CS 594—Spring 2021
Enterprise and Cloud Security
Assignment Two—Enterprise App Security

In this assignment, you will complete an enterprise app with some security mechanisms and deploy it in the cloud. Most of the code is complete, you will just have to fill in the missing code. Some of the code is missing to get you to read, and demonstrate understanding of, the code that handles authentication and authorization. Since this is not a course in Web programming, in general you are not responsible for the Web user interface (i.e., Java Server Faces), but you will be provided with some explanation to help you understand. For this assignment, you should use the latest version of Eclipse to edit and compile the project. You will also need to download Payara, not to run it but so Eclipse can compile against the runtime. You are also advised to install Docker Desktop on your machine, and develop running Payara and Postgresql locally. You will need to eventually deploy the app in your VPC from Assignment 1.

The app is a simple bulletin board. The projects that make up this assignment consist of:

- ChatApp: The umbrella enterprise application project.
- ChatDomain: The domain model for the app.
- ChatServiceClient, ChatService: The service oriented architecture (SOA) for the app.
- ChatDTOs: The data transfer objects (DTOs) for the SOA.
- ChatInit: Initializes the app during deployment.
- ChatWebApp: The Web interface for the application.
- ChatRoot: A Maven parent project for the modules above.

You will find the Java EE tutorial useful, particularly Part X on Java EE security:

https://javaee.github.io/tutorial/partsecurity.html#GIJRP

Where your changes involve modifying Java source, this is indicated by TODO items that you can find in the Markers tab in Eclipse.

The app uses a database of users and passwords to authenticate users. In addition, it supports two-factor authentication using Time-based One-Time Passwords (TOTP), as described in RFC 6238. This works by having a secret shared between a personal device and the user database. When a user logs in, a code is generated by the personal device, using this secret and the current time. The user authenticates only if their password is correct, and in addition the code that is provided matches the code generated by the app from the secret stored in the user record. In this case, the personal device will be a smartphone (either Android or iPhone) running the Google Authenticator app. When a user is added to the database, their TOTP details are displayed as a QR code, which is scanned by the personal device to initialize Google Authenticator. Once a user has been

authenticated, role-based access control (RBAC) is used to constrain what they are able to do with the app.

You will need to do the following with the app server to deploy the app:

1. You will need to set up the JDBC connection pools and resources in the application server for the database, as before. Use a resource name of `jdbc/cs594` for the JNDI name of the connection pool for the app database. You should provide a demo of the app running in VPC, including a view of the JDBC connection pool setup, and deployment and running of the app.
2. You will have to initialize the database with its tables. The persistence descriptor `persistence.xml` in `ChatDomain` is set up to automatically drop and recreate the database tables each the app is deployed. The scripts for dropping and creating the database are in `ChatDomain/resources`, copied into the jar file for the project by Maven, and executed automatically when the app is deployed. You will need to copy the `createDDL.sql` script into your VPC, first to the private instance, and then to the docker container, and then run `psql` to create the tables for the first time[1]:

   ```
   docker cp createDDL.sql \
       <container-id>:/docker-entrypoint-initdb.d/createDDL.sql
   docker exec -it <container-id> psql -U postgres -d cs594 -a \
       -f docker-entrypoint-initdb.d/createDDL.sql
   ```

   Here is what you need to do that concerns the security API for the app server:

1. In the Web app, a backing bean (`LoginBacking`) manages logging users in and out. You should complete the login operation, as well as the operation that checks if someone is currently logged in (This is used to control which parts of the login screen get rendered). You should use the security context, which is injected using dependency injection.

   A note on the login page: It is possible to define the authentication logic for form-based authentication declaratively, specifying a built-in Java function (`j_security_check`) for performing authentication, but it requires the use of the HTML `form` element instead of the JSF `h:form` element. In order to stay with a consistent use of JSF markup, and because of the need to specify a role when a user logs in and to check the TOTP, the login logic provided by the HTTP servlet is invoked programmatically in the backing bean, `LoginBacking.java`.

2. In the `ViewMessages` backing bean in the Web app, use the injected security context to look up the currently logged-in user, as part of initializing the backing bean. This user name is displayed in the Web pages for showing messages.

---

[1] Note that the `docker-entrypoint-initdb.d` directory has special significance for the Postgresql docker container.

3. In the `MessageService` bean in the `ChatService` project, you will need to complete parts of adding a user (hashing their password, adding roles and generating their OTP authorization), updating a user's information (hashing their password), and adding a message (making sure that the poster of a message is the currently logged in user). The initialization of this bean configures various parameters of the password hashing algorithm, you will need to make sure that the built-in authentication mechanism uses the same configuration.

4. You will configure authentication in the `AppConfig` class in the Web app. Use the `@CustomFormAuthenticationMechanismDefinition` annotation to configure forms-based authentication with the appropriate login page. Use the `@DatabaseIdentityStoreDefinition` to configure the use of a backend database for storing user credentials:

    a. Use the JNDI name `"jdbc/cs594"` for looking up the database.
    b. The SQL query for retrieving a user password is `"select PASSWORD from USERS where USERNAME = ?"`.
    c. The SQL query for retrieving a user's groups (to check their permissions) is `" select ROLENAME from USERS_ROLES where USERNAME = ?"`.
    d. You should use the `hashAlgorithm` and `hashAlgorithmParameters` attributes of the annotation to ensure that password hashing is consistent with your service logic (see the initialization of `MessageService`).

5. Path-based access control is configured in the Web app. In the `web.xml` descriptor, you need to define the security roles. You also need to define security constraints that restrict access to protected Web pages (in `/addUser/*`, `/admin/*`, `/moderator/*` and `/poster/*`) to users who have appropriate role-based authorization. Some of this is already done, to give you an example of how it is done. You do not need to configure login authentication in this descriptor, since this is done with annotations on the `AppConfig` class.

6. Jakarta EE access control uses roles to specify permission requirements, but users are associated to groups. This level of indirection supposedly provides some flexibility in determining if a user has permission for an access control. The default is to map roles to groups one-to-one. This is spelled out in the deployment descriptor `payara-web.xml`. This also maps the "`admin`" role name to a principal "`admin`", whose identity will be used for executing initialization code. The `MessageService` bean defines access requirements for executing some of its operations, including the role "`admin`" for creating roles and users. You will need to use the `@RunAs` annotation on `InitBean` in `ChatInit` to ensure that the initialization code runs with the authority of the `admin` principal, which `Payara-web.xml` enables with the `admin` role. Remember that the initialization code must run before the user database has been created, using operations that require permissions defined in this database. This is the workaround for this situation.

7. This list may not be exhaustive. See the unfinished tasks in Eclipse.

At this point, you should have a moderately secure Web app (module the vulnerabilities in the app server). Once you have the app working with these specifications, complete the following extensions.

As provided, the enterprise app passes passwords between client and server as cleartext, easily readable and modifiable by malicious network attackers. To prevent these attacks, you will need to protect communications, at least those that include passwords and any other type of confidential information, using SSL. The easy part of this is to add a user data constraint (in the Web deployment descriptor `web.xml`) that requires that the confidentiality of user data be protected during communications:

```
<security-constraint>
	<display-name>RootConstraint</display-name>
	<web-resource-collection>
		<web-resource-name>Root</web-resource-name>
		<description></description>
		<url-pattern>/*</url-pattern>
	</web-resource-collection>
	<user-data-constraint>
		<transport-guarantee>CONFIDENTIAL</transport-guarantee>
	</user-data-constraint>
</security-constraint>
```

Payara conflates this with preserving the integrity of the communications. Since Payara initializes with self-signed certificates, you will get browser warnings when you access the Web app. We will consider aspects of PKI in later assignments.

The deployment descriptor for the Web app enforces path-based RBAC for the Web interface, but you should also enforce RBAC at the service level. EJBs support the use of annotations on services to specify roles that are required for access to an API (For example, `@DeclareRoles` and `@RolesAllowed`). The container performs RBAC checks on API requests, at runtime, based on these annotations. Use these annotations to secure the services in `MessageService`. Operations for managing users and roles should only be available to administrators. Both moderators and posters have access to posted messages, but only a poster can post a new message, and only a moderator can delete a message. Remember that the initialization logic in `ChatInit` requires privileged access to the administration API in the app service, before anyone has logged in, so it will require the `@RunAs` annotation to give it the appropriate permission.

In addition to the declarative security checking above, there are also a couple of places where you will need to perform security checking programmatically. When a user logs in, they have to specify the role under which they are logging in. This is used to simplify navigation. However we must be careful to prevent someone from logging in with a role for which they do not have permission. Complete a check in the login logic that ensures that if the role they specify is not one that they are allowed to take, then a message is

displayed and they are logged out. The security context provides an operation for checking if a principal can act in a particular role.

RBAC only allows someone with posting permissions to post a new message, but the service API does not prevent a poster from posting something under someone's name. The code in `MessageService` for posting a message should check that the poster specified in the request is the same as the currently logged-in user[2]. You can get this information from the security context that is injected into the bean.

Here is further explanation of time-based one-time passwords. When a user is added, their authorization code is created from the following pieces of information:

1. A secret, which is a base 32 encoding of a 10-digit hexadecimal number, that should be unique to each user. For example, the string "8-polytope" is base 32 encoded as "HAWXA33MPF2G64DF". This secret is automatically generated from a (cryptographically) random number generator when a user record is added to the database.
2. An issuer, which we will assume is "Stevens Institute of Technology".
3. A label or subject, which we will take as the username being added to the system, e.g. "joe".

These three pieces of information are encoded as a key URI of the form:

`otpauth://totp/`*`issuer`*`:`*`subject`*`?secret=`*`secret`*`&issuer=`*`issuer`*

For example, for the sample above, we have the following key URI:

```
otpauth://totp/Stevens+Institute+of+Technology:joe?secret=HAWXA33MPF2G6
4DF&issuer=Stevens+Institute+of+Technology
```

The Web app provides an interface for adding users interactively, and generates a QR code that includes a randomly generated secret. The QR code is displayed by the Web app after the user is added to the system. You should scan the QR code with Google Authenticator, and use this when authenticating as the user to the Web app[3]. For debugging purposes, you can use the ZXing bar code scanner app to see the QR codes that you generate.

Since it would be difficult to manage the secrets of users that are added for testing purposes during app initialization, the service API provides a separate operation for adding test users, that leaves the TOTP secret null. Two-factor authentication is not enforced for these users.

---

[2] The Web tier uses the currently logged-in user to generate the request to the service, but we do not want to depend just on the Web tier for security.

[3] Note that the secret is stored in cleartext in the user database, instead of e.g. encrypting it with the user password

Submission

Deploy your working application in the Payara (Payara) application server in the VPC you set up in the previous assignment, using Postgresql as your database server, and record videos to demonstrate the working of your system. Use your own name as a user name in the demonstration, so it shows in the demo videos. See the rubric for what is expected in terms of your demonstration videos. You should provide a video of your use of Google Authenticator on a mobile device to authenticate with TOTP, see Canvas for some references on how to record this.

The source for your solution should be uploaded via the Canvas classroom, as a zip file. This zip file should have the same name as your Canvas userid. It should unzip to a folder with this same name, which should contain the files and subfolders with your submission. These should include the fully compiled Eclipse projects for your app, source code and all. You should retain the structure of the app provided to you: a collection of Maven modules with `ChatRoot` as the parent module. You should not make any changes to the domain model (`ChatDomain`), and changes to the business tier (`ChatService, ChatInit`) and Web tier (`ChatWebApp`) should be limited to what is described here and in the code. Your submission should also include an enterprise archive file (`ChatApp.ear`), ready for deployment.

As part of your submission, export your Eclipse Maven projects to your file system, and then include those folders as part of your archive file. Also include videos demonstrating your app working, as well as a completed rubric.