Securing Web Services

Rami Jaamour

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 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 metadata (headers).

COMMON THREATS TO WEB SERVICES AND WEB SITES

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 might 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. A hacker who knows his SQL can use this technique to gain access to privileged data, log-in to password-protected areas without a proper log-in, remove database tables, add new entries to the database, or even log-in to an application with admin privileges.

RAMI JAAMOUR is a software engineer at Parasoft (www.parasoft.com) and a member of the company's Web Services Solutions team. He holds committer status at the Apache Software Foundation as he contributes to the WSS4J project, and he is also a contributing member of the WS-I Testing Tools Work Group. His experience with Web Services includes the development of effective Web services testing methodologies to ensure Web service quality and security.

Capture and Replay Attacks

As Web messages are transmitted over the Internet, they are prone to man-in-the-middle attacks. Such an attack occurs when a malicious party gains access to some point between the peers in a message exchange. For example, a hacker might 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 database 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 or not the username is valid; this would make it easier for an attacker to identify valid usernames, and 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 expands further when enterprises establish an SOA (Service-Oriented Architecture) framework, which is an industry trend toward transforming an organization's 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 Access and Scanning

A WSDL document contains information pertaining to the Web service such as available operations, the content of the messages each of these operations accepts and returns, and the endpoints that are available to

SOA's reliance on Web services as the backbone to the system can introduce a more complex system, thereby increasing security risks.

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 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 might 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 because WSDLs are mostly generated automatically and contain a full description of the available operations. Developers may later comment out the operation section to hide 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.

XML External Entity Attacks

XML has the ability to build data dynamically by pointing to a URI where the actual data is located. An attacker might 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 point 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 2^{100} "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. Second, 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 it, the attacker might send a request to return all the available items in a large database, 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 and then requesting the result from other operations 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 into 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 one 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 database, 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 a Web services system.

The key to effective Web services security is to know the threats as described previously, understand the technical solutions for mitigating these threats, and 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, one can ensure complete Web service security.

Step 1: Determine a Suitable Web Services Security Architecture

Web services security architecture depends not only on the required security measures, but also on the service scope and scale of deployment. For example, 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

The attacker might send a request to return all the available items in a large database, which would consume resources and enable a DoS attack against the Web service.

Transport Layer Security is a mature technology, so both standards and tools have already been developed.

decoupling the security layer from the application server 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 a particularly 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. In addition, all intermediaries on the message path would need to have the proper certificates and keys to decrypt the entire message to process it and 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 need not be in the document being signed, so one 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 message authenticity or integrity, in which case 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 one's own custom security implementation because it has been developed by experts in the field with threat protection in mind. Furthermore, one can reduce development time by using a readily available implementation of the specification, and one's service would be able to interoperate with other implementations of the same standard.

Another issue to consider with regard 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 one should take 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 that threats are mitigated to the maximum extent. Thinking about security as early as possible throughout the Web service life cycle is key to achieving 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 must 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.

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 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

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.

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.

reduce the time and effort that is required to perform them because static analysis tools can scan the entire source code to identify unsafe coding patterns; then the code reviewer can analyze these instances to verify their severity. Without such automation, much more time would be spent 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 that 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 discrete 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 that find all random objects and time date objects, called "triggers," and that find custom class loaders and security managers, can help a code reviewer identify and inspect suspicious code patterns.

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 that 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, 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 one is using Username Tokens, one 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.

To test a Web service's vulnerability to DoS attacks caused by heavy loads, such DoS attacks should be simulated in a fashion that is suitable to Web services. One

cannot tell if a service can sustain a certain load scenario unless such a scenario has been tested. However, it is important to execute such load tests in a manner that is effective.

Some test engineers have a tendency to perform load tests with the same static request to generate a load. Although this is a viable test scenario, it 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 should fall into their hands. For example, to test a Web service that accepts Username Tokens with time stamps and nonces, it is important to apply a load on the service where the time stamps 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, and then using that token for subsequent Web service invocations on different services. To test that scenario, load tests that keep using the same authorization token over and over again do not represent the real-world scenario because a real-usage scenario would have multiple users requesting and using multiple tokens at the same time. Executing such a realistic load test might 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.

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. Runtime 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. Because each security testing tier provides a methodology exposing vulnerabilities from a unique

Not only should Web service load tests generate dynamic requests, but such tests should also simulate real use-case scenarios or usage patterns.

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.

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.

Runtime 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 runtime analysis during load testing to find memory leaks.

Step 4: Create and Maintain Reusable, Rerunnable 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 by 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, inefficient process. It is therefore better to keep and maintain all the Web service tests that are created so they can be re-run quickly and easily, and so one can run them all automatically as regression tests whenever a Web service is updated.

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

When a problem is discovered, 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 as to prevent that error from occurring again.

CONCLUSION

Securing one's 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, one needs 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; then one is on one's way to developing Web services that are less vulnerable to attack.