

Securing Java Web Services

Version 5.0

Instructor's Guide

Overview

The industry has been grappling with the problem of security for web services for several years. There has been a great deal of progress, resulting in a growing stack of interoperable security standards. But the range of requirements and use cases is still so broad that many solutions are built, of necessity, to a combination of accepted standards and home-grown approaches.

This course attempts to focus as much as possible on the broad concepts of service security – especially message-level security – and on individual standards that have been finalized and enjoy wide acceptance, such as XML Signature and Encryption, WS-Security, WS-SecurityPolicy, WS-Trust, SAML, and XACML.

Generally we prefer to deploy courseware that relies on accepted standards, because it helps to assure that what skills students develop will translate to their own work. That proposition is less compelling in this case – this is just the nature of the beast, but it may take some explaining. Even where there is an interoperable standard for message content or service metadata, there may not yet be a Java standard for implementing it in a portable way. For instance we implement WS-Security, policies, and related specifications with the help of Metro/WSIT – a leading open-source tool, but not a standard. To implement SAML we use OpenSAML – same story there. And for SAML SSO we use OpenSSO.

Try to emphasize the standards and patterns as much as possible, and remind students that while most of the later course exercises use APIs that may or may not be part of their company's architecture, they nonetheless prove the concepts of message-level security and that the SOAP messages they pass are excellent examples of interoperable security tokens, signatures, and encrypted data.

This course is designed to support, with some small customizations, at least two different audience requirements. Primarily, we expect senior Java developers. But so much of the content here is about XML (or SOAP, or WSDL, etc.) and not necessarily about Java, that another sort of student can benefit: someone who is technically proficient, is perhaps not a Java developer, or any sort of developer, but who needs to understand the nuts and bolts of web-service and SOA security.

For this reason – and because the technology area sprawls mightily – the course starts to feel a bit like reading an encyclopedia after a while. There is intentionally more material here than any one audience is likely to finish – or wish to cover – in one week. See the timeline section below and the chapter notes for ideas on how to focus the material for your actual student group.

Timeline

The following breakdowns are approximate, and every class will vary.

Day 1

2 hours	Chapter 1
1½ hours	Chapter 2
3 hours	Chapter 3

Day 2

2 hours	Chapter 3
2 hours	Chapter 4
2½ hours	Chapter 5

Day 3

2½ hours	Chapter 5
2½ hours	Chapter 6
1½ hours	Chapter 7

Day 4

3 hours	Chapter 8
3½ hours	Chapter 9

Day 5

1 hour	Chapter 10
1½ hours	Chapter 11
1 hours	Chapter 12
3 hours	Chapter 13

As mentioned above, there is more here than any one group is likely to finish in a week. To avoid letting the class develop into a race to plough through all this material, there are several likely areas for skipping or skimming, depending on your audience:

- Details of the DSig API
- The backgrounder on JAX-WS handlers
- The lab on encryption with Apache XML Security
- The backgrounder on JAAS
- The “full tour” of WS-SecurityPolicy symmetric bindings
- The “full tour” of WS-SecureConversation options
- Optional sections of many labs, especially the SAML Labs 8 and 9

Tools Deployed with the Lab Software

This course's software requires separate setups of a several major tools:

- The Java EE 5 SDK, Update 7
- Project Metro, version 1.5 (note that Metro 1.1 is already in Update 7, but has a few important bugs and limitations that 1.5 fixes)
- OpenSSO, build 7
- The Crimson text editor, or some similarly capable coding tool

For custom deliveries, note that Metro is only needed for Chapters 6-11, and OpenSSO is only used in the final chapter's SSO demonstrations. OpenSSO is a huge download, so if you're not expecting to use it, it's well worth skipping in the classroom setup.

Then, the lab software bundles a few more tools:

- JSTL 1.2 – this is here solely to support a couple of web applications used in the SSO demonstrations in the last chapter
- OpenSAML, version 2.2 – this is used in Chapters 7-12
- Tomcat 6.0 – instances of Tomcat host web applications used in the SSO demonstrations in the last chapter, so that we truly have multiple web/application servers, and not just multiple applications, participating in single sign-on
- Capstone's XMLValidator tool, and an overlay filesset meant to plug this tool in to Crimson so that students can do XML parsing and validating easily while editing the many XML files involved in the course

Ant Build Process

Though most students will be happy to let the **ant** command take care of things, some students (and most instructors) will want to understand the inner workings here a little better. Each project in the course has its own **build.xml** and **build.properties** files; these rely on the master environment variable **CC_MODULE** to import targets and properties in a central directory, **%CC_MODULE%\Ant** which is typically **c:\Capstone\JavaWSecurity\Ant**. Most applications follow one of two approaches to build a JAX-WS service in a deployable EAR, but especially in early chapters there's more of a mix-and-match of a few key targets:

- **compile**, which builds Java source code in **src** to **build/classes**
- **WAR**, **EJB**, and **EAR**, which build various Java EE archive formats
- **WSDL-to-Java**, which runs the **wsimport** tool
- **add-service-security-config** and **add-client-security-config**, which copy keystores and property files to appropriate spots in the application or web application class path, so that they can be loaded as class resources
- **deploy**, which deploys the completed EAR
- **run**, which launches a predefined application class

Those familiar with Ant will get the picture pretty quickly by reviewing four files:

- **c:/Capstone/JavaWS/Ant/JavaTargets.xml**
- **c:/Capstone/JavaWS/Ant/GlassFish2.1Targets.xml**
- **c:/Capstone/JavaWS/Ant/XMLSecTargets.xml**
- **c:/Capstone/JavaWS/Ant/XMLSecJavaEETargets.xml**

Teaching Notes

Chapter 1

This overview chapter is designed to be appropriate for those taking the entire course and also for audiences who will not learn the details of the W3C and OASIS specifications or the Java implementations but who do want a high-level understanding of the terminology, concepts and issues. There are no hands-on exercises, not even a code demonstration; it is more of an overture for the rest of the course.

Still, there is a lot to discuss here. Most students will be curious to understand the positioning of web-service security technology, and to learn what sorts of tools are used to solve what sorts of problems. Especially, this is the time to make the argument for message-level security as a distinct solution from transport-level (HTTPS) – maybe better overall, or maybe not, but certainly able to solve a class of problems that TLS cannot.

At this level, we use a handful of “scenarios” to guide the discussion. Many later chapters begin with one or more “use cases,” and the line between the two is maybe not straight and solid. But basically the use cases are more specific and, even if they are not rigorously worked in UML, they should each indicate to students one well-bounded and clear security requirement or practice that must be supported. The aim is for each of these use cases to motivate a large part of the discussion in its chapter; there are so many variations that it’s not possible to present use cases that then drive 100% of our coverage, but each one should be appealing at least as a common case for which a given technology (WSS, SAML, whatever) is well suited. Lab exercises are often full expressions of the use cases.

Chapter 2

This quick chapter illustrates what is really more of a web-application technique, which is HTTP BASIC authentication. But the demonstration does highlight some interesting issues specifically for web services: who supplies the logon, since there’s no interactive user? And how can the build continue to rely on the WSDL if it’s been secured?

The SOAPSniffer that’s introduced in this chapter will be an essential tool the rest of the way through; make sure students are comfortable with its use.

Chapter 3

Now we get into the nuts and bolts; this chapter and the rest of the course will be much more detailed and laden with hands-on work. We cover signature in good detail here, partly because it will help students understand the effects of using a WS-Security implementation to sign their messages later, and partly because at some point many applications develop a real need for direct implementation of signature and/or encryption of data or messages.

The chapter is quite long. We have a lot to introduce all of a sudden, before we can do anything really practical: signature; XML signature; Java API for XML signature; and even JAX-WS handlers (so we can plug our code into a web service). As with a lot of the course, take an easy pace with the examples, and plan for breaks mid-chapter.

The tools introduced in this chapter are helpful in later exercises. The SOAPSneak should be especially fun, as it allows students to simulate real service-oriented attacks.

Chapter 4

This chapter is another quickie, mostly because encryption is such a deep subject, and much more than signature it is something usually best left to professionals and/or generic tools. The theory in this chapter outstrips the practice. The exercise that's given already takes students into some pretty deep waters. So we leave it at that, and expect to let tools such as WSIT do our encrypting for us soon.

Chapter 5

This chapter is usually a lot of fun to cover. It brings in a number of big concepts, introducing WS-Security and the various token types. It gets us started with real message-level security examples. And the ability to modify existing client-to-service traffic by plugging in different XML configs, and see fully-worked examples of WSS headers as a result, all this tends to perk students up. Though the sample messages are of course quite detailed, it's worth taking plenty of time to deconstruct them, following the references and IDs around the message header elements and the body itself. Again, as a standard for interoperable security, WSS is much more interesting (or should be) than XWSS, so try to channel the discussion in that direction as much as possible.

You might point out that the two use cases at the top of the chapter are closely analogous to two major web-application security techniques: the first is much like HTTPS, except at the message level, and the second serves the same purpose for web services that HTTP BASIC or DIGEST schemes serve for web applications.

The big shortcoming of XWSS proper – that it requires configurations according to a proprietary grammar – happens to lead very nicely into the next chapter! It's a good idea to point out, maybe a few times, that the XWSS configuration files are not long for our world, as we'll be driving XWSS, as part of WSIT, from WS-SecurityPolicy files very soon.

Chapter 6

This and the previous chapter line up rather like the SOAP and WSDL chapters in our web services course: data and metadata, and what happens over the wire between pieces of software vs. contracts that drive the software in the first place. The policy grammars do take some getting used to, especially WS-Policy with its weakly-typed, anything-goes model. We give a series of progressively more complex examples to ease folks in. The difference between compact and normal form, too, is quite the brain-teaser. Our approach is it's worth knowing the difference and that policies can look a bit like this or more like that, because students will see each approach used in practice. But certainly it's nothing on which to dwell for too long.

When we start to attach policies to WSDL descriptors, suddenly it will seem a lot more real! And quickly after that we start to see the concrete assertions and bindings of WS-SecurityPolicy. We cover quite a lot of detail here, and still there's plenty of options left unexplored. The big Pizza/WS-SecurityPolicy example could be abbreviated considerably if people's eyelids are drooping or you're running late. But the lab exercise is a lot more focused, and should be worthwhile, even the optional steps if time allows.

You might expand on this chapter a bit by revisiting the issues raised at the end of Chapter 5 regarding nonces and timestamps: i.e. how well does WSIT perform against the same sorts of replay/tampering attacks? If you run these attacks using SOAPSneak against the Pizza service, you should see that, on one hand, username tokens are repeatable/replayable, because WSIT doesn't support hashed passwords yet and so there's no nonce to rotate. On the other hand, WSIT is better about observing expiration timestamps.

Chapter 7

SAML represents a different basic approach to a range of security problems that overlaps significantly with those solved by the WS-* stack, but is not exactly the same set. So this chapter is naturally going to feel like changing gears, as we launch into several chapters on SAML and hence starting to look at things more from this other perspective. Our exercises, though, will sew WS-Security and SAML together in various ways, and hopefully that will help students to keep a sense of a single narrative in their heads.

This brief first chapter is a good place to talk about the general usefulness of SAML; where and why it is used; how it relates to LDAP, ADS, and so on. A nice pithy little statement that helps to establish SAML as useful apart from WSS itself is that WSS sets a standard for carrying security information in the headers of messages between parties who are interested in other things; while SAML expresses information that (especially SAML protocol and the SOAP binding) might also be the actual body of conversation between two parties whose primary interest is the security assertion itself.

In this chapter we bring the OpenSAML toolkit into play, and get it working with GlassFish. The coursebook explains pretty well the tension between the two, as OpenSAML hates the Sun XML parser and GlassFish naturally wants to load that same parser. Even after the configuration laid out in the book, you may see error messages from OpenSAML on server restarts. We're not certain why this is, though it seems to have to do with class loading in various server and application spaces. If you do notice errors when re-starting the server, and then certain services don't work that worked before, almost always the fix is simply to re-deploy the affected service: this seems to clean things up as the new service classes are loaded into the server process.

Chapter 8

Most of the information in this and the next chapter is pretty pedestrian, to be honest: just a lot of details about a particular XML model. There are some interesting concepts tucked in there though, and hopefully the exercises help to drive home the point that with SAML we can express a much broader range of security information than WSS is designed to support. The lab focuses on the sender-vouches use case, which seems to hit the sweet spot of interaction between WS-Security and SAML, and helps us retain some continuity with earlier chapters via the Healthcare case study.

Be sure students get the point of the trust discussion that drives the optional part of the lab: to wit, why do we suddenly need a client certificate when we didn't before? And it's eye-opening at this point for many to discover that just because they've been calling tokens "signed" via a WS-SecurityPolicy, that doesn't necessarily mean signed by a client certificate just because the tokens are in the request: until now we've been seeing HMAC signatures based on a negotiated symmetric key, which in turn means that we've had no real authentication of the client!

Chapter 9

This chapter is again mostly just model details, but the final exercise should prove interesting, especially because it takes us into the sort of multi-node, federated service architecture for which WSS and SAML are really designed.

The lab exercise may also trigger some consideration of the relationships between the Clinic, Hospital, and HealthcareIdP: how is it that two of these entities know the same users by the same names? So far we don't have truly federated identity, because the user records are still scattered over multiple realms, but this does get one thinking about centralization. This can be a nice way to foreshadow Chapter 13 on federated identity and SSO.

It seemed to be too much for the coursebook, but either before or after the lab you might point out some code in an earlier step of the case study that would have done an enveloped signature of the sender-vouches assertion. Specifically, in **Examples/Healthcare/Step10**, open **Clinic/src/cc/wss/LoginHandler.java** and search for `"/*SIG"` to see the section that's commented out of the **handleSAMLCallback** method. This would put a signature into the assertion itself, and the policy signature requirement could be removed. But, WSIT on the service side chokes when we try this: it scans messages for signatures and attempts to validate them, even if not called for by the policy, and it can't seem to handle the enveloped sig. (You may notice that the code doesn't put the signature in exactly the right place, as the answer to this chapter's lab does for the attribute query. But we have tested this to discover that it doesn't help; this is just a WSIT limitation at the moment.)

Finally, when building the HealthcareIdP application, you will see several hundred “Visiting non-standard Signature object” messages in the verifier. There’s a tool embedded in the verifier that drops these messages, and they can’t be filtered out. Our other projects see only a handful of these; the big difference is that this is an SAAJ servlet rather than a JAX-WS service, and this triggers different validations having to do with classes being loadable at runtime. This is unfortunate, but harmless; just tell students to ignore it.

Chapter 10

Chapter 11

Chapter 12

Chapter 13

Revision History

Version 5.0 re-invents the course for Java EE 5: WS-Security 1.1, SAML 2.0, JAX-WS services, WSIT, and other latter-day advances. The basic structure of the original course has been mostly preserved, but there are major changes:

- We've tried to motivate the overture chapter with a handful of high-level "scenarios," and most of the remaining chapters with more specific "use cases."
- Some of the material from Chapter 6 was folded into the end of Chapter 5, and a lot of it was just dropped as it wasn't that compelling anymore given the advent of WS-SecurityPolicy and Metro.
- A new Chapter 6 covers WS-SecurityPolicy and how WSIT will drive WS-Security behavior from interoperable policy documents.
- The main case study, Healthcare, still develops into a three-party scenario with a back-channel SAML conversation, but we've dropped the authorization-decision query to an Assurance service, as this now looks stale as XACML seems to have won out over SAML for this sort of logic and messaging. Instead, the Hospital makes an attribute query to a "HealthcareIdP;" this seems much more in keeping with current practice and should be a more compelling architecture to discuss.
- A new chapter on XACML doesn't have any working code but does give a good idea of how XACML fits into the bigger picture and how it is replacing SAML authorization-decision activities.
- A new chapter gives an overview of WS-Trust and WS-Federation.
- A new chapter covers the SAML bindings, with emphasis on the HTTP bindings preparatory to the final chapter on federated identity and SAML SSO.
- A new chapter covers federated identity and SSO scenarios using SAML 2.0. This has no lab exercises, but there are two long and interesting examples that show the SSO scenario in excellent detail.

Instructors should plan for significant preparation time before teaching this version, even if they've taught earlier versions before.

Version 1.4 was the initial release of this course, targeted to technology that was the state of the art circa J2EE 1.4.

Troubleshooting

If you run into any trouble with code exercises, the first and best thing to do is to double-check that the classroom machines have been set up precisely according to the course setup guide. Especially, the wrong version of a tool can cause significant problems; don't wander off-book in this way unless absolutely sure you can support the software that you prefer and that we haven't tested. Check environment variable settings carefully, too; these are the cause of a great many classroom glitches.

Below are some specific pitfalls that have come up in previous offerings of the course:

- The setup guide requires that the Java EE SDK be set up with an admin logon of **admin/ccstudent**. If any other username or password is set when the server is set up, the Ant builds will fail. You can adapt to the real admin logon by setting values in two files found in **c:/Capstone/JavaWS/Ant**: **AS9.0.properties** has the admin username, and **AS9.0.password** has the admin password.
- See the notes in Chapter 7 about OpenSAML vs. GlassFish, and how you can get errors even after doing the required server configuration. In brief, the workaround is to re-deploy the affected web service after a server restart.
- If the Java EE SDK is set up somewhere other than **c:\Sun\SDK**, this should be no trouble until around Chapter 6. Then, unavoidably, some hard-coded paths do start to show up in policy files. If you start running into problems as of the **Pizza/WS-SecurityPolicy** example, look to the client-side WSDL files and see about editing the keystore paths that begin with **c:\Sun\SDK**. This will come up in the final step of the Housing example as well.

Errata

There are no errata outstanding for this course at this time.

Feedback

We very much appreciate whatever feedback we can get on our courseware – especially from the instructor’s perspective. Naturally, the more specific, the better, and we strongly encourage you to make notes on issues you may encounter in the classroom, whether they’re typos, missing files, or suggestions for clearer language to explain a concept. We can’t guarantee that we’ll act on every suggestion, but we’re aggressive about stamping out problems and try to be highly responsive. Hopefully this means that when you give us good feedback, you get a better course the next time you need to teach it.

Please direct all courseware feedback to

Will Provost
Capstone Courseware
<mailto:provost@capcourse.com>
877-227-2477

For anyone who’s interested, we have a very informal defect-tracking system, based in Excel spreadsheets with columns to capture defect location, nature, status, and author feedback. Ultimately, feedback goes into these sheets, so if you want a template, we’ll be happy to provide one, to facilitate the reporting process.