NativeSPBackDoor

The V2.5 release of the SP includes a pair of mechanisms that support integration with authentication mechanisms that are outside the SP itself. You can think of this as the reverse of the approach described here under "Provisioning the Application's Database or Session".

Instead of transitioning from the SP session to something else, the SP can now support transitioning from "something else" into an SP session. The external mechanism can be anything, including local authentication by password, or other SSO protocols. Technically it doesn't even have to live on the same server as the SP, because this mechanism is in principle a form of SSO in its own right.

Obviously, there are security implications to using these features, and you should understand them well before enabling them.

SAML Artifact Spoofing

One of the supported mechanisms takes advantage of the less-commonly used SAML binding called the "Artifact Binding". Instead of pushing a SAML response through the client, an encoded reference called an "artifact" is generated by the IdP, passed to the SP via a redirect, and then the SP turns the artifact into the SAML response using a SOAP request.

This support has been extended to support "file-based" resolution of artifacts. In this approach, the local file system is used as a "secure transport" for turning the artifact into a SAML response. It is assumed that the file system between the external mechanism and the **shibd** daemon is secure, and the artifact is expected to be generated in a secure manner so that valid artifacts cannot be predicted by an attacker. Its security properties are essentially similar to standard artifact usage in SAML.

Some things to keep in mind:

- The external mechanism is in some sense like a SAML IdP in that it is generating SAML responses and assertions, generating artifacts, and passing them through the client using the SAML HTTP-Artifact binding.
- The mechanism relies on hard-to-predict artifact values, so a good PRNG is required.
- The SAML response and assertion do not need to be digitally signed, because the file system is assumed to be a secure exchange medium. This greatly simplifies the work of the external mechanism in mocking up a SAML response to use.
- Except for the signature, the rest of the SAML message and assertion(s) are processed identically to any other response generated by an IdP, and will be passed through all of the usual security checks. This means a lot of extra "boilerplate" SAML needs to be generated, including unique message and assertion IDs and good timestamps. It's simple to do, but a little tedious.
- Artifacts are an encoded binary data structure, so they may be more complex for some programming languages to generate.

Technical Specs

So, what do you actually have to do?

Generate the SAML

First, the external mechanism needs to generate a `<samlp:ArtifactResponse>` message that will be used to generate the user's session. This is a weird animal because it's very layered. It's a SAML assertion inside a SAML response inside a SAML artifact reponse. Two layers of protocol. An example follows.

```
<saml2p:ArtifactResponse xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
    ID="_0aedefc7f62695c068174a4b5362e0a1" IssueInstant="2012-04-17T17:07:01.260Z" Version="2.0">
    <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"
        >https://idp.example.org/idp/shibboleth</saml2:Issuer>
    <saml2p:Status>
        <saml2p:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
    </saml2p:Status>
    <saml2p:Response ID="_bdcdf2fdfc9564976ebf2d5fc68ca2ff" IssueInstant="2012-04-17T17:07:01.120Z"
        Version="2.0">
        <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"
            >https://idp.example.org/idp/shibboleth</saml2:Issuer>
        <saml2p:Status>
            <saml2p:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
        </saml2p:Status>
        <saml2:Assertion ID="_80a8853dedd373976a1a39f5fd7231fd" IssueInstant="2012-04-17T17:07:01.120Z"
            Version="2.0">
            <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"
                >https://idp.example.org/idp/shibboleth</saml2:Issuer>
            <saml2:Subject>
                <saml2:NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"
                    NameQualifier="https://idp.example.org/idp/shibboleth"
                    SPNameQualifier="https://sp.example.org/shibboleth"
                   >O2S5XNIZEEF7LG7OKYUDGEIO7NBNWMPMST2A4T6NJZPPSH</saml2:NameID>
                <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
                    <saml2:SubjectConfirmationData Address="164.107.160.193"
                        InResponseTo="_9daf8aa18f2b25c822b22c26e8d9a431"
                        NotOnOrAfter="2012-04-17T17:12:01.120Z"
                        Recipient="https://sp.example.org/Shibboleth.sso/SAML2/Artifact"/>
                </saml2:SubjectConfirmation>
            </saml2:Subject>
            <saml2:Conditions NotBefore="2012-04-17T17:07:01.120Z"
                NotOnOrAfter="2012-04-17T17:12:01.120Z">
                <saml2:AudienceRestriction>
                    <saml2:Audience>https://sp.example.org/shibboleth</saml2:Audience>
                </saml2:AudienceRestriction>
            </saml2:Conditions>
            <saml2:AuthnStatement AuthnInstant="2012-04-17T15:19:11.781Z"
                SessionIndex="6c7fb0b96a3450a759e39eda61fc527abd662e1291d55edc8b6b2dcf241092bd">
                <saml2:SubjectLocality Address="164.107.160.193"/>
                <saml2:AuthnContext>
                    <saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:
PasswordProtectedTransport</saml2:AuthnContextClassRef>
                </saml2:AuthnContext>
            </saml2:AuthnStatement>
            <saml2:AttributeStatement>
                <saml2:Attribute FriendlyName="eduPersonPrincipalName"
                    Name="urn:oid:1.3.6.1.4.1.5923.1.1.1.6"
                    NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                    <saml2:AttributeValue>doe@example.org</saml2:AttributeValue>
                </saml2:Attribute>
                <saml2:Attribute FriendlyName="displayName" Name="urn:oid:2.16.840.1.113730.3.1.241"
                    NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                    <saml2:AttributeValue>John Doe</saml2:AttributeValue>
                </saml2:Attribute>
            </saml2:AttributeStatement>
        </saml2:Assertion>
    </saml2p:Response>
</saml2p:ArtifactResponse>
```

The message ID and IssueInstant in the very outer message are not evaluated by the SP, but everything else is, just as if it were a real IdP. That means the values must be unique and fresh, and it means that you have to load metadata corresponding to the "issuer" value you use, just as if it were a real SAML 2.0 IdP.

This is required anyway, because of steps two and three, generating an artifact and storing the XML above in a file.

Generate the Artifact

The SAML artifact format is a binary structure that you have to construct carefully, and then base64-encode to produce the value the SP receives. The structure is as follows (zero offset):

Bytes	Field
0-1	Artifact Type, this MUST be 0x04
2-3	Endpoint Index
4-23	SHA-1 Digest of IdP's entityID
24-43	Randomly generated message handle

The first field is hardcoded.

The second field is a 16-bit unsigned integer that is a reference to the `index` attribute in the `<md:ArtifactResolutionService>` element in the IdP's metadata. In the case of this back-door trick, you can define multiple file system locations that will be used to pass these messages in, and use the index field to identify which one is used.

The third field is how the SP figures out the IdP sending it the artifact. This is just a hash of the entityID.

The fourth field is the unique part. This **MUST** be strongly random. Taking the SHA-1 hash of suitably random input is one way of generating the 20 bytes needed.

Once it's all together, you base64-encode the 44 bytes of data and that's the artifact.

Storing the SAML

Once you have the artifact, you take the last 20 bytes (the message handle), and you hex-encode them into a 40-byte string (lower case a-f). That's your filename. The XML you generate is written to a file with that name, stored in whatever location you're using to pass the data to the SP (see last step).

Trigger the Login

To make the SP consume the file, you generate a redirect to the SP's artifact-eating endpoint, usually `/Shibboleth.sso/SAML2/Artifact`, with a query string parameter of `SAMLart` set to the base64-encoded artifact. You can also pass a `RelayState` parameter with a URL to land on, or a value obtained from an initial interaction with the SP. This is standard SAML 2.0.

Finding the File

The SP finds the file that you created earlier by looking up the directory to use in the IdP's SAML metadata. The endpoint index in the artifact must correspond to an `<md:ArtifactResolutionService>` endpoint that contains a `Binding` attribute set to `urn:mace:shibboleth:2.0:bindings:File` and a `Location` containing a relative or absolute filesystem directory path. It can start with "file://", or not. If the path is relative, the SP's default "runtime file" location will be prepended (e.g., **/var/run/shibboleth**).

In order for this mechanism to be supported, you must also turn on the `artifactByFilesystem` property for the application and/or relying party. This allows you to selectively enable the feature for only the application, or specific IdP, involved.

Example Endpoint in Metadata

```
<ArtifactResolutionService index="1" Binding="urn:mace:shibboleth:2.0:bindings:File"
    Location="file:///opt/shibboleth-sp/var/artifacts" />
```

External Authentication Handler

The other supported mechanism is a "loopback" handler designed to be invoked via HTTP by a server-side integration script that generates appropriate inputs for a user session based on its own authentication mechanism. In other words, a server-side piece of code does "something" to authenticate the browser, and based on that information it generates a submission to the SP that it sends directly to this handler (so, server to server), telling it to provision a user session. The handler evaluates the input and assuming it works, a session is generated and data is returned by the handler so that the integration code can set cookies and such in the client and redirect the client to resources protected by the SP.

For reference documentation, see the NativeSPHandler topic, but the basic syntax to enable the handler for access by localhost is simply:

```
<Handler type="ExternalAuth" Location="/ExternalAuth" />
```

Some things to keep in mind:

- Because the creation of the session is left entirely up to the calling code, the SP performs essentially no security checks on the information and trusts the calling code implicitly.

- The calling code usually has to run on the same virtual host as the handler, because the handler will generally be returning one or more cookies for the calling code to set on the original client browser. Those cookies are meant for the SP, which means they can only be set by resources running on the same vhost, unless the cookie domain is expanded.
- The only security mechanism supported between the calling code and the handler is IP address checking. If non-loopback addresses are allowed, then the network segment in between needs to be secured somehow.

Technical Specs

The specific protocol supported between the external integration script and the handler are as follows.

Input Format and Processing Model

Two input formats to the SP handler are supported, XML/SAML and URL-encoded Form POST. Both use the "POST" HTTP method.

To use the XML option, the HTTP request must contain a `Content-Type` header set to either `text/xml` or `application/xml+samlassertion`. The body of the request must contain a SAML 2.0 `<Assertion>` element to use to provision the user's session. The URL MAY contain a query string parameter called `RelayState` that conforms to a relay state mechanism supported by the SP. Usually this will be something generated earlier by the SP, or an actual resource URL.

Most of the assertion is essentially ignored. The meat of the assertion that is processed are the `<NameID>` element in the subject, the first `<AuthnStatement>` element, and all `<AttributeStatement>` elements. No security checks are performed. The `<Issuer>` element is used to attempt to locate SAML 2.0 metadata for an IdP, and if that's found, that entity is considered the "source" of the session (for logging, policy, etc.). If not, no such issuer is associated with the session.

Of particular note, the client's actual IP address should be placed into the usual spot in the assertion, inside the authentication statement in the `<SubjectLocality>` element's `Address` attribute.

To use the Form POST option, the HTTP request must contain a `Content-Type` header set to `application/x-www-form-urlencoded`. The set of parameters supported are as follows:

- `protocol`
  - A protocol identifier used for logging purposes, and to acquire metadata for the `issuer`.
- `issuer`
  - (optional) An entityID or similar unique identifier for the authentication source. If present, used to look up metadata. If not, no issuer is associated with the session.
- `address`
  - The client's IP address.
- `NameID`
  - Used as the primary subject identifier for the session.
- `Format`
  - Describes the format of the `NameID` value. If omitted, the SAML constant for an "unspecified" format is applied.
- `AuthnInstant`
  - Time of authentication, in UTC ISO date/time format. Example: `2013-07-02T17:09:21Z`
- `SessionIndex`
  - An identifier for the session that allows for unique identification of a particular session among all others associated with the `NameID`.
- `AuthnContextClassRef`
  - Associates a SAML authentication context class with the session.
- `AuthnContextDeclRef`
  - Associates a SAML authentication context declaration with the session.
- `lifetime`
  - The intended lifetime for the session in seconds, if other than the default.
- `attributes`
  - Comma-delimited list of attribute names to look for in the rest of the parameter set. Each value associated with the named parameter is used as the value for a simple attribute with the name specified. For example, if the value of `attributes` is `"uid,displayName"`, then form parameters called `uid` and `displayName` are read.

A requirement of using the form post option is that the attribute names used **MUST** correspond to attribute IDs specified in the **attribute-map.xml** file or other attribute extractor plugins. This is to ensure that the SP defends against header spoofing properly, because it relies on those plugins to self-identify the possible set of attribute names to protect. Simply put, don't invent new attribute names to populate; use existing attribute names that are already in use for SAML-based sessions, creating new mappings if necessary.

Attributes placed into the ExternalAuth POST will be subjected to attribute policy as defined in the `attribute-policy.xml` file. This could be particularly significant if you define an attribute whose policy enforcement rests upon metadata properties. For example, any scoped attribute (such as `eppn` or `affiliation`) may incorporate the `ScopingRules` rule, which is predicated on a metadata definition of scope, referenced by the `issuer`'s entityID. In this case you may need to adjust policy to accomodate arbitrary scope on these attributes.

Output Format

If an error occurs, the SP handler will return an HTTP error status, typically 500, but in any case, not 200.

If successful, the HTTP response body will contain a result that allows the calling code to provision the session into the client and finish the process. The result can be in either XML or JSON format. By default, XML is used. If the calling code supplies an `Accept` header in its request of `application/json`, then the result will be JSON.

The XML format consists of a root element called `<ExternalAuth>` that contains the following child elements:

- `<SessionID>`(required)
  - The session key issued (mostly for information only).
- `<Cookie>`(zero or more)
  - Each value MUST be returned to the original client in a `Set-Cookie` response header in order to provision the session into the client and perform any cleanup expected by the SP.
- `<RelayState>`(optional)
  - If present, the value will be a URL to send the client to via redirect when done.

The JSON format is an anonymous structure containing the following members:

- "SessionID" (required)
  - The session key issued (mostly for information only)
- "Cookies" (optional)
  - If present, a list of strings. Each value MUST be returned to the original client in a `Set-Cookie` response header in order to provision the session into the client and perform any cleanup expected by the SP.
- "RelayState" (optional)
  - If present, the value will be a URL to send the client to via redirect when done.

Completion

The responsibilities of the calling code are outlined above in the XML and JSON cases. The SP will return a set of cookie headers to set, and supply a URL to redirect to (or leave that decision up the caller).