AttendedRestartConfiguration

⚠ This feature is available in V3.4 and later of the software.

Current File(s): *conf/admin/general-admin.xml, conf/admin/unlock-keys.xml, views/admin/unlock-keys.vm*

Format: Native Spring, Velocity

Overview

V3.4 introduces a feature called "attended restart", which means that by design the IdP is designed to be unusable if it restarts without human intervention. Most systems obviously focus on the opposite of that characteristic. The reason behind this feature is key protection. There aren't a lot of cost-effective, simple solutions to protecting private and secret keys (there are complex ones, expensive ones, and complex, expensive ones, of course). The concept behind this feature is that it allows keys to be encrypted on disk, but without the password stored anywhere with the key and entered manually when the service is restarted.

Background

An issue with the IdP is that it is a huge focal point for risk, and the way that SAML is typically used focuses on deployability and not on security, namely the use of a pushed, signed token through the client whose only security rests on the protection of the signing key. With no back-channel step to add additional hurdles for an attacker, everything depends on the key.

If all SPs ran Shibboleth or SimpleSAML.php, this would be a reasonable thing to deal with by simply rotating the signing key through metadata on a semi-regular basis. Unfortunately virtually nothing else out there has any story for handling either key revocation or rotation, and if your auditors find out about this, you may be facing some difficult conversations that start with questions like how many administrators of your backup systems have access to the private key.

My (Scott's) solution to this problem was to check out how much Amazon's HMS solution costs, vomit, and then start looking into options. One of them would be some kind of "vault" solution to house the key, but that ultimately just moves the problem and potentially adds a different set of staff with access to the key. One of the things I thought about was how many times in the 20+ year history I've run an IdP that I had to rely on a node restarting without me asking it to, and the answer was zero. It also helps that I don't care about Docker or other technologies that solve deployment problems I don't have.

So my choice was to investigate what it would take to get the IdP to start with both the private and secret key(s) locked or unconfigured, and change that state at runtime. Somewhat surprisingly, this not only mostly worked anyway, but in fact the IdP was happy to start up with no private key and just issue unsigned responses, which while harmless (and perhaps a nice testing feature), seemed a bit much, so that's also been locked down a bit.

Overview

This feature is a bit unusual so it's not just a "turn on setting" sort of thing, but something you have to make a few adjustments to the configuration to use. There are two parts to using this:

- Adjust the private and secret key settings to defer availability.
- Enable the unlock-keys webflow to get them installed after startup.

There are two significant risk points, the private key(s) used to sign messages and the secret key(s) used to support client-side session storage. Not every system uses the latter, but client-side storage is the default, so it's typically a factor. There are also private key(s) used to decrypt XML, but the impact of those keys is much less significant, and they don't get much use. Nevertheless, the feature supports unlocking private keys regardless of purpose so if you wanted to leave the decryption keys locked, that's certainly possible.

So how do you do it?

Credential Changes

Normally the configuration of keys and certificates for signing and encryption is found in *conf/credentials.xml*. To use this feature, it's necessary to move the critical beans defining these credentials out of this file and up into a globally visible place. The recommendation is to create a file called *conf/admin /unlock-keys.xml* and move the bean(s) you want to "defer" to that file (with a couple of minor adjustments).

You can copy all the Spring boilerplate from *conf/global.xml* and/or follow examples below, but the idea is to move the signing and/or encryption beans to this file, and then simply remove the lines referring to the private keys. You simply omit that part, and any such credential will initialize itself with only the certificate "half" in place, but no private key.

Examples are included below.

DataSealer Changes

The secret key component, if it's in use, is normally configured in *idp.properties*. The change needed here is simply to eliminate the passwords that unlock the keystore and key entry, normally these properties:

idp.properties

```
idp.sealer.storePassword = password
idp.sealer.keyPassword = password
```

If you remove these lines, or comment them out (prefix with a # character), that automatically kicks the underlying secret key source into a "delayed init" mode that allows it to be unlocked at runtime.

Conditions at Startup

So what happens if you do all this and then start the Java servlet container? By default, it's nothing really noticeable unless you try to login to an SP. The lack of a secret key causes some loud warnings in the log any time the software tries to load or store session data to the client, but it doesn't actually cause a request to fail. The lack of a signing key, however, will cause SAML requests to terminate with an error because of the inability to sign messages. This is not ideal, but it's assumed that if you're using this feature you're aware of what you're doing and you should know that you need to unlock the system before putting it into use.

Unlock Flow

To use this feature, a new administrative flow is provided that has a crude user interface to collect password(s) via a web form and manipulate the system's internals to unlock and install the keys necessary to put the system into a normal state as if the feature hadn't been used. From that point on, the system should behave normally until the next time it restarts. If the flow believes that it's done this successfully, it records that fact so that if it runs again it simply skips these steps. If it detects a failure, it leaves any remaining work undone and redisplays the form, and the log should usually indicate what didn't work.

To get this working, the flow has to be defined and enabled in *conf/admin/general-admin.xml,* and some Spring beans defined to describe to the flow what it needs to unlock.

Enable the Flow

The following is added to (or uncommented in) the **shibboleth.AvailableAdminFlows** bean:

Added to conf/admin/general-admin.xml

```
<bean parent="shibboleth.OneTimeAdminFlow"
        c:id="http://shibboleth.net/ns/profiles/unlock-keys"
    p:loggingId="UnlockKeys"
        p:authenticated="true"
        p:policyName="AccessByAdminUser" />
```

The last couple of properties are local. This example presumes that the rule for accessing the flow is that the user must login first, and that a map entry will be defined in *conf/access-control.xml* keyed under "AccessByAdminUser" that defines which usernames can access the flow. The access control features are described under AccessControlConfiguration.

There's total flexibility on this, it's up to you to define the rules. You can even set properties that typically are used in *relying-party.xml* like `defaultAuthenticationMethods` to control what kind of authentication has to be done (e.g. requiring MFA). Authentication in general won't rely on the keys being unlocked here, so there won't usually be any circular dependency there.

Configuring the Flow

It's possible to define everything necessary in *global.xml,* but the system reserves the file *conf/admin/unlock-keys.xml* for doing this and will load it if it's present.

In addition to populating it with any "credential" beans whose private keys will be left out, this is where you define the information the flow needs so it knows what to do, using these beans:

- **shibboleth.unlock-keys.KeyStrategies**
- **shibboleth.unlock-keys.Credentials**

- **shibboleth.unlock-keys.PrivateKeys**

All of these beans are ordered collections.

The KeyStrategies collection contains bean references to the DataSealer key strategy objects that you're going to unlock. Typically this is just a collection of one reference to **shibboleth.DataSealerKeyStrategy**, which is the name of the system bean that is included by default to support the secret key features of the software. Unless you're doing something unusual and have created your own objects for some reason, that's all you'd need to include. Of course, if you don't actually want to leave that keystore locked, you don't have to specify it.

The Credentials collection is where you enumerate the X509Credential beans that you move out of *conf/credentials.xml* and from which you chop out the private key property.

The PrivateKeys collection is where you provide pointers to the private keys to unlock and inject into the Credentials objects. These generally need to be turned into explicit Spring FileSystemResource beans to avoid problems with paths being misinterpreted as relative to the wrong location.

These two collections have to "line up" such that the position of the private key in one list is meant for the Credential in the same position in the other list.

Configuring the Form

This is a fairly low tech feature; based on the collections here, you may need to then modify the view template in views/admin/unlock-keys.vm and add or remove sets of form fields with specific names to collect the passwords you need. All of them are collected at once, and the template includes comments noting what you have to do. The flow defines the form fields "keystorePassword", "keyPassword", and "privateKeyPassword" to carry the passwords of each type, and multiple copies of the fields can be defined and should be processed in order by the flow based on the order of the objects in the collections described earlier.

You are expected to define the right number of fields and give them appropriate labels for your own use.

The template also illustrates a useful idea of embedding a SSO push link that can be used at the end to verify that the unlocked IdP works correctly (though you'd have to look at the log to know for certain the secret key is working).

Example

In a typical example, the following assumes you want to unlock both the system-supplied secret keystore and the default signing key:

Example conf/admin/unlock-keys.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util" xmlns:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
/spring-beans.xsd
                        http://www.springframework.org/schema/context http://www.springframework.org/schema
/context/spring-context.xsd
                        http://www.springframework.org/schema/util http://www.springframework.org/schema/util
/spring-util.xsd"

    default-init-method="initialize"
    default-destroy-method="destroy">

    <!-- Attended restart unlock beans. -->

    <!-- Enumerate the system key strategy beans to unlock, typically just one. -->

    <util:list id="shibboleth.unlock-keys.KeyStrategies">
        <ref bean="shibboleth.DataSealerKeyStrategy" />
    </util:list>

    <!-- Enumerate credential bean refs and private key resources. -->

    <util:list id="shibboleth.unlock-keys.Credentials">
        <ref bean="shibboleth.DefaultSigningCredential" />
    </util:list>

    <util:list id="shibboleth.unlock-keys.PrivateKeys">
                <bean class="org.springframework.core.io.FileSystemResource"
            c:_0="%{idp.signing.key}"/>
    </util:list>

    <bean id="shibboleth.DefaultSigningCredential"
        class="net.shibboleth.idp.profile.spring.factory.BasicX509CredentialFactoryBean"
        p:certificateResource="%{idp.signing.cert}"
        p:entityId="%{idp.entityID}" />

</beans>
```

As you can see, mostly this is a lot of boilerplate, and some cut and paste out of *conf/credentials.xml* that reuses the same properties typically used to define the paths to the key and certificate, entityID, etc. The use of a FileSystemResource class bean is usually required.

Because the existing signing Credential object is already assumed to be named **shibboleth.DefaultSigningCredential,** transplanting that with minor changes is possible. If you were defining additional keys yourself those names would be different.

If you were going to transplant the encryption/decryption key(s), you would need to give them a bean name (via an `id` attribute), move them here, and then reference them in the list in *credentials.xml,* via the `<ref bean="id"/>` syntax. In practice it's probably not that necessary to worry about those keys as they're hardly ever used, and much less critical to the security of the IdP.