authority, then using that token for subsequent Web service invocations on different services. In order to test that scenario, load tests that keep using the same authorization token over and over again do not represent the real-world scenario since a real-usage scenario would have multiple users requesting and using multiple tokens at the same time. Executing such a realistic load test may expose concurrency or scalability problems that result in vulnerabilities. In this example, it is possible for the Web service to reject valid requests or accept unauthorized ones under a certain load even if such problems do not occur during regular functional testing.

In order to detect invalid responses during a load test, the load tests should be backed with sufficient response validations that ensure the detection of regressions from the correct behavior, because it is difficult to verify that all requests were met with the correct responses unless regression detection was performed while the load is being generated. Without response validation, only network connections and HTTP errors would be exposed which does not provide sufficient test coverage. For example, responses can be well-formed SOAP messages but with invalid data, or perhaps they contain an error message when they should not. Without placing sufficient response validation during a load test, such incorrect responses can go undetected.

Tier Three Testing: Runtime Analysis

Run-time analysis of the state of a web application code is needed to detect certain security problems that cannot be detected with the previous two tiers of testing. For example, in C/C++ applications that are exposed as Web services, memory corruption (especially memory corruption on the stack) indicates a potential for buffer overflows, which could cause serious security problems, and memory leaks make the application more vulnerable to DoS attacks. Dynamic analysis may find security vulnerabilities that can result from the integration of otherwise secure components because it takes data flow analysis into consideration, whereas static analysis provides large code coverage with narrower scope on data flows.

Combining the Three Tiers

Since each security testing tier provides a methodology exposing vulnerabilities from a unique aspect, combining two or more of the three tiers could provide a powerful approach to security testing. For example, static analysis can be used to determine the scope of the required penetration testing by recommending a more selective set of possible vulnerabilities to penetrate.

Run-time analysis combined with penetration testing provides the tester with visibility into the application as it performs under a variety of conditions. For example, one can perform run-time analysis during load testing in order to find memory leaks.

Step 4: Create and Maintain Reusable, Re-runnable Tests

The above testing practices can become too expensive to perform unless proper automation is applied to the testing process. Many organizations do not have the resources to perform these tests if they were to be performed manually and repeated for each project milestone.

Modern software development processes are iterative. Software Engineering activities should be performed on a recurring, iterative basis rather than following a rigid, one-directional development model that tests only at the end. Testing only at the end of the development cycle is the one of the main reasons behind late deliveries and exceeded project costs, and Web services are no exception to this fact.

However, such an iterative development model can only be effective if the engineering activities are backed with proper automation. Therefore, it is necessary to establish a Web services testing environment that is driven by automation that can help create the tests, maintain them, manage them and execute them on regular basis; typically every night as part of the existing “nightly” build and test process for the product. The alternative would be to run the various Web services tests manually, each one at a time, by modifying a client's request, which is a tedious, non-efficient process. It is therefore better to keep and

maintain all the Web service tests that are created so they can be re-run quickly, easily and so you can run them all automatically as regression tests whenever a Web service is updated.

After running security tests along the three tiers that were described, one may find problems that require fixes that ripple through Web service at a time when they be to risky or too expensive to fix, which is why such tests are better executed early regularly.

When a problem is discovered then the test that exposed the problem should ideally be added to the existing test pool and re-run on a recurring basis with all the other tests so it prevents that error from occurring again.

Conclusion

Securing your Web services is a vital aspect of ensuring a successful deployment. When deployed externally for consumption by partners or customers, only secure Web services can provide a justifiable integration solution, because the benefits they expose should far outweigh the risks. For true security, you need to understand the potential security risks and proactively minimize those risks. Using the right tool for the job is important, both in terms of products and technologies. Make sure that every security decision is followed by attention to detail in the implementation and by extensive testing, and you are on the way to developing Web services that are less vulnerable to attack.