ENCRYPTING AND DECRYPTING CONFIGURATION PROPERTY VALUES IN SPRING CLOUD

As a follow up to my recent post on Microservices with Spring Cloud, this post will cover encrypting and decrypting config property values. The Spring Cloud docs cover this aspect briefly, so I'll go through the steps I took to get this working both in test on my laptop, and on a dev server. I'll be using the same environment as my last blog post, so that is a standalone Eureka server, a config server, and simple microservice. Full disclosure, I am by no means a security expert, so if you see something wrong, feel free to let me know.

INSTALL JCE

Head on over to Oracle's website and download the unlimited strength JCE. Once downloaded, you'll need to extract the zip file somewhere and the result will be three files.

```
local_policy.jar
README.txt
US_export_policy.jar
```

You need to copy the local_policy.jar and US_export_policy.jar into $JAVA_HOME/jre/lib/security directory. Before doing this though, you should make a backup copy of the existing policy files. I'm running on OSX, so for me it's as easy as this. The same applies to doing this on a Linux server, as is the case for me.

```
sudo mv US_export_policy.jar US_expo
sudo mv local_policy.jar local_polic
```

Now you can safely copy the JCE policy files you downloaded into $JAVA_HOME/jre/lib/security.

CREATE AN ENCRYPTION KEY OR KEYSTORE

Depending on your needs, you can create a symmetric (shared) or asymmetric (RSA) key. The Spring Cloud docs point out that the symmetric key is convenient as you merely need to provide a secret key to the environment, but that comes at the expense of being less secure. So for a symmetric key, either add an *encrypt.key* property to application.yml with a secret key, or add an environment variable *ENCRYPT_KEY*. For this blog post, I'll focus on the asymmetric key approach which uses a keystore file.

The JDK comes with the *keytool* utility which allows you to create a keystore. This example is copied directly from the Spring Cloud docs. So you should use a different password, alias and secret key.

```
keytool -genkeypair -alias mytestkey
  -dname "CN=Web Server,OU=Unit,O=Or
  -keypass changeme -keystore server
```

> ***TIP!*** *- By default, this command will create a key that is valid for only 90 days. If you want something longer than that, you'll*

> *need to add the validity argument which takes a value in days.*

```
keytool -genkeypair -alias mytestkey
  -dname "CN=Web Server,OU=Unit,O=Or
  -keypass changeme -keystore server
  -validity 365
```

The simple approach to using this new keystore from your config server would be to include the password, alias and secret key in application.yml.

```
encrypt:
  key-store:
    location: file://${user.home}/se
    password: letmein
    alias: mytestkey
    secret: changeme
```

A better approach would be to use environment variables. The location property is probably safe in the application.yml file, but you could also control it with an environment variable.

```
ENCRYPT_KEY_STORE_PASSWORD=letmein
ENCRYPT_KEY_STORE_ALIAS=mytestkey
ENCRYPT_KEY_STORE_SECRET=changeme</p
```

LET'S ENCRYPT!

Now that we have a config server that's able to encrypt and decrypt data, let's start by encrypting a property value. The config server exposes a couple of handy endpoints. There's the */encrypt* endpoint to encrypt data, and the */decrypt* endpoint to decrypt data.

```
curl localhost:8888/encrypt -d 'Hell
```

The result of this command will be the *'Hello Spring Boot!'* string as an encrypted string that we can add to the git repository backing the config server. Here's the application.yml file served by my config server, but now updated with the encrypted message.

```
message: '{cipher}AQCLJG5FvxQrLAC6Q9
```

Notice the *{cipher}* prefix. This is a hint to the config server that this property needs to be decrypted before sending it to a client application.

LET'S DECRYPT!

The Spring Cloud docs for Brixton.M5 mention that any client needs to include *spring-security-rsa* on it's classpath in order to decrypt the encrypted properties.

```
<dependency>
    <groupId>org.springframework.sec
    <artifactId>spring-security-rsa<
</dependency>
```

If you're application connects to Eureka to get it's configuration from a remote config server, then the encrypted properties will be returned to your application already decrypted during the bootstrap phase of your application, before the main application context is initialized. So this dependency is not needed in your application.

This has been my own observation without seeing anything explicitly mentioned in the

docs. I believe this dependency is only required if your application connects to the config server directly, if you've included *spring-cloud-starter-config* as a dependency.

USE DECRYPTED PROPERTIES

The following is a simple message service. The only real difference is the *@RefreshScope* annotation on the service. I covered @RefreshScope in my previous post. To reiterate, this annotation will create a proxy that caches the configuration properties in the environment. When this application's context is refreshed, the cache will be invalidated and repopulated with the most current configuration properties, making those new values available to the application. In this case, the value of *message* will be the decrypted string *'Hello Spring Boot!'*

```
@Service
@RefreshScope
public class MessageServiceImpl impl

    private Environment env;

    @Inject
    public MessageServiceImpl(Enviro
        this.env = env;
```

```
    }

    public String getMessage() {
        return env.getProperty("mess
    }
}
```

CONGRATULATIONS!

You now have a config server that's able to decrypt encrypted property values. The real benefit to this is that you can centralize sensitive information that multiple services can consume and you don't need to configure these values in every application that needs them.