ShibbolethWalkthroughStepOne

Initial Access to ServiceProvider, AuthnRequests, SessionInitiators

In the general case, the Shibboleth sequence begins with every access to a Shibbolized web server. In its current incarnation, the SP runs as a "filter" in the various web servers it supports, meaning it executes as part of the basic HTTP request processing of the server.

RequestMap Processing

When a request comes into the filter, the first important step is to "map" the request.

A deployer uses the RequestMap facility in ShibbolethXml to apply settings to resources based on the URL (scheme://host:port/path). The SP supplies RequestMap implementations that maintain settings in an XML syntax as well as processing of native web server commands, such as in Apache. These are entirely equivalent, and native commands generally override the same settings in XML.

In order to determine the right settings to use (in particular when the settings are specified in XML form), the request URL has to be "decomposed" into pieces so that the RequestMap contents can be "overlaid" onto those pieces. This process turns out to be fairly tricky because web servers vary widely regarding how they provide access to the requested URL, and because much of the information they make available is actually based entirely on what the browser asked for. To see why this is bad, think of some of the ways you might access a simple URL like https://sp.example.org

- https://sp.example.org (the "official" URL)
- https://sp (automatic DNS lookup for example.org domain)
- https://mynameforsp (a private name in a hosts file)
- https://192.168.1.1 (the IP address)

Now imagine trying to create a rule for this request that would work for every possibility. No chance. In a web server by itself, you don't worry about this because the commands you define are applied in various ways that are independent of the specific URL requested. For Shibboleth to support settings in a portable XML form, something more is needed to prevent the client circumventing settings simply by entering the same effective URL in a different fashion. What is this something?

Hostname Canonicalization

The trick is to establish the "official" hostname for a request inside the Shibboleth filter so that only one canonical rule needs to be defined using the RequestMap syntax. No matter what the client actually says, the hostname is changed internally to the official value. How? It varies by server, but this aspect of configuring Shibboleth is probably the easiest to screw up or forget, and it usually leads to questions like "why is the filter not blocking the requests and just passing them along?"

Apache

In Apache, a command called UseCanonicalName is used to cause Apache itself to provide Shibboleth with the right hostname. Unfortunately, this command is Off by default, and must be added or turned On in the Apache configuration. **Only in very rare cases should a Shibboleth deployment leave this setting off. It is almost always a potential security hole if you do.**

But for the process to work, you MUST tell Apache what the right server hostname should be. This is generally done with the `ServerName` command. This allows Shibboleth to ask Apache what the hostname is for each request, a key to the mapping process.

IIS

Unfortunately, IIS provides **no** support for establishing the canonical hostname for a request. This is a serious bug. To work around it, Shibboleth has a special IIS configuration section in ShibbolethXml in the `<ISAPI>` element. A deployer on IIS MUST fill in this element by creating a `<Site>` definition for each web site instance that is using Shibboleth. This element maps the IIS internal site number (usually termed the INSTANCE_ID) to the canonical hostname of the site.

As an additional feature, you can supply nested `<Alias>` elements that allow alternate hostnames to be supplied by a browser for a given site. This causes requests for the alias hostname to be "remapped" internally to the official hostname so that the RequestMap can be simpler.

iPlanet

iPlanet is less well-documented on this subject. The NSAPI filter will attempt to determine the right hostname in various ways, but a value can (and should) be supplied by the deployer in the filter initialization command.

URL Components

During the mapping process, the request URL is "split" into:

- Scheme (http or https)
- Hostname (see above for how this is determined)
- Port
- Path

Note that the query string, if any, is ignored for mapping purposes.

It should also be noted that to a lesser extent, the scheme and port information for the request is also sometimes difficult to determine, particularly when the web server is heavily virtualized. The browser may be requesting https and port 443, but the web server itself sees http and port 80.

Apache has commands (different across versions) for establishing the external port when the internal port is different. There is no such command for the scheme, but Shibboleth includes a ShibURLScheme command to override the scheme and fool itself into assuming https is used.

With IIS, this information can be supplied inside the <Site> elements using the `scheme` , `port` , and `sslport` attributes, if the external and internal values do not match.

The Mapping Process

Believe it or not, all of the above work happens on **every** single request. That's literally just the bootstrap for the real work, which is to map the request, now that it's been chopped into bits. Each bit is fed into the mapper, which is a collection of <Host> (and inside those, `<Path>=) elements that are used to apply settings to the requests. By default, a single =<Host>` is used to match all requests for a given hostname on the default http and https ports.

It's possible to fine tune this and map http and https requests separately. You might do this to force Shibboleth to ignore non-https requests so that the web server can run a built-in blocker to reject non-SSL requests.

The goal of the mapping process is to find the deepest nested element that applies to the request, to serve as the starting point for determing each property that is in effect. A pointer to this spot is passed back to the filter so that when a property is requested, if it's not set there, it can be inherited from up above by walking back up the XML tree until it's found or not.

Protection Settings

All of this work serves the basic goal of figuring out how to deal with each request...should the filter look for an active session and validate it? If no session exists, should the filter insist on one by kicking control to a SessionInitiator? If so, which one? Or should nothing happen, and control just pass along to the web server and eventually the resource?

The settings that determine the answers are:

- AuthType="shibboleth" (Apache Equivalent: AuthType shibboleth)
- applicationId="label" (Apache Equivalent: ShibApplicationId label)
- requireSession="true|false"(Apache Equivalent: ShibRequireSession On|Off)
- requireSessionWith="label" (Apache Equivalent: ShibRequireSessionWith label)

The `AuthType` setting is sort of the master control. If this isn't set to `shibboleth` (case doesn't matter), then the filter says "must not be my job" and just passes control back to the server. This is usually because the content is public, but could also be content protected with some other security module, such as a basic-auth plugin, IIS/Windows security, etc. For legacy reasons, the `requireSession` command (when on/true) implies the filter should handle things, because `AuthType` was added later on as a way to get portability between the Apache and IIS filters.

Having established that the filter shouldn't just get out of the way, the next job is to see if a session cookie is supplied. The name of the session cookie is not part of the "public" behavior and you cannot depend on it across Shibboleth versions. The name is currently a computed value produced by hashing various configuration settings. If the cookie is present, then processing proceeds with step 6 at the end of this walkthrough to determine if the session remains valid.

If not present (or if the session is determined to be invalid for some reason), then the request is considered to be "unauthenticated". This may be acceptable (see for example the LazySessions topic): the `requireSession` / `requireSessionWith` property pair determines this. If neither is active, then control will pass back to the server, and proceed.

Otherwise, much fun ensues. The ultimate goal of this step (when this point is reached) is for the SP software to issue an AuthnRequest on behalf of the application being accessed. The AuthnRequest has to find its way to an !IdP in step 2 below. To make all this happen flexibly, the SP contains SessionInitiators to perform this task.

SessionInitiators and IdPDiscovery

Once it is determined that a session is required, but doesn't exist, control is passed by the filter to a SessionInitiator to decide what to do. This happens as a kind of "internal redirect". There can be multiple SessionInitiators defined for a ShibbolethApplication. The application itself is determined based on the `applicationId` setting returned for the request by the RequestMap.

This is a recurring theme...any time you need to know what the SP is going to do, look at the URL and determine the `applicationId` using the RequestMap to figure out which <Application> element will be used to find the overall configuration settings in effect.

If the `requireSessionWith` property is set for the request, then the value of the property is used to look up the SessionInitiator to use (based on a matching `id` attribute). If not, the default SessionInitiator will be used.

Using multiple SessionInitiators is generally used to support referrals to multiple DiscoveryServices or IdentityProviders based on the request URL. When invoked in this automated way, the SessionInitiator only has enough information to refer the browser to the location in the `wayfURL` attribute, using the protocol laid out by the `wayfBinding` attribute. What this means in practice today is that the SP will create an AuthnRequest using the specified protocol (usually Shibboleth's own) and send it to a WAYF for referral. If the URL of that WAYF is actually the SingleSignOnService of an !IdP, then this will simply be a direct referral of the user to that !IdP.

> ***NOTE:*** *This general area of the software is likely to undergo significant refinement and enhancement in future versions.*

AuthnRequests and RelayState

The specific steps that happen in this final stage depend on the AuthnRequest protocol chosen. This in turn depends on the value of the `wayfBinding` attribute in the SessionInitiator selected. In the supported protocols today, the software needs to know three basic things to create the request:

- the original resource requested
- the location of the AssertionConsumerService to use for the eventual response
- the unique name of the SP

The name of the SP comes from the `providerId` attribute in the applicable `<Application>` element. In this scenario, the location of the AssertionConsumerService is defaulted (either the first one specified, or the one with =isDefault="true"=)

The resource, of course, is actually the active URL for the request so is already know. Depending on the value of the `localRelayState` attribute, this URL is either sent along with the AuthnRequest (a privacy leak) or stashed locally in a cookie that is sent to the client along with the AuthnRequest redirect.

Finally, the HTTP response containing the AuthnRequest in the redirect URL is returned to the browser and this first stage is done.

IdPDiscovery

Depending on how the software is configured, the AuthnRequest redirect issued above is either directly to an !IdP or to a DiscoveryService of some sort to figure out which !IdP to use. Eventually, magic happens, and the request can finally be sent along to the SingleSignOnService for step two.

%COMMENT%