Java Technologies

Java EE Security

Securing Applications

- **Software Security** - Protecting applications against malicious / unauthorized actions
- **Desktop**
  - *What kind of code* is executed by the application, on the client machine?
  - *Where does the code* comes from?
  - *Who* wrote the code?
- **Web**
  - *Who* is accessing the application?
  - *What operations* does the client want to execute?

Securing Java EE Applications

- Java EE applications are made up of components that are deployed into various containers.
- Security for components is provided by their containers
- **Web Tier Security**
- **Enterprise Tier (EJB) Security**
- **Microservices Security**
- Transport / Messages / Data Security

Characteristics of Application Security

- **Authentication**
  - the users are who they say they are
- **Authorization, or access control**
  - the users have permissions
  - data integrity / (confidentiality) data privacy
- **Non-repudiation**
  - the transactions can be proved to have happened.
- **Auditing**
  - records of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms.

Security Layers

- **Application-Layer**
  - **Declarative security** expresses an application component's security requirements by using either deployment descriptors or annotations.
  - **Programmatic security** is embedded in an application and is used to make security decisions.
- **Transport-Layer**: cryptographic protocols: TLS, *SSL* for securing the network communication
- **Message-Layer**: security information is contained within the message and/or attachment, travelling along with it.

Realm, User, Group, Role

- A **realm** is a *security policy domain* defined for a web or application server. A realm contains a collection of **users**, who may or may not be assigned to a **group**.
- A **user** is an individual or application program identity that has been defined in the server.
- A **role** is an abstract name for the permission to access a particular set of resources in an application.
- **Credentials** – data that contains or references security attributes used for authentication.

Subject, Principal

javax.security.auth java.security

- A **Subject** represents a grouping of related information for a single entity, such as a person. Such information includes the Subject's identities as well as its security-related attributes (passwords and cryptographic keys, for example).
- Subjects may potentially have multiple identities. Each identity is represented as a **Principal** within the Subject. Principals simply bind names to a Subject.
- For example, a Subject that happens to be a person, Alice, might have two Principals: one which binds "Alice Bar", the name on her driver license, to the Subject, and another which binds, "999-99-9999", the number on her student identification card, to the Subject. Both Principals refer to the same Subject even though each has a different name.

Securing the Web Layer

- Create the user domain (realm)
  - a common scenario is using a relational database
- Create the security roles
- Define the authentication mechanism
- Define the security constraints for accessing the Web resources
- Map users to roles

Example: the *users* and *groups* tables

```
create table groups (
    id varchar(32) unique not null,
    name varchar(100) not null,
    primary key (id)
);
insert into groups values ('admin', 'System Admin');
insert into groups values ('user', 'Common people');
insert into groups values ('manager', 'The Boss');

create table users(
    id varchar(32) unique not null,
    password varchar(64) not null,
    name varchar(100) not null,
    email varchar(100),
    primary key(id)
);
insert into users values (
   'admin',
   encode(digest('admin', 'sha256'), 'hex'),
   'Administrator'
   'admin@mycompany.com');
```

how bytes are represented using chars

The digest is the output of a cryptographic hash function

Example: grouping the users

```
create table user_groups(

  group_id varchar(32) not null
       references groups(id) on delete restrict,

  user_id varchar(32) not null
       references users(id) on delete cascade,

  primary key (group_id, user_id)
);

insert into user_groups values ('admin', 'admin');
```

Configuring the Security Realm (GF)

- GF: Configuration → Security → Realms

```
realm name: myapp-realm
classname:
  com.sun.enterprise.security.ee.auth.realm.jdbc.JDBCRealm
jaas-context: jdbcRealm
datasource-jndi: jdbc/myDataSource
user-table: users
user-name-column: id
password-column: password
group-table: user_groups
group-name-column: group_id
group-table-user-name-column: user_id
digestrealm-password-enc-algorithm: SHA-256
encoding: Hex
charset: UTF-8
db-password
db-user
```

- Or use *asadmin: create-auth-realm* command

Creating the *roles*

- web.xml

```
<security-role>
    <description>Administrator</description>
    <role-name>admin</role-name>
</security-role>

<security-role>
    <description>Management</description>
    <role-name>manager</role-name>
</security-role>

<security-role>
    <description>Normal user</description>
    <role-name>user</role-name>
</security-role>
```

Login Configuration

- web.xml

```
<login-config>
     <auth-method>FORM</auth-method>
     <realm-name>myapp-realm</realm-name>
     <form-login-config>
         <form-login-page>/faces/login.xhtml</form-login-page>
         <form-error-page>/faces/login.xhtml</form-error-page>
     </form-login-config>
</login-config>
```

- NONE, BASIC, FORM,
  DIGEST, CLIENT CERTIFICATE

Digest Authentication

- Like basic authentication, digest authentication authenticates a user based on a user name and a password.
- However, unlike basic authentication, digest authentication does not send user passwords over the network. Instead, the client sends a one-way cryptographic hash of the password and additional data.
- Although passwords are not sent on the wire, digest authentication requires that clear-text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

Client Certificate

- The web server authenticates the client by using the client's public key certificate. Client authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's public key certificate.
- You can think of a public key certificate as the digital equivalent of a passport. The certificate is issued by a trusted organization, a certificate authority (CA), and provides identification for the bearer.
- Before using client authentication, make sure the client has a valid public key certificate.

Implementing the Login in JSF

```
FacesContext context = FacesContext.getCurrentInstance();
ExternalContext externalContext = context.getExternalContext();
HttpServletRequest request = (HttpServletRequest)
        externalContext.getRequest();
try {

  request.login(userId, password);

  User user = userRepo.findById(userId);
  ...

  externalContext.log("Login successful: " + userId);

} catch (ServletException | RuntimeException e) {
  externalContext.log("Login failed: " + userId + "\n" + e);
  throw new MyLoginException(userId, e);
}
```

In the submit method of the login.xhtml backing bean.

(or maybe in a helper *AuthService* class)

Implementing the login in HTML

Using standard HTML form tags allows developers to specify the correct action and input IDs for the form.

```
<form action="j_security_check" method="POST">
  <input type="text" name="j_username" />
  <input type="secret" name="j_password" />
...
</form>
```

Security Constraints (web.xml)

```
<security-constraint>
    <display-name>Admin Only</display-name>
    <web-resource-collection>
        <web-resource-name>User management</web-resource-name>
        <url-pattern>/faces/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>

<security-constraint>
    <display-name>Manager Only</display-name>
    <web-resource-collection>
        <web-resource-name>Application config</web-resource-name>
        <url-pattern>/faces/config/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

<security-constraint>
    <display-name>All Pages</display-name>
    ...
</security-constraint>
```

Mapping *users* to *roles*

- glassfish-web.xml

```
<security-role-mapping>
    <role-name>admin</role-name>
    <group-name>admin</group-name>
</security-role-mapping>

<security-role-mapping>
    <role-name>manager</role-name>
    <group-name>manager</group-name>
</security-role-mapping>

<security-role-mapping>
    <role-name>user</role-name>
    <group-name>user</group-name>
</security-role-mapping>
```

Configuring the *access denied* page

- **HTTP 403 Forbidden**

  The request was valid, but the server is refusing action. The user might not have the necessary permissions for a resource, or may need an account of some sort.

- web.xml

```
<error-page>
   <error-code>403</error-code>
   <location>/faces/pages/forbidden.xhtml</location>
</error-page>
```

Security for Servlets

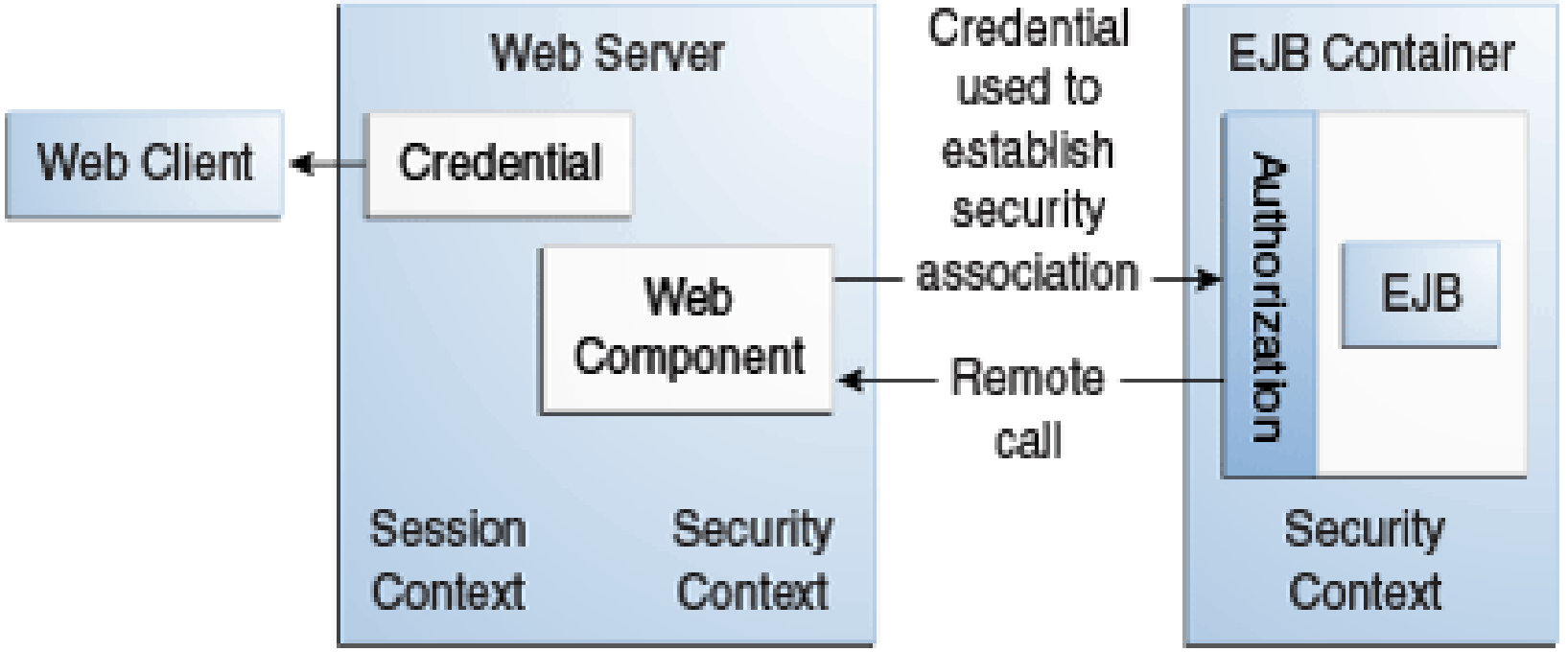
```
@WebServlet(name = "HelloWorldServlet",
            urlPatterns = {"/hello"})


@ServletSecurity(
 @HttpConstraint(
    transportGuarantee = TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"admin", "manager"}
  ))
public class HelloWorldServlet extends HttpServlet {
 …
}
```

All user data must be encrypted by the transport (typically using SSL/TLS)

Securing EJB Methods

- The EJB container is responsible for enforcing access control on the enterprise bean method.



EJBContext, SessionContext

- The **EJBContext** interface provides an instance with access to the container-provided runtime context of an enterprise bean instance. This interface is extended by the SessionContext, EntityContext, and MessageDrivenContext interfaces to provide additional methods specific to the enterprise interface bean type.
- The **SessionContext** interface provides access to the runtime session context that the container provides for a session bean instance. The container passes the SessionContext interface to an instance after the instance has been created. The session context remains associated with the instance for the lifetime of the instance.

Using the SecurityContext

java.security.Principal getCallerPrincipal()

```
@Stateless public class EmployeeServiceBean
                              implements EmployeeService {
  @Resource SessionContext ctx;
  @PersistenceContext EntityManager em;

  public void changePhoneNumber(...) {
    // obtain the caller principal.
   callerPrincipal = ctx.getCallerPrincipal();

    // obtain the caller principals name
  callerKey = callerPrincipal.getName();

  // use callerKey as primary key to find EmployeeRecord
  EmployeeRecord myEmployeeRecord =
     em.findByPrimaryKey(EmployeeRecord.class, callerKey);

  // update phone number
  myEmployeeRecord.setPhoneNumber(...);
  }
}
```

Using the SecurityContext

boolean isCallerInRole(String roleName)

```
@DeclareRoles({"admin", "manager", "payroll"})
@Stateless public class PayrollBean implements Payroll {
  @Resource SessionContext ctx;

  public void updateEmployeeInfo(EmplInfo info) {
    oldInfo = ... read from database;
    // The salary field can be changed only by callers
    // who have the security role "payroll"
    if (info.salary != oldInfo.salary &&
        !ctx.isCallerInRole("payroll")) {
           throw new SecurityException(...);
     }
     ...
  }
}
```

Using Declarative Security

- @RolesAllowed, @PermitAll, @DenyAll

```
@RolesAllowed("admin")
@Stateless public class SomeBean {
  public void aMethod () {...}
  public void bMethod () {...}
...
}

@Stateless public class MyBean extends SomeBean {
  @RolesAllowed("guest")
  public void aMethod () {…}

  @PermitAll()
  public void cMethod () {...}
...
}
```

Securing REST Services

- Updating the *web.xml* deployment descriptor to define security configuration.
- Using the *javax.ws.rs.core.SecurityContext* interface to implement security programmatically.
- Applying *annotations* to your JAX-RS classes.
- Using Jersey OAuth libraries to sign and verify requests

Using Annotations

```
import javax.annotation.Security.RolesAllowed;


@Path("/hello")
@RolesAllowed({"admin", "manager", "guest"})
public class HelloWorld {


   @GET
   @Produces("text/plain")
   @RolesAllowed("admin")
   public String sayHello() {
      return "Hello World!";
   }
}
```

Using SecurityContext

```
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.Context;


@Path("/hello")
public class MyService {
...
    @GET
    @Produces("text/plain;charset=UTF-8")
    public String sayHello(@Context SecurityContext sc) {
        if (sc.isUserInRole("admin"))  return "Hello World!";
        throw new SecurityException("User is unauthorized.");
    }
}
```

web.xml

```
<web-app>
    ...
     <security-constraint>

          <web-resource-collection>
              <web-resource-name>Orders</web-resource-name>
              <url-pattern>/orders</url-pattern>
              <http-method>GET</http-method>
              <http-method>POST</http-method>
          </web-resource-collection>

          <auth-constraint>
              <role-name>admin</role-name>
          </auth-constraint>
     </security-constraint>
    ...
</web-app>
```

Issues

- The various EJB and servlet containers comprising Java EE defined similar security-related APIs, but with subtly different syntax. For example, a servlet's call to check a user's role was HttpServletRequest.isUserInRole, while an EJB would call EJBContext.isCallerInRole.
- Existing security mechanisms like JACC (Java Authorization Contract for Containers) were tricky to implement, and JASPIC (Java Authentication Service Provider Interface for Containers) could be difficult to use correctly.
- Existing mechanisms did not take advantage of modern Java EE programming features such as CDI injection.
- There was no portable way to control how authentication happened on the backend across containers.
- There was no standard support for managing identity stores or the configuration of roles and permissions.
- There was no standard support for deploying custom authentication rules.

The Java EE 8 Security API

- A new standard and a portable way of handling security concerns in Java containers
- Enable developers to manage and control security themselves, by defining portable APIs for authentication, identity stores, roles and permissions, and authorizations across containers.
- Provides an alternative way to configure identity stores and authentication mechanisms, but *does not replace existing security mechanisms*.
- The Java EE Security API empowers developers to enable security in Java EE web applications in a consistent and portable manner—with or without vendor-specific or proprietary solutions.

API Core Features

- **HTTP Authentication Mechanism**
  - A mechanism for obtaining a caller's credentials in some way, using the HTTP protocol where necessary.
- **Identity Store**
  - A mechanism for validating a caller's credentials and accessing a caller's identity attributes, obtaining data from a persistent store, such as a database, LDAP server, or file.
- **Security Context**
  - Provides an access point for programmatic security. This type must be usable in all Java EE containers, specifically the Servlet and EJB containers.

Implementations

- Java EE 8 compliant servers already provide an implementation for the Java EE Security API
  - Payara, Wildfly, Tomee, OpenLiberty, etc.
  - Maven: **javaee-web-api**
- **Eclipse Soteria** provides the reference implementation for Java EE Security API
  - https://github.com/javaee/security-soteria
  - Maven
    - javax.security.enterprise
    - org.glassfish.soteria

Support for Servlet Authentication

- A Java EE container must provide *HttpAuthenticationMechanism* implementations for three authentication mechanisms, which are defined in the Servlet 4.0 specification.
  - Basic HTTP authentication

    → *@BasicAuthenticationMechanismDefinition*
  - Form-based authentication

    → *@FormAuthenticationMechanismDefinition*
  - Custom-form authentication

    → *@CustomFormAuthenticationMechanismDefinition*
- It is no longer necessary to specify the authentication mechanism in the web.xml file.

Example

```
@CustomFormAuthenticationMechanismDefinition(
        loginToContinue = @LoginToContinue(
                loginPage = "/faces/login.xhtml",
                errorPage = "",
                useForwardToLogin = false
        )
)
@ApplicationScoped
public class AppConfig {
}
```

The LoginBean

<p:commandButton value="Submit" action="#{loginBean.login()}" />

```
@Named
@RequestScoped
public class LoginBean {

    @Inject
    private SecurityContext securityContext;

    @NotNull private String username;
    @NotNull private String password;

    public void login() {
        Credential credential = new UsernamePasswordCredential(
          username, new Password(password));
        AuthenticationStatus status = securityContext
          .authenticate(
            getHttpRequestFromFacesContext(),
            getHttpResponseFromFacesContext(),
            withParams().credential(credential));
        // ...
    }
    // ...
}
```

SEND_CONTINUE, SEND_FAILURE, SUCCESS

The LoginBean (cont)

The AuthenticationStatus is used as a return value by primarily the HttpAuthenticationMechanism to indicate the result (status) of the authentication process.

```
switch (status()) {

    case SEND_FAILURE:
        facesContext.addMessage(…); //Invalid credentials!
        break;

    case SUCCESS:
        //Forward or redirect to the desired page
        break;

    case SEND_CONTINUE:
        //Multi-step authentication dialog
        facesContext.responseComplete();
        break;

     case NOT_DONE:
        //Optional authentication; decided not to authenticate
        ...
}
```

The *IdentityStore* API

javax.security.enterprise.identitystore

- An identity store is a database that stores user identity data such as user name, group membership, and information used to verify credentials.
- An application can provide its implementation of the *IdentityStore* interface or use one of the two built-in implementations provided by the container for:
  - **Database** → *@DataBaseIdentityStoreDefinition*
  - **LDAP** → *@LdapIdentityStoreDefinition*
- Identity stores are queried in order, determined by the priority of each IdentityStore implementation. The list of stores is parsed twice: first for authentication and then for authorization.

Example

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup = "jdbc/myDataSource",
  callerQuery =
     "select password from users where id = ?",
  groupsQuery =
     "select group_id from user_groups where user_id = ?",
  Priority = 30)
@ApplicationScoped
public class AppConfig {
}
```

The *SecurityContext* API

- **Programmatic security** enables a web application to perform the checks required to grant or deny access to application resources.
- An alternative when the declarative security model enforced by the container isn't sufficient.

```
@Inject
SecurityContext securityContext;
```

- Unless otherwise indicated, this type must be usable in all Java EE containers, specifically the Servlet and EJB containers.

Example

```
@DeclareRoles({"admin", "user", "demo"})
@WebServlet("/demoServlet")
public class DemoServlet extends HttpServlet {
  @Inject
  private SecurityContext securityContext;

  @Override
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
          throws ServletException, IOException {

    boolean hasAccess = securityContext
                .hasAccessToWebResource("/protected", "GET");
    String user = securityContext
                .getCallerPrincipal().getName();
    if (hasAccess && !user.equals("hacker")) {
      req.getRequestDispatcher("/protected").forward(req, res);
    } else {
      req.getRequestDispatcher("/logout").forward(req, res);
    }
  }
}
```