Susinda Perera Follow

May 8, 2020 · 5 min read · Listen

Securing Spring Cloud Config Server

When it comes to configurations management in Microservice architectures Spring Cloud Config Server is the easiest and widely used solution for spring based Microservice implementations. Although there are a lot of tech blogs on configuring this for its basic use, very little information is available on securing the server for production level deployment. In this article, my focus is to bring together the concepts and configurations that we could use to tighten the security of Spring Cloud Config Server.

Spring Cloud Config Serve provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments[1]. Following are some of the features which make it the best candidate for configuration management.

- HTTP, resource-based API for external configuration (name-value pairs)
- Encrypt and decrypt property values (symmetric or asymmetric)
- Embeddable easily in a Spring Boot applications with annotations
- Git as the configuration source (supports versioning)
- Support of refresh properties at runtime

It is extremely important to keep the configuration server secured as it might hold sensitive configuration information such as access credentials to other applications and services. Hence leaving the security unattended here, will open the doors to every other system and could end up in an irreparable damage. Following are some simple but powerful concepts that help to tighten the security of the configuration server.

- Apply authentication on configuration server
- Always use secured channels when configurations are in transit
- Use secured, private repository as the configuration source
- Never store passwords in plain text
- Disable the decrypting at the cloud config server.

Apply authentication on configuration server

The Spring Cloud Config Server (lets say SCCS from here onwards) support basic authentication out of the box.To enable basic authentication for the SCCS configure the following properties in the application.properties file.

```
spring.security.user.name=usernameofyourchoice
spring.security.user.password=passwordofyourchoice
```

Basic auth might not be the best option, but it is the default mechanism provided by config server and would be the easiest to implement for both config server and clients connecting to it. Additionally its best practise to add physical network level security. Once the server is configured with basic auth, the Microservice (clients) connecting to the config server need to be configured accordingly. In clients following need to be specified in the bootstrap.properties file as follows

```
spring.cloud.config.username=usernameofyourchoice
spring.cloud.config.password=passwordofyourchoice
```

Make sure to choose a strong password which cannot be easily guessed. Also it would be better not to hardcode this password in the application.properties file in both clients and server, instead use a build variable and inject it in the build time using a build script. For example spring application properties support environment variables in the following syntax.

```
spring.security.user.password=${CONFIGSERVER_PASSWORD}
```

Always use secured channels when configurations are in transit

Having basic authentication enabled, would not help if an unencrypted channel is used, as a middle man can eavesdrop the username and password being posted. Therefore having a secured channel (in this case HTTPS) is a must when you implement basic auth. The same applies for configuration data, as sensitive configuration data should never traverse the network in an unencrypted channel. To enable SSL apply the following configurations in the application.properties file.

```
server.port=8443

server.ssl.key-store=classpath:ssl-server.jks
#ssl-server.jks file needs to be in the src/main/resources directory

server.ssl.key-store-provider=SUN

server.ssl.key-store-type=JKS

server.ssl.key-alias=selfsigned_localhost_sslserver

server.ssl.key-password=changeit
```

For testing purposes, a self sign key can be used, one can be generated using the command below

```
keytool -genkey -alias selfsigned_localhost_sslserver -keyalg RSA -
keysize 2048 -validity 700 -keypass changeit -storepass changeit -
keystore ssl-server.jks
```

If a self sign certificate is used, the clients should have the public cert of self sign certificate used by the server installed to the JVMs trust-store to avoid SLL handshake issues. The commands below can be used to import the public cert to java cacerts file.

```
keytool -export -alias selfsigned_localhost_sslserver -keystore ssl-server.jks -file configserver.pem

keytool -import -storepass “changeit” -keystore “/PATH/TO/JRE/lib/security/cacerts” -alias configserverselfsigned -file “./configserver.pem”
```

Use secured, private repository as the configuration source

Spring cloud config server supports few backends as the configuration source. As git is the widely used one, I will be focusing on git here. Git repos can be secured with basic auth or ssh key based authentication. Spring cloud config server supports both of these modes.

Username/password authentication

```
spring.cloud.config.server.git.uri=https://susinda.github.com

spring.cloud.config.server.git.username=yourusername

spring.cloud.config.server.git.password=yourpassowrd
```

SSH-Key based authentication

```
spring.cloud.config.server.git.strictHostKeyChecking=false

spring.cloud.config.server.git.uri=https://susinda.github.com

spring.cloud.config.server.git.ignore-local-ssh-settings=true

spring.cloud.config.server.git.private-key= — — -BEGIN RSA PRIVATE KEY — — -\nyourprivatekeyline1\nline2goeshere==\n — — -END RSA PRIVATE KEY — — -
```

Never store passwords in plain text

The SCCS supports handling encrypted properties and further provides REST endpoints for encryption and decryption. The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is just a single property value to configure. To enable encryption, use following configuration in both config-server and connecting clients in their bootstrap.properties file.

```
encrypt.key=mysecretkey
```

The encryption endpoint can be invoked as follows and it will return the encrypted value.

```
curl -X POST -k ‘https://localhost:8443/encrypt' – header ‘Content-
Type: application/x-www-form-urlencoded’ – header  ‘Authorization:
Basic cm9vdDpzM2NyM3Q=’ \ – data-raw ‘myPasswordHEre’
#  above  will  return  something  like  this
24f800bf048d9f4341d1d12ed0972bddbac84a7b3d0dbe33ff031e376dc8ef12
```

The encrypted configuration is stored in the following format in the properties files. For example myservice-dev.properties file look like below.

```
my.test.server.password=
{cipher}24f800bf048d9f4341d1d12ed0972bddbac84a7b3d0dbe33ff031e376dc8ef
12
```

Similarly , the decryption endpoint can be invoked as follows and it will return the plain text value.

```
curl -X POST -k ‘https://localhost:8443/decrypt' – header ‘Content-
Type: application/x-www-form-urlencoded’ – header  ‘Authorization:
```

```
Basic cm9vdDpzM2NyM3Q=’ \ – data-raw
‘0259096f4748c2423df982311dab5f73e1194edc2a866fc77abd05e828a5597e’
```

Disable the decrypting at the cloud config server.

For additional security we can configure the server to not to decrypt the configuration but only decrypt at the client side. In servers bootstrap.properties file add the following to configure the same.

```
encrypt.key=mysecretkey
spring.cloud.config.server.encrypt.enabled=false
```

Further, we disable the decrypt endpoint from spring configuration server, such that even if one has access to the server could not get the plain text value of an encrypted property. Adding the following class to the project will do the magic.

```
@Configuration
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {
@Override
  public void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests().antMatchers(“/decrypt”).denyAll();
      super.configure(http);
  }
}
```

Summary

In micro — services architecture, configuration properties usually are the endpoint urls and credentials for the other connecting applications and services. When we decide to use a centralized config server, that server needs to be secured as it stores these sensitive information.

The basic means of security is to configure authentication to the server. So that only authenticated users (in this case micro — services)have access to the config server. Further the channel between micro — services and SCCS needs to be encrypted to avoid eavesdropping attacks, therefore SSL is configured. Further it is best to implement physical network security to only allow traffic from Microservices to this server as an additional measure.

In this article, git is used as the backend for storing actual configuration property files. In addition to using a private secured git repository, when storing sensitive config properties it should be stored encrypted. The SCCS provides support to decrypt while reading, but for better security we have disabled this feature and configure those to be decrypted at the client. Further we have disabled the decryption endpoint from config server, disabling the decryption ability from users who have access to the server.

The git backend credentials and config server credentials are best not to hard code in the application.properties files in both server and Microservice. A build server or build scripts can be used to inject these at the build time for better handling of these passwords.

Hope This Helps :)

References

[1] https://spring.io/projects/spring-cloud-config

Spring Cloud Config Security Git Spring Integration Microservices