Software Security (SSI)

8. Java Web Application Security – Part 2

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

Access Control and Authentication

The extensions of this section are integrated in marketplace_v03.

Marketplace – Admin Area Extension (1)

- We extend our application with an admin area
- The admin area allows sales and marketing personnel to view the purchases
- It also allows to complete a purchase, which results in the removal of the entry from the Purchase table
- All admin servlets/JSPs are placed in the subdirectory /admin/
  - This allows protecting access using declarative security mechanisms
- A new button is added to the index page to get to the admin area:

Marketplace – Admin Area Extension (2)

- Clicking the "Admin area" button opens the Purchases page
  - Shows all entries in the table Purchase

ubuntu.dev:8080/marketplace/admin/listpurchases

Purchases

First Name	Last Name	CC Number	Total Price	
Ferrari	Driver	1111 2222 3333 4444	$250,000.00	Complete Purchase
C64	Freak	1234 5678 9012 3456	$444.95	Complete Purchase
Script	Lover	5555 6666 7777 8888	$10.95	Complete Purchase

Return to search page | Logout and return to search page

- Completing a purchase removes the entry from the database and the list

ubuntu.dev:8080/marketplace/admin/completepurchase?purchaseId=1

Purchases

Purchase completed

First Name	Last Name	CC Number	Total Price	
C64	Freak	1234 5678 9012 3456	$444.95	Complete Purchase
Script	Lover	5555 6666 7777 8888	$10.95	Complete Purchase

Return to search page | Logout and return to search page

Marketplace – Admin Area Extension (3)

What is needed to extend the application with the new functionality?

- A servlet ListPurchasesServlet that is invoked to list the purchases
- A JSP /admin/purchases.jsp that shows the list
- A servlet DeletePurchaseServlet that deletes a purchase
- Two new methods in PurchaseDB.java
  - `public static ArrayList<Purchase> getPurchases()`
    gets all purchases
  - `public static int delete(int purchaseId)`
    deletes the purchase with the specified PurchaseID
- Entries in web.xml to define the servlets and the mappings

Implementing this is very similar to what we have discussed in the basic application, so we do not discuss the code in detail

Marketplace – Code Snippets – web.xml

- The new servlets require the following entries in web.xml:

```
<servlet>
  <servlet-name>ListPurchasesServlet</servlet-name>
  <servlet-class>servlets.ListPurchasesServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ListPurchasesServlet</servlet-name>
  <url-pattern>/admin/listpurchases</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>DeletePurchaseServlet</servlet-name>
  <servlet-class>servlets.DeletePurchaseServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>DeletePurchaseServlet</servlet-name>
  <url-pattern>/admin/completepurchase</url-pattern>
</servlet-mapping>
```

The servlets are mapped to URLs below /admin/, which is important for access control purposes!

Access Control with Declarative Security (1)

- With these extensions, the entire admin functionality is available as intended, but unfortunately to every user…
- We therefore need some access control mechanisms to make sure only the intended users are granted access to the admin area
- We could implement this programmatically:
  - Define users and corresponding rights and store this in the database
  - Implement an authentication mechanism to authenticate user
  - Store the identity of the authenticated user in the session
  - Whenever a user access the admin area, check if he is authenticated and has the necessary rights
- However, this is cumbersome and error-prone because it's very easy to make security-relevant programming mistakes
  - E.g. forgetting to consistently check the access rights with every request

Access Control with Declarative Security (2)

- To overcome this problem, Java EE (and other technologies) provide declarative security mechanisms to handle access control
  - Using this is much more simple and more secure than with programmatic security
- Java EE supports this with a role-based access control mechanism:
  - Users and roles are defined
  - Access rights are assigned to roles
  - Roles are assigned to users, a user can have multiple roles
  - A user gets all rights of the assigned roles
- Configuring which roles are allowed to access what resources is done in web.xml

Marketplace – Define Access Control Rules in web.xml (1)

- Restrict access to /admin/* to roles sales and manager:

```
<security-role>
  <description>Sales Personnel</description>
  <role-name>sales</role-name>
</security-role>

<security-role>
  <description>Marketing Personnel</description>
  <role-name>marketing</role-name>
</security-role>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Area</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>sales</role-name>
    <role-name>marketing</role-name>
  </auth-constraint>
</security-constraint>
```

Define the roles sales and marketing, `<description>` is purely informative

Restrict access to all resources in `/admin/` to the roles `sales` and `marketing`

`<auth-constrain>`

Using the special role name * in an auth-constraint means all roles described in the deployment descriptor.

`<security-role>`

Usually, the application will also function correctly without the <security-role> elements. Nevertheless, you should include them to make sure that it conforms “to the standard” and that no runtime problems occur in any servlet/JSP engine.

Marketplace – Define Access Control Rules in web.xml (2)

- Running the application and trying to access the admin area results in the following:

- This is reasonable:
  - We have restricted access to the roles sales and marketing, but haven't assigned these roles to users
  - A HTTP 403 response ("Forbidden") is the correct behavior of Tomcat
- Side note: This also should get a standard error message

Java EE Realms

- What we need now are users that can be authenticated and that get the corresponding roles
- In Java EE, this usually done via so called Realms
  - A Realm is storage location that contains usernames, passwords and associated roles of valid users
- The actual implementation of the Realms is provided by the underlying servlet/JSP engine or application server
  - Different products provide different Realms, but the popular Realms are usually supported in most products
- Tomcat supports several Realms, the most popular ones are:
  - UserDatabaseRealm: Usernames, passwords and roles are stored in a file (tomcat-users.xml)
  - JDBCRealm: Usernames, password and roles are stored in a database and are accessed via JDBC

Realms

Although the different products offer different Realms, they are often similar. For instance, Tomcat, JBoss and GlassFish all support the JDBCRealm, but the individual configurations may differ.

UserDatabaseRealm

- This is enabled per default in Tomcat, so except configuring tomcat-users.xml, no additional configuration is required
- Example of tomcat-users.xml

```
<tomcat-users>
  <role rolename="sales"/>
  <role rolename="marketing"/>
  <user username="john" password="wildwest" roles="sales"/>
  <user username="alice" password="rabbit" roles="sales,marketing"/>
  <user username="robin" password="arrow" roles="marketing"/>
</tomcat-users>
```

Define two roles sales and marketing

Define three usernames with passwords and assigned role(s)

- Usage of the UserDatabaseRealm?
  - OK for web applications with only a few user or for testing purposes
  - But not really well suited with many users, as maintaining a flat file of users and roles is cumbersome

UserDatabaseRealm

This is used per default in Tomcat and is specified in server.xml:

```
<Resource name="UserDatabase" auth="Container"
  type="org.apache.catalina.UserDatabase"
  description="User database that can be updated and saved"
  factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
  pathname="conf/tomcat-users.xml" />
```

Using the UserDatabaseRealm?

This security realm can certainly be used in smaller, simpler web applications with a relatively small number of users or also for testing purposes to quickly specify some users and roles. For “more serious” web application, this security realm is less well suited. One reason is that with a growing number of users, it gets cumbersome to maintain the file. Also, the file can only be maintained manually and not “from within the application”, which means the UserDatabaseRealm cannot be used if users should be able to register themselves in the web applications or if they should be able to change their password. In these cases, using the JDBCRealm is much better suited, as accessing the database from within the application is easily possible.

JDBCRealm

- With "real" web applications, users, passwords and roles are usually stored in a database → JDBCRealm
- This requires configuring the realm in the application's context.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>

<Context path="/marketplace">

  <Resource name="jdbc/marketplace" auth="Container" ... />

  <Realm className="org.apache.catalina.realm.JDBCRealm"
    driverName="com.mysql.jdbc.Driver"
    connectionURL="jdbc:mysql://localhost:3306/marketplace"
    connectionName="marketplace" connectionPassword="marketplace"
    userTable="UserPass" userNameCol="Username" userCredCol="Password"
    userRoleTable="UserRole" roleNameCol="Rolename" />

</Context>
```

Defines user table and columns for username and password

Defines user role table (contains the mapping of usernames to roles) and the column for the role (username column must be the same as in user table)

context.xml

The first part (for connection pooling) has been configured before.

Marketplace – UserPass and UserRole Tables

- Corresponding to the JDBCRealm configuration, we define the following two tables
- Table UserPass contains the usernames and passwords

Username	Password
john	wildwest
alice	rabbit
robin	arrow
donald	daisy
luke	jollyjumper
bob	patrick

- Table UserRole contains usernames and associated roles

Username	Rolename
alice	sales
bob	burgerman
donald	productmanager
john	sales
luke	productmanager
robin	marketing

Sending Credentials to the Web Application

- So we have:
  - Database entries that define usernames, passwords and assigned roles
  - A configuration entry in context.xml so that these tables are used for the Marketplace application
- In addition, we have to specify how username and password are sent to the web application during authentication
- We can choose between the following variants:
  - HTTP BASIC authentication – A dialog box opens and username and password are included in the HTTP request authorization header
  - HTTP DIGEST authentication – Browser does not send the password but a hash over the password and a challenge from the server
  - FORM-based authentication – Use a login form for authentication

Authentication Security

Of course, all three variants should only be used over HTTPS. While HTTP Digest Authentication avoids easy interception of the password by an attacker if HTTP is used, it still can easily be attacked by, e.g., a man in the middle. To do so, the attacker simply modifies the HTTP 401 response from the server and adapts it such that BASIC authentication is used by the browser.

CLIENT-CERT Authentication

Java EE also supports CLIENT-CERT authentication: Use a certificate for user authentication (always uses HTTPS / HTTP over TLS).

When using certificates for user authentication, a realm (e.g. UserDataRealm or JDBCRealm) is still needed. Since no user password is needed, the password can be set to null and the X.500 name of the subject in the certificate is used as the username.

Marketplace – BASIC Authentication (1)

- Enabling HTTP BASIC authentication is done by adding the following to web.xml:

```
<security-role>
  ...
</security-role>

<security-constraint>
  ...
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Marketplace</realm-name>
</login-config>
```

Defines the authentication method (here: BASIC)

With BASIC authentication, `<realm-name>` is just the string displayed in the login dialogue box

- This `<login-config>` entry has the following effect:
  - If the user accesses a resource that is protected in a `<security-constraint>` with an `<auth-constraint>` (`/admin/*` in our case)...
  - ...and the user is currently not authenticated...
  - ...it is authenticated using HTTP Basic authentication

Marketplace – BASIC Authentication (2)

- Accessing the admin are now causes the following:

- Successful login allows access to the admin area, clicking Cancel shows an HTTP 401 status message (another standard error message candidate)

HTTP Authentication

When trying to access the admin area, the server replies with an HTTP 401 “Unauthorized” message, which causes the browser to show the login dialogue box. Entering wrong credentials causes the dialogue box tp pop up again and again, clicking “Cancel” displays the HTTP 401 status message.

BASIC Authentication – Limitations

- While simple and supported by every browser, BASIC authentication has one major drawback:
  - Once you have authenticated, the only real way to "log out" or switch to another user is closing the browser
- The reason is that the browser sends the authorization header (contains base64-encoded username and password) with every subsequent request to the admin area
  - e.g. `Authorization: Basic am9objp3aWxkd2VzdA==`
- This means there is no simple way to log out the user or log in as a different user: the server always gets the same credentials
  - Because the web browser does not forget them

Logging out

Well, you can enforce a new login dialogue box programmatically: write a servlet that sends an HTTP 401 status message, but that's not declarative security and therefore more complex. In addition, even if the user wants to log out, he'd get again the logon dialogue box, which he'd have to "click away" with "Cancel", which is ugly from a usability point of view.

Marketplace – FORM-based Authentication

- FORM-based authentication is the preferred authentication method today
  - Does not have the limitations of BASIC authentication
  - The credentials are sent only once: during actual authentication
  - Can be integrated into the look-and-feel of the application

- Enabling FORM-based authentication requires also a `<login-config>` entry in web.xml:

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/admin/login.jsp</form-login-page>
    <form-error-page>/admin/login_error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Defines the authentication method (here: FORM)

The login and error pages to use

Logon and Error Pages in the /admin Directory

This would not be necessary. The reason we place them in the /admin directory makes integration with TLS (see later) a little easier as we can configure using TLS for the entire /admin directory, which then includes the login and error pages. Maybe you ask yourself whether these resources can be accessed at all during login, as they are below /admin/. The reason why this works is because calling the login JSP is an application-internal forwarding, which is never affected by security constraints (this will be discussed later in this chapter).

Servlets, JSPs or HTML Pages

Note that one can use arbitrary resources for the login and error pages. In our case, we use JSPs so we can make use of the c:url tag.

Marketplace – Login Page /admin/login.jsp

- The login page contains a form that must use specific values for the action and the parameter names for username and password
  - These values are defined by Java EE and must be used so this request is interpreted as a login request

```
<h1>Admin Login Form</h1>

<p>Please enter your username and password to continue.</p>

<form action="<c:url value='j_security_check' />" method="post">
  <table cellspacing="5" border="0">
    <tr>
      <td align="right">Username</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td align="right">Password</td>
      <td><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td><input type="submit" value="Login"></td>
    </tr>
  </table>
</form>
```

It is mandatory to use these values for the action and the names of the username and password parameters

Request to j_security_check

The request can use GET or POST, both works. But passwords should never be transmitted in GET requests.

Marketplace – Login Page /admin/login_error.jsp

- The login error page is sent to the browser if login fails, this can basically contain any content
  - E.g. again a login page with an additional message

```
<h1>Admin Login Form - Error</h1>

<p>You did not log in successfully.</p>

<p>Please check your username and password and try again.</p>

<form action="<c:url value='j_security_check' />" method="post">
  <table cellspacing="5" border="0">

    ...

  </table>
</form>
```

Marketplace – FORM-based Authentication

- Accessing the admin are now causes the following:

ubuntu.dev:8080/marketplace/admin/listpurchases

Admin Login Form

Please enter your username and password to continue.

Username john

Password ••••••••

Login

- Successful login allows access to the admin area, wrong login shows the login error page

ubuntu.dev:8080/marketplace/admin/listpurchases

Purchases

First Name	Last Name	CC Number	Total Price	
C64	Freak	1234 5678 9012 3456	$444.95	Complete Purchase
Script	Lover	5555 6666 7777 8888	$10.95	Complete Purchase

ubuntu.dev:8080/marketplace/admin/j_security_check

Admin Login Form - Error

You did not log in successfully.

Please check your username and password and try again.

Username

Password

Login

Logging Out

- The web application stores information about the logged in user in the session
- To log a user out, simply invalidate the session by calling the `invalidate()` method of the `HTTPSession` object
  - It basically removes the old session (including all attributes currently stored in it) and creates a new one
  - It also causes the web application to send a new session ID to the browser (`Set-Cookie: JSESSIONID=A64E2...`)
  - If some attributes – e.g. the shopping cart – should be preserved, the attributes must be copied manually from the old to the new session
- As a reminder: this does not really work with BASIC authentication, as the browser continues to send the authorization header

New Session

Calling the invalidate means that all attributes of the session are lost. If you want, you can "transfer" attributes from the old to the new session (e,g, the shopping cart in the Marketplace application), but you have to do so manually.

Marketplace – LogoutServlet

- To log out, we implement a Logout servlet and add the mapping to web.xml

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

  // Get the session and invalidate it
  HttpSession session = request.getSession();
  session.invalidate();
  request.setAttribute("message", "You have been logged off.");

  // Forward to JSP
  String url = "/index.jsp";
  ...
}
```

```
<servlet>
  <servlet-name>LogoutServlet</servlet-name>
  <servlet-class>servlets.LogoutServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>LogoutServlet</servlet-name>
  <url-pattern>/admin/logout</url-pattern>
</servlet-mapping>
```

Marketplace – Logging Out (1)

- We also add a new button "Logout and return to search" to the Purchases page, which calls the logout servlet:

Purchases

First Name	Last Name	CC Number	Total Price	
Commodore	Fan	4321 8765 1234 5678	$444.95	Complete Purchase

Return to search page | Logout and return to search page

- Clicking the link results in invalidating the session (logging out) and displays the search page including a message:

ubuntu.dev:8080/marketplace/admin/logout

Welcome to the Marketplace

You have been logged off.

To search for products, enter any search string below and click the Search button

Search

Show cart | Checkout | Admin area

Marketplace – Logging Out (2)

- Trying to access the admin area again redirects the user to the login page
  - This is reasonable, as the application has "forgotten" about the previously logged in user

Hashed Password

- Storing plaintext passwords is not a good idea for security reasons
  - An intruder who manages to access the database server or who performs an SQL injection attack easily gets all passwords
- Tomcat allows to store the password hashes instead of the passwords in the database, this works with all realms
- It only requires very little adaptation in context.xml:

Specifies the hash algorithm to use, e.g. MD5, SHA-1, SHA-256, SHA-512

```
<Realm className="org.apache.catalina.realm.JDBCRealm" digest="SHA-256"
  driverName="com.mysql.jdbc.Driver"
  connectionURL="jdbc:mysql://localhost:3306/marketplace"
  connectionName="marketplace" connectionPassword="marketplace"
  userTable="UserPass" userNameCol="Username" userCredCol="Digest1"
  userRoleTable="UserRole" roleNameCol="Rolename" />
```

The column in the UserPass table that contains the hashes

Hashed Passwords in Tomcat

For more information, see *http://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html#Digested_Passwords*

Helper Class for Crypto Stuff

```
package util;

import java.security.MessageDigest;

public class Crypto {

  public static byte[] computeSHA256(String input) throws Exception {
    MessageDigest md = MessageDigest.getInstance("SHA-256");
    md.update(input.getBytes());
    return md.digest();
  }

  public static String printHex(byte[] input) {
    String output = "";
    for (int i = 0; i < input.length; i++) {
      int v = input[i] & 0xFF;
      if (v < 16) output += "0";
      output += Integer.toString(v, 16);
    }
    return output;
  }

  public static void main(String argv[]) throws Exception {
    System.out.println(printHex(computeSHA256(argv[0])));
  }
}
```

Compute SHA-256 hash

Convert byte-array to Hex string

Compute the Hashed Passwords

- We use this helper class to compute the hashed passwords:

```
$ java util.Crypto wildwest
9de6156696ac75758c79be74b981021c428a0a01b
eb5a6212b408848fd133b2e
...
```

- And add a new column "Digest1" to the UserPass table:

Username	Password	Digest1
alice	rabbit	d37d96b42ad43384915e4513505c30c0b1c4e7c765b5577
bob	patrick	23ddda4810068cc44360dffd31b6c5a9ad13fb9e6a69c935
donald	daisy	42029ef215256f8fa9fedb53542ee6553eef76027b116f8fac
john	wildwest	9de6156696ac75758c79be74b981021c428a0a01beb5a62
luke	jollyjumper	5ed7d3b54834d251a362ff851dff2186fd3ef37fa2feb41cbe
robin	arrow	864ea39fe7e2b27155a29e46fb9458d05e2ccade4151d02b

- Remarks:
  - Tomcat requires that the hashes are stored in hex format
  - Of course, one would no longer store the raw password in the table in a real application
- Question: Is using a raw hash of the password a good idea?

Tomcat also provides Functionality to compute Digests

One can also use the following, tomcat built-in methods:

- *main*-method in the class *RealmBase*: `java -classpath {TOMCAT-HOME}/lib/catalina.jar:tomcat/bin/tomcat-juli.jar org.apache.catalina.realm.RealmBase -a SHA-256 wildwest`
- *TOMCAT_HOME/bin* contains a script *digest.sh* to compute digests and the output is provided as a hex-string. The usage to compute a SHA-256 hash is as follows: `digest.sh -a SHA-256 password`
- In addition, the class *RealmBase* contains a *digest* method, which can be used in the code to compute digests. The usage is as follows: `String digest = RealBase.Digest("password", "SHA-256", null);` to compute a SHA-256 hash, again in hex-format.

In our examples, we have used our own class *Crypto* to do the same, which has the advantage that it is portable beyond Tomcat.

Salt

Actually, a patch has been submitted to add this functionality to Tomcat, but it hasn't officially been added to Tomcat yet. *See http://stackoverflow.com/questions/205153/how-would-you-implement-salted-passwords-in-tomcat-5-5*

Access Control and Authentication – Summary

- A `<security-constraint>` with an `<auth-constraint>` entry in web.xml is used to restrict access to certain resources to authenticated users with specific roles
- To specify usernames, passwords and roles, a Realm is used
  - Often, JDBCRealm is used, which requires two tables with usernames, passwords and roles
  - The Realm to be used must be configured in `context.xml`
- A `<login-config>` entry is used in web.xml to specify how the credentials are sent to the web application during authentication
  - Often, FORM-based authentication is used, the corresponding form must use the correct values for the action and the parameter names
- Authentication is requested by the web application when a non-authenticated users accesses a restricted resource

Note that all this uses declarative security, we have not implemented a single line of code (except the login form)

Access Control and Authentication with Programmatic Security

Note that Access Control and Authentication could also be implemented using a completely own mechanism enforced directly in the code, but this is much more error-prone and also more complex. Therefore, whenever possible, you should use the declarative mechanisms provided by Java EE.

Secure Communication

The extensions of this section are integrated in marketplace_v04.

Secure Communication

- We have added authentication and access control to the web application
  - Without securing the communication between the browser and server, it's very easy to eavesdrop on login data
- As a result, we should protect the link between browser and server as soon as we enter protected areas of the application → TLS (HTTPS)
- How can this be done?
  - We simply use HTTPS for everything and don't even make the HTTP port available → could be done, but implies a performance overhead and offers little flexibility
  - Specify the protocol in the URLs presented to the user (https://...) when accessing protected areas → does not work well as the protected resources can still be accessed with HTTP
  - In the web application, specify which areas can only be accessed with HTTPS → this is a secure and the most flexible option, which we will use

Marketplace – Secure Communication

In the Marketplace application, the following areas should be protected:

- The entire admin area
- The step where the user completes the purchase and enters the credit card information

With Java EE, this can be configured with declarative security:

- Add a `<user-data-constraint>` element to a `<security-constraint>`
- Within `<user-data-constraint>`, specify with `<transport-guarantee>` what protection is desired:
  - CONFIDENTIAL: full protection (confidentiality, authenticity, integrity) – which is what we usually expect from TLS → use this!
  - INTEGRAL: Only authenticity and integrity protection must be enforced
  - NONE: No specific protection needed

CONFIDENTIAL / INTEGRAL

In practice, Java EE servers treat the two alike, the always use integrity protection and confidentiality. Still, you should always use CONFIDENTALITY when protection is desired.

Marketplace – `<user-data-constraint>` in web.xml (1)

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Area</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>sales</role-name>
    <role-name>marketing</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Purchase</web-resource-name>
    <url-pattern>/checkout</url-pattern>
    <url-pattern>/purchase</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Access to admin area is only allowed via HTTPS

Access to check-out and purchase functions is only allowed via HTTPS (no auth-constraint means every user can access!)

Configure Tomcat to support HTTPS

- Per default, Tomcat is usually not configured to support HTTPS
- Typical configuration can be as follows (server.xml):

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
        keystoreFile="/home/rennhard/tomcat/.keystore"
        keystorePass="changeit"
        maxThreads="150" scheme="https" secure="true"
        clientAuth="false" sslProtocol="TLS" />
```

  - The keystoreFile contains the server's (self-signed) certificate and private key, which can easily be generated with keytool
  - keystorePass is the password to access the keystore
  - In a productive setting, you would most likely use a "real" certificate signed by an official CA

Marketplace – Securely accessing protected areas

- Accessing the admin area now causes the following:

- And checking out also uses HTTPS:

What happens in the background?

- Clicking the link in the browser causes an HTTP request
- The server – when a protected resource is accessed – sends back an HTTP 302 redirect to the same resource, but with https://...

Securely Accessing the Checkout Form? – Exercise

- Note that we have not only protected access to /purchase (which sends the payment data to the server), but also to /checkout (which displays the form)
  - Question: Why does this make sense? We only submit sensitive data when accessing /purchase.

From HTTPS to HTTP

- In the Marketplace application, when navigating from an HTTPS page to an HTTP page, the following happens:

- The "HTTP"-page is still accessed via HTTPS
  - Reason: The JSPs in the Marketplace application don't specify the protocol to access a resource, so the browser uses the current protocol (HTTPS)
    - This is not a real problem (although it costs a little performance)
  - How could it be fixed:
    - Use absolute links including http://... in the JSPs
    - Write a servlet filter that redirects the browser to http://... (out of scope)

Marc Rennhard, 04.06.2014, SSI_JavaWebAppSecurity2.pptx 38

HTTPS to HTTP Downgrading

The idea is to implement a filter that intercepts every request and determines programmatically whether the target resource is guarded by a transport guarantee of CONFIDENTIAL. If it is not, and the request came in over HTTPS, the filter will issue a redirect to the target resource over HTTP. See *http://blogs.oracle.com/jluehe/entry/how_to_downshift_from_https* for more details.

Specifying the Protocol in the JSP?

This is possible by including the full absolute URL, e.g. <c:url value='http://ubuntu.dev:8080/marketplace/cart' />. But hard-coding hostnames and the name of the application (marketplace) in the code is not desirable.

Session Handling

The extensions of this section are integrated in marketplace_v04.

Cookies (1)

- Cookies were introduced to solve the session handling problem in the Web as HTTP itself is stateless
- Cookies are set by the web application using the Set-Cookie HTTP response header and are stored in the browser
- Whenever the browser sends an HTTP request to a web application from which it has received a cookie before, the cookie is included in the HTTP request (Cookie header)
- Cookies can not only be used for session tracking but also to re-recognize users during subsequent visits (marketing reasons)

Cookies (2)

- Example cookie set by a web application:

```
Set-Cookie: cookie-identifier; expires=Fri, 23-Dec-2017 11:09:37 GMT;
Domain=ubuntu.dev; Path=/marketplace; Secure; HttpOnly;
```

- Explanation:
  - cookie-identifier: String that unambiguously identifies this cookie
    - Typically one or more name/value pairs (e.g. JSESSIONID=9F4BD...)
    - This identifier is sent back by the browser to the web application (HTTP request Cookie header)
    - Used by the web application to track the session or re-recognize the user
  - expires: Cookies with an expiry data are kept until this date even after closing the browser (persistent cookies), if no expiry date is used, the cookie is deleted when closing the browser (session cookies)
  - Domain, Path: Specifies when to send the cookie to the web application, here: Any request to resources below ubuntu.dev/marketplace/
  - Secure flag: Only send the cookie across HTTPS
  - HttpOnly flag: JavaScript cannot access the cookie (via document.cookie)

Cookies (3)

Some best practices with respect to cookie usage:

- Use them only if necessary, and think twice before using them beyond pure session tracking purposes
  - Cookies have a somewhat bad reputation due to significant tracking abuse years ago (before the times of privacy policies)
- Never use persistent cookies for authentication purposes
  - Don't offer "easy log in functionality" based on a previously stored persistent cookie
  - They will remain on the disk and are easily accessible by intruders
- Use the Secure flag
  - A cookie that was set over HTTPS will never be sent back over HTTP (so such a cookie can never be sniffed)
- Use the HttpOnly flag
  - Helps to protect from XSS-based cookie stealing attacks

Session Handling

Session handling is critical in web applications and a lot can go wrong:

- Session IDs are not picked randomly
  - Allows an adversary to guess it and hijack the session
- Session IDs are not changed when switching to HTTPS and logging in
  - Allows an adversary to read a session ID when the user uses HTTP and hijack the session after the user has authenticated
- When having an authenticated session over HTTPS and accessing a resource over HTTP, the authenticated session ID is sent over HTTP
  - Allows an adversary to read the authenticated session ID over HTTP and hijack (the still authenticated) session
- The cookies used for the session ID do not use the HttpOnly flag
  - Allows reading the cookie with JavaScript (as part of an XSS attack)

Good news: When using Java EE with Tomcat, this all works quite well "out of the box"

OWASP Top Ten

These session handling issues are all related to OWASP Top Ten Nr. 2: "Broken Authentication and Session Management" – so it indeed appears to be a critical issue!

Session Handling in Java EE / Tomcat (1)

- The name of the cookie used for session handling is JSESSIONID
- The session ID has a length of 16 bytes and can be assumed to be "random" (at least, no vulnerabilities have been reported)
- Connecting to the Marketplace application sets a cookie as follows:
  - Set-Cookie: JSESSIONID=9F4BD25B00CB5F385CB273DF92DB6BEB; Path=/marketplace; HttpOnly
  - HttpOnly flag is used, which is good
  - No Secure flag is used, because the cookie is set over HTTP
- When accessing the checkout page and switching to HTTPS, the session ID is not changed
  - The browser continues to include the session ID in the requests
  - This is fine as there is no reason to change the session ID as long as the user does not authenticate himself

Secure Flag

A cookie that is set with the secure flag will only be sent over HTTPS by the browser and never over HTTP.

Session Handling in Java EE / Tomcat (2)

- When accessing the admin area, the session ID is changed
  - Set-Cookie: JSESSIONID=7F45EB00D3F5B66641CEDF95913B8A53; Path=/marketplace; Secure; HttpOnly
  - Using the Secure flag is reasonable because if the cookie is set over HTTPS, we likely don't want that the browser sends it over HTTP later
  - But: it's not clear why a new session ID is set at all – there's no apparent threat that is prevented with this practice
  - Tomcat always sets a new cookie when accessing a resource that is protected with an `<auth-constraint>`

- When logging in, the session ID is always changed
  - Set-Cookie: JSESSIONID=240243EF765BA71A8BCFADB6D1CC565B; Path=/marketplace; Secure; HttpOnly
  - Unlike the session ID switch above, this is paramount
  - Otherwise, an attacker who could sniff the previous session ID over HTTP could use it to hijack the authenticated session, which is good

Session Handling in Java EE / Tomcat (3)

- For the web application, it's always the same session, but it's identified with a different session ID

- Important: the session ID is only changed if one uses the official declarative Java EE login and access control mechanisms as discussed before
  - `<security-constraint>`, `<auth-constraint>`, `<login-config>`
  - If one uses an own login approach with own access control handling, Java EE won't recognize logins and won't change the session ID
  - In this case, you have to make sure the session ID is changed programmatically

Session Handling in Java EE / Tomcat (4)

- When logging out the session ID is changed again
- The reason is that we use the `invalidate()` method
  - Which removes the current authenticated session and creates a new one with a new session ID
  - Since the authenticated session no longer exists, it can no longer be hijacked
- When having an authenticated session over HTTPS and accessing a resource over HTTP, the browser does not include the session ID
  - Because it was set with the Secure flag, so it must not be transmitted over HTTP
  - As the web application does not receive the session ID, it creates a new session with a new session ID
  - This new session is completely unrelated to the authenticated one, so sniffing the session ID won't help the attacker in any way

HTTPOnly Flag in Java EE / Tomcat

- The HttpOnly flag is only supported since Tomcat 6.0.19
- It can be disabled in context.xml:
  - Global or application-specific context.xml

```
<Context useHttpOnly="false">
  ...
</Context>
```

- Accessing the session ID via JavaScript is then possible:
  - E.g. using the vulnerability with the product search in the first Marketplace version

- The session ID cookie is now set as follows:
  - Set-Cookie: JSESSIONID=7F45EB00D3F5B66641CEDF95913B8A53; Path=/marketplace; (Secure)
- Don't disable it, there's no point in doing so...

Accessing the Cookies

Accessing the cookies (and therefore also the session ID) with JavaScript is easy, e.g. `<script>alert(document.cookie);</script>`.

`<cookie-config>` since Java EE 6 / Tomcat 7

- Since Java EE 6, a new `<cookie-config>` element in the `<session-config>` element is supported
  - Allows specifying the usage of the Secure and HttpOnly flags in web.xml
  - Enabling these flags in web.xml means they will be used independent of the actual settings of the underlying Servlet/JSP engine

```
<session-config>
  <session-timeout>10</session-timeout>
  <cookie-config>
    <secure>true</secure>
    <http-only>true</http-only>
  </cookie-config>
</session-config>
```

Access Control – Again

The extensions of this section are integrated in marketplace_v05.

More Access Control

- We have added access control mechanisms to protect access to the admin area
- However, there are other resources which should not be accessed directly by the user
  - Especially JSPs that should only invoked by servlets
- Preventing direct access to these resources usually increases the robustness of web applications
  - Mainly because developers place validation mechanisms in the servlets
    - E.g. whether there are products in the cart before going to the checkout
  - Calling the JSPs directly circumvents these checks and may result in erroneous application behavior
- There are two well-established methods to achieve this:
  - Place the not-directly-accessible resources below the WEB-INF directory
  - Or define security-constraints in web.xml to prevent access

WEB-INF Directory

The WEB-INF directory (and any content in it) cannot be accessed from the outside (by the browser).

Circumventing a Servlet

- Example: The user checks out with no products in the cart
- Clicking the button – which calls the servlet – behaves as intended
- But calling the JSP directly works – and even circumvents HTTPS!
  - As access to checkout.jsp is not restricted by a security constraint

Prevent Access using WEB-INF Directory

- The only directly available JSP is index.jsp, so it remains where it is
- Anything else should not be accessed directly and is therefore placed in a directory pages below WEB-INF
- Of course, one then also has to adapt some servlets to use the correct paths, e.g.

```
// Forward to JSP
String url = "/WEB-INF/pages/products.jsp";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
dispatcher.forward(request, response);
```

- Also, adapt the login and login error pages in web.xml

```
<form-login-page>
  /WEB-INF/pages/admin/login.jsp
</form-login-page>
<form-error-page>
  /WEB-INF/pages/admin/login_error.jsp
</form-error-page>
```

Directory pages

This directory would not be needed, so the JSPs could be placed directly below WEB-INF, but using a directory is certainly cleaner from an organizational point of view.

Prevent Access using Security Constraints

- index.jsp again remains where it is
- Anything else is placed in a directory pages
- Configure web.xml to prevent any access to the pages directory

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Non-accessible
    </web-resource-name>
    <url-pattern>/pages/*</url-pattern>
  </web-resource-collection>
  <auth-constraint />
</security-constraint>
```

Specifying an empty auth-constraint element is important!
- It specifies the roles that are allowed access, so an empty list means no access is possible at all
- Not using the auth-constraint element would mean that every user gets access to the resource

- Like before, adapt the paths where necessary

`<auth-constraint>`

An auth-constraint without any roles (as above) means that accessing the resource is not possible at all. It the tag were not used, access would not be restricted at all, i.e. every user/role (authenticated or not) would be granted access! We have used that case before, when enforcing HTTPS

Below WEB-INF or Security Constraints?

- Basically, the result is the same in both cases, with one exception
- Trying to access a resource below WEB-INF creates a 404 error

- Accessing a resource protected with a security constraint creates a 403 error

- Security-wise, both approaches are equally suited
  - Using standard error messages, a user/attacker won't be able to distinguish the two approaches
- We prefer using the WEB-INF variant because:
  - It does not require configuring a security constraint, so it's a bit less work
  - It fits better with access control annotations, see later

Annotations

As we will see later, one can use access control annotation to control access to servlets – but not to JSPs. As a result, the second variant with a security-constraint cannot be "translated" to annotations, which means that a security constraint would still be needed to protect access to the JSPs.

Further Elements of `<security-constraint>`

- Restrict a security-constraint to some HTTP methods

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Restricted area</web-resource-name>
    <url-pattern>/secret/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
```

Constraint only applies to GET and POST requests

- Note that this means that there are no constraints for the other HTTP methods, i.e. they can access the resource with no restrictions!
  - If a constraint should apply to all methods, don't use `<http-method>`
- Since Java EE 6, an additional element is supported to apply a constraint to the methods that are NOT specified

```
<http-method-omission>GET</http-method-omission>
```


Multiple Security Constraints

When multiple constraints cover exactly the same <url-pattern>, the constraints are combined. This allows to specify basically any rules one can imagine. For details, refer to the Java Servlet Specification.

<http-method> and <http-method-omission>

http-method or http-method-omission is used to specify which methods should be protected or which methods should be omitted from protection. An HTTP method is protected by a web resource-collection under any of the following circumstances:

- If no HTTP methods are named in the collection (which means that all are protected)
- If the collection specifically names the HTTP method in an http-method subelement
- If the collection contains one or more http-method-omission elements, none of which names the HTTP method

Source: Java EE 7 Tutorial, http://download.oracle.com/javaee/7/tutorial/doc/

<http-method-omission>

In this case, the security constraints applies to all methods except the ones that are listed with http-method-omission elements. Note that one cannot use both http-method http-method-omission at the same time in a security constraint.

Restrict General Access to GET and POST

- Usually, web applications only use GET and POST in requests
  - From a security point of view, its reasonable to prevent all other requests such as HEAD, PUT, DELETE, TRACE, OPTIONS
- To disable all other methods, we put the following in our web-xml:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>restricted methods</web-resource-name>
    <url-pattern>/*</url-pattern>
    <url-pattern>/admin/*</url-pattern>
    <url-pattern>/checkout</url-pattern>
    <url-pattern>/purchase</url-pattern>
    <http-method-omission>GET</http-method-omission>
    <http-method-omission>POST</http-method-omission>
  </web-resource-collection>
  <auth-constraint />
</security-constraint>
```

Applies to all methods except GET and POST

No access is possible at all

Why isn't /* enough to apply this constraint to all resources?
- When accessing a resource, only the constraints with the best match are considered
- So when accessing a resource in /admin/*, this security constraint would be omitted
- To make sure this constraint is always considered, one must include all url-patterns for which other security constraints are configured

Marc Rennhard, 04.06.2014, SSI_JavaWebAppSecurity2.pptx 57

Disabling HTTP Methods in Tomcat

Tomcat does not offer a configuration option to disable specific HTTP method (other servlet/JSP engine may offer this, though). Therefore, the only way to do this is in web.xml.

PUT, DELETE etc. in Servlets?

To handle other requests than GET and POST, the HTTPServlet class offers methods doPut, doDelete etc. They are usually not used in most applications, but they are supported. To make sure they are not used when they are not desired, deactivate them in web.xml as described above. This can prevent possible security problems when these requests are used, e.g. because a programmer tried to include a backdoor (or a "feature") by using HTTP requests other than GET or POST or because a programmer who accidently included processing such HTTP requests introduced a vulnerability.

Java EE 5 / Servlet 2.5

Without `<http-method-omission>`, this can also be configured as follows:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>restricted methods</web-resource-name>
    <url-pattern>/*</url-pattern>
    <url-pattern>/admin/*</url-pattern>
    <url-pattern>/checkout</url-pattern>
    <url-pattern>/purchase</url-pattern>
    <http-method>HEAD</http-method>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    <http-method>TRACE</http-method>
    <http-method>OPTIONS</http-method>
  </web-resource-collection>
  <auth-constraint />
</security-constraint>
```

<deny-uncovered-http-methods> in Java EE 7

- Java EE 7 introduces <deny-uncovered-http-methods> that can be specified directly below <web-app> in web.xml
- It denies any HTTP methods that are not explicitly allowed in a <security-constraint>

```
<security-constraint>
  <deny-uncovered-http-methods/>
  <web-resource-collection>
    <web-resource-name>Restricted area</web-resource-name>
    <url-pattern>/secret/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
```

No other method than GET or POST can be used to access /secret/*

Security Constraints only affect Requests from Clients

- When the browser sends an HTTP request, then the matching security constraints specified in the deployment descriptor are applied
- However, they do not apply to application-internal forwarding, e.g. from a servlet to a JSP or a servlet to a servlet
- Example
  - If this forwarding is done by a servlet in the Marketplace application…

```
// Forward to servlet
String url = "/admin/completepurchase";
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher(url);
dispatcher.forward(request, response);
```

  - …then access to /admin/completepurchase will always be allowed – no matter whether the user is logged in and what roles he has
- So during application development, keep in mind that internal forwarding is not restricted by declarative security measures!

Unrestricted Application-Internal Forwarding

In the example above, one would have to make sure that this forwarding is only used by a servlet that is also below /admin/, i.e. that is protected by the the same security constraints as /admin/completepurchase.

Preventing Access to JSPs with a Security Constraints

Remember that we have shown how a security constraint can be used to restrict access to JSPs? This only affects direct access to the resources (from the “outside”). But within the servlets, forwarding to these JSPs is (and must be) possible, so one can see that the security constraints do not affect application-internal forwarding. Likewise, internal forwarding to resources in the WEB-INF directory is always possible, only access from the outside is not possible.