WebServices - Axis

1. Web Service Security

1.1. Table of Contents

1.2. The challenge of server security

A standard attack on a web site is usually that of identifying and abusing badly written CGI scripts. Anything that gives read access to the file system is a security hole, letting people get at the code behind the site, often including database passwords and other sensitive data, plus of course there are the core parts of the underlying platform, which may contain important information: passwords, credit card lists, user-private information, and the like. Unauthorized access to this data can be embarrasing and expensive.

Having write access to the system leads to even greater abuses; defaced web sites may be created, spoof endpoints written to capture caller's data, or the database directly manipulated.

1.3. Is SOAP fundamentally insecure?

Some people, such as Bruce Schneier, have claimed that SOAP is a security disaster in the making, because of its ability to punch through firewalls. However, because in SOAP over HTTP the client can only make SOAP calls, not receive them, SOAP is no more insecure than any other application which POSTs XML files to a web server. The clients are safe unless the server (or its DNS address) have been subverted; the server is vulnerable, and does need to be secured.

Similarly, Bilal Siddiqui makes the claim that SOAP cannot distinguish between sensitive and non-sensitive web services and cannot perform user authentication, authorization, and access control.

Again, this is another example of excess panic, perhaps combined with a lack of knowledge of how SOAP servers are implemented. You do not need to follow this author's advice and have separate SOAP servers for every level of sensitivity, or XML and SOAP aware firewalls, any more than you need separate Web Servers for different users, or require HTTP aware routers to restrict parts of a web server to different IP addresses.

1.4. Common Attack Types

- Denial of Service to a server
- Interception and manipulation of messages
- Forged client requests
- Forged server responses
- attempts to read the server file system/database
- Attempts to write to the server file system/database

The most significant security risk comes from the fact that you are writing code to provide functionality to calling programs. If that functionality is offered to the wrong people, or if the code you wrote creates a security hole, "unexpected functionality", then you have a problem.

There is a large body of literature which covers securing web sites, such as the Open Web

Application Security Project Top Ten List of vulnerabilities, and their Guide to Building Secure Web Applications.

1.4.1. Special XML attacks

XML messages have a few intrinsic weakness, that Web Service creators should know about. None of these problems are unique to SOAP; anyone processing incoming XML needs to know and resist these.

1. Large XML Documents
   Have a client post an XML doc of extreme length/depth
   <foo><foo><foo>...</foo></foo></foo> This does bad things to DOM parsers and memory consumption on the server: a DoS attack. The issue here is that the costs of handling a large XML document are much greater than the cost of generating one.
2. Entity Expansion Attacks.
   If an XML doc header declares some recursive entity declarations, and the file refers to them, then bad things happen. Axis became immune to this between versions 1.0 and 1.1.
3. Entities referring to the filesystem.
   Here you declare an entity referring to a local file, then expand it. Result: you may be able to probe for files, perhaps even get a copy of it in the error response. As Axis does not support entities any more, it resists this. If your code has any way of resolving URLs from incoming messages, you may recreate this problem.

The other thing to know about XML is that string matching is not enough to be sure that the content is safe, because of the many ways to reformat the same XML.

1.5. Authenticating the caller

The new Web Service security proposals offer to authenticate your callers to your end point, and vice-versa. Axis does not yet implement these, but we do support XML signatures via a sister project.

The other approach is to validate at the transport level, using HTTPS. Configuring your web server to support https is definitely beyond the scope of Axis documentation: consult your server docs. To support https in the Axis client, you need to ensure the client has https support in the runtime. This is automatic for Java1.4+; older versions need to add JSSE support through Sun or an alternate provider.

Once you have HTTPS working at both ends you need to have the client trust the server certificate -usually automatic for those signed by central certification authorities, a manual process for home rolled certificates.

Clients can authenticate themselves with client certificates, or HTTP basic authentication.

The latter is too weak to be trustable on a non-encrypted channel, but works over HTTPS. The `MessageContext` class will be configured with the username and password of the sender when SOAP messages are posted to the endpoint; use the appropriate getters to see these values. Note that Axis does not yet integrate with the servlet API authentication stuff. Although the forms authentication is literally off-axis when it comes to SOAP calls, the UserPrincipal notion and integration with server configuration gives some incentive for integration. (this is a hint to developers out there)

Axis does not (yet) support HTTP1.1 Digest Authentication; if it does get added it will be via the HttpClient libraries.

1.6. Securing your Services

One of the key security holes in any Web Service is the code you write yourself. It won't have as many eyes examining it as the Axis source gets, deadlines get in the way of rigorous testing, and a complex web service will bind to the valued items: private data, databases, other servers, etc, that you want to defend against.

The key to this is not to trust the caller: their identity, their IP address and most of all, their data. Here are some attacks to consider.

1.6.1. XML attacks

We listed these attacks earlier. If your service takes XML from an attachment, or in a base-64 encoded string, parsing it as a standalone document, then you are exposed to all these attacks. Also watch out for standard XML syntaxes that integrate xlink or other ways of describing URLs to fetch -such as SVG. You need to ensure the renderer only fetches approved URLs.

1.6.2. Session Theft

Axis uses a good random number generator to generate session IDs, but someone listening to an unencrypted conversation could hijack a session and send in new messages. Recording sender info, such as the originating IP address helps, though beware of proxied systems (e.g. AOL) that may change the apparent origin of calls mid-session.

1.6.3. DOS attacks via load-intensive operations

Any request that takes time to process is a DOS attack target, as it ties up the CPUs. Authenticate before long requests, and consider watchdog threads to track really long execution times. If any bug causes a request to spin forever.

1.6.4. Parameter Attacks

If any parameter in the XML is fed straight into a database query, or some other routine that depends on valid data, then that data must be validated. Otherwise someone malicious could send a database update request, or some other string which lets a malicious user manipulate the system. This could even be as simple as changing their UserID in a request from that they set up in the session. Database attacks come from any situation where a parameter is inserted into an SQL query; the insertion of a semicolon ";" often permits the caller to append a whole new SQL command to the end of the first, and have it executed with the rights of the Web Service.

The key to defending against malicious parameters is to validate all data. Only accept a string containing only the characters/regular expression expected, and check its length. Better yet apply any other higher level checks 'userID==session.userID' that you can. Prepared Statements are the followon way of defending against SQL injection, as the JDBC runtime handles escaping of things. Don't try and build SQL strings by hand; it is a recipe for security holes.

Note that this would seem to argue strongly against mapping Session EJB objects to SOAP Endpoints. This is not the case. The Session bean must merely assume that all incoming data is untrusted, and so validate it all before processing further. This is exactly the kind of task a Service Layer should be doing.

1.6.5. Cross Site Scripting

In theory, a pure Web Service should be immune to XSS attacks, at least those that rely on having uploaded script displayed in an HTML Web Page server-side, script that is executed when the client views it. But the moment one takes Axis and integrates with one's own webapplication, any loopholes in the rest of the webapp expose this exact problem. We don't think Axis itself is vulnerable, because although it may include supplied data in a SOAPFault, this is displayed as XML, not HTML. Clients which don't distinguish the two could be an issue, as could anything we missed, especially in GET handling.

1.7. Securing Axis

A core philosophy is 'defend in depth', with monitoring for trouble.

1.7.1. Disguise

One tactic here is to hide the fact that you are running Axis...look at all the headers that we send back to describe the service, and if any identify Axis, edit that constant in the source.

While obscurity on its own is inadequate; it can slow down attacks or make you seem less vulnerable to known holes.

1.7.2. Cut down the build

Rebuild Axis without bits of it you don't need. This is a very paranoid solution, but keeps the number of potential attack points down. One area to consider is the 'instant SOAP service' feature of JWS pages. They, along with JSP pages, provide anyone who can get text files onto the web application with the ability to run arbitrary Java code.

1.7.3. Rename things

The AxisServlet, the AdminService, even happyaxis.jsp are all in well known locations under the webapp, which is called 'axis' by default. Rename all of these, by editing web.xml for the servlet, server-config.wsdd for the AdminService; the others are just JSP and WAR files you can rename. You may not need the AdminService once you have generated the server config on a development machine.

1.7.4. Stop AxisServlet listing services

To do this, set the Axis global configuration property `axis.enableListQuery` to false.

1.7.5. Keep stack traces out of the responses

By default, Axis ships in production mode; stack traces do not get sent back to the caller. If you set `axis.development.system` to true in the configuration, stack traces get sent over the wire in faults. This exposes internal information about the implementation that may be used in finding weaknesses.

1.7.6. Stop autogenerating WSDL

Trusted partners can still be given a WSDL file through email, or other means; there is no need to return the WSDL on a production server. How do you stop Axis returning WSDL? Edit the .wsdd configuration file, as described in the reference, to return a WSDL resource which is simply an empty <wsdl/> tag.

1.7.7. Servlets2.3: use filters for extra authentication

Servlets 2.3 lets you use filters to look at all incoming requests and filter them however you like -including validating IP address, caller credentials, etc. Caller address validation is useful for securing admin services and pages, even when other endpoints are public. Of

course, router configuration is useful there too.

1.7.8. Log things

Although full logs are a DoS attack tactic in themselves, logging who sends messages is often useful, for auditing and keeping track of what is going on. Add more log4j tags to whatever bit of Axis appeals to you to do this.

1.7.9. Run Axis with reduced Java rights

Java has a powerful and complex security system. Use it to configure Axis with reduced rights. Axis tries to write to WEB-INF/server-config.wsdd when updating the server config; and somewhere else (its configurable) when saving compiled .jws pages.

1.7.10. Run the web server with reduced rights

On Unix this is pretty much a given, but even on Windows NT and successors you can run a service as a different user. Make it one with limited rights. Make sure the core of the system has its access permissions tightened up so that the restricted-rights user can not get at things it shouldn't.

1.7.11. Monitor Load

To track DoS attacks, a load monitor is useful. `AxisBaseServlet` tracks the number of callers inside its subclasses at any point in time; the `AdminServlet` shows how to get at this data.

1.7.12. Consider 'tripwire' and 'honeypot' endpoints

With the core endpoints moved, why not create tripwire implementations of the admin endpoint, or a spoof endpoint listing under Axis/AdminServlet pointing to a honeypot endpoint that does nothing but send an alert when anyone sends a SOAP message to it. You then need a policy to act on the alerts, of course. A real honeypot would emulate an entire back end service -it would be an interesting little experiment to build and play with.

1.7.13. Monitor the Mailing Lists

We tend to discuss security on Axis-Dev, whenever it is an issue, but if demand is high we may add an axis-announce mailing list for important announcements.

1.8. What to do if you find a security hole in Axis

These days a lot of people love to make a name for themselves by finding security holes, and Axis, as part of the Apache product family, is a potential target. A hole in Axis could make many Web Services vulnerable, so could be serious indeed. So far we have only found a few of these, primarily in quirks of XML parsing rather than anything else.

1. Don't Panic. We have a process in place for verifying and fixing holes.
2. Don't rush to issue the press release to BugTraq. It is polite to let us know, and even verify that you are correct.
3. Test against the latest CVS version, not the (older) release builds. We may like already have fixed it, hacker dudes :-)
4. Email security@apache.org. Not the public axis-dev list, not jira. The security alias list is a list with representives from all Apache projects, so your report will be taken seriously.
5. Let us do a fix if possible, so we can announce that a fix is ready when you announce your finding. This doesn't take any of the credit way from the finder, just stops people panicing.

1.9. Automate Security Tests

If you find a security problem, write a test for it, such as a JUnit or HttpUnit test, so that you can regression test the application and installations for the problem. This is particularly important where it is a configuration problem that creates the hole; it is almost inevitable the same problem will re-occur on future installations.

1.10. Conclusions

We have shown some of the issues with Web Service security, things you need to think of in your own service, and how to harden Axis itself. Securing a system is much harder than getting a system to work, as 'work' usually means 'one or two non-critical bugs are OK'. From a security perspective, no security holes can exist for a system to be secure: no matter how obscure it is, someone may find it and exploit it. Be paranoid: you know it makes sense.

Finally, don't get put off writing SOAP services through a fear of the security implications. Any CGI-BIN or ASP/JSP page that takes parameters is as much of a security risk as a SOAP endpoint. For some reason, SOAP attracts dramatic press stories about infinite risk, perhaps because it is new and unknown. It isn't: it is XML posted to a web application, that's all. Only if you are scared of that, should you not write a SOAP service.