

8. Java Web Application Security – Part 3

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

Programmatic Security Features in Java EE

Programmatic Security Features in Java EE

- Java EE also offers some **programmatic security features**
- They are **based on the declarative security mechanisms**
 - Which means they can only be used if one uses declarative security features (<security-constraint>, <login-config> etc.)
- Up to **Java EE 5**, this consists of the following methods of the **HttpServletRequest** interface:
 - **String getRemoteUser()**: Returns the user name of the current user
 - **boolean isUserInRole(String role)**: Returns whether the current user is logged in and has the specified role
 - **Principal getUserPrincipal()**: Returns an object that contains name and roles of the current user
- This provides the basis to make decisions about logged in users and their roles **in the program code**

Marketplace – Adding Programmatic Security (1)

To demonstrate this, we extend the Marketplace application:

- Roles marketing and sales still have access to the admin area
- But the **marketing** role can only see the purchases while **sales** people can also complete purchases
- 1st step: ListPurchaseServlet.java: **Set a request attribute** that specifies whether the user is allowed to complete purchases:

```
// Check if the user is allowed to complete purchases
if (request.isUserInRole("sales")) {
    request.setAttribute("completeAllowed", true);
} else {
    request.setAttribute("completeAllowed", false);
}
```

Marketplace – Adding Programmatic Security (2)

- 2nd step: In the subsequent `purchases.jsp`, generate the table based on the request attribute

```
<table cellpadding="5" border=1>
  <tr valign="bottom">
    <td align="left"><b>First Name</b></td>
    ...
    <td align="left"><b>Total Price</b></td>
    <c:if test="${completeAllowed}">
      <td align="left"></td>
    </c:if>
  </tr>

  <c:forEach var="item" items="${purchases}">
    <tr valign="top">
      <td><c:out value="${item.firstName}" /></td>
      ...
      <td><c:out value="${item.totalPriceCurrencyFormat}" /></td>
      <c:if test="${completeAllowed}">
        <td><c:url var="url"
          value="/admin/completepurchase?purchaseId=${item.id}" />
        <a href="<c:out value="${url}" />">Complete Purchase</a></td>
      </c:if>
    </tr>
  </c:forEach>
</table>
```

Only insert the rightmost column if the user is allowed to complete purchases

Marketplace – Adding Programmatic Security (3)

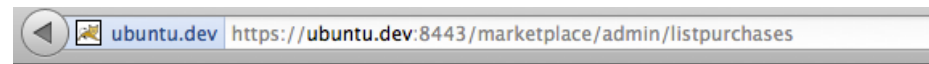
- That's it? Nope!
 - Even without the visible "link in the table" one can still manually submit the corresponding request
- Marketing people can therefore still complete a purchase by submitting the correct request manually (**forced browsing**)
 - <https://ubuntu.dev:8443/marketplace/admin/completepurchase?purchaseId=523>
- To prevent this, we must **check permissions again in DeletePurchaseServlet.java** (which performs the actual completion):

```
// Check the rights of the user
if (!request.isUserInRole("sales")) {
    message = "You are not allowed to perform this operation";
} else {

    // Delete purchase...
}
request.setAttribute("message", message);
```

Marketplace – Adding Programmatic Security (4)

- Login as **alice** (sales)

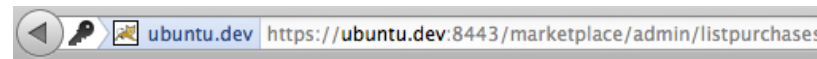


Purchases

First Name	Last Name	CC Number	Total Price	
Script	Lover	5555 6666 7777 8888	\$10.95	Complete Purchase
I buy	everything	0000 0000 9999 9999	\$250,461.85	Complete Purchase

[Return to search page](#)

[Logout and return to search page](#)



Purchases

First Name	Last Name	CC Number	Total Price
Script	Lover	5555 6666 7777 8888	\$10.95
I buy	everything	0000 0000 9999 9999	\$250,461.85

[Return to search page](#)

[Logout and return to search page](#)

Purchases

You are not allowed to perform this operation

First Name	Last Name	CC Number	Total Price
Script	Lover	5555 6666 7777 8888	\$10.95
I buy	everything	0000 0000 9999 9999	\$250,461.85

[Return to search page](#)

[Logout and return to search page](#)

- Login as **robin** (marketing)...
and completion attempt

Programmatic Security – Discussion (1)

- As you have seen, programmatic security can be added **where pure declarative security is not enough**
 - E.g. to adapt a web page depending on the user's rights
- But you have also seen it's **more complex than declarative security**
 - We did not only make sure the table with the purchases is presented correctly in the first step
 - But we also had to check whether the user has the rights to complete a purchase in the second step

Programmatic Security – Discussion (2)

- The correct approach to implement programmatic access control checks is **complete mediation**:
 - For **every single action**, check whether the user has the necessary rights
 - This is **often done incorrectly** in web applications especially if access control checks are done completely programmatically
- **Vulnerable applications** often show the following behavior:
 - Assume the application has a **protected area** with restricted access
 - Often, an access control check is done when “**entering**” a protected area
 - In the protected area, the user has access to **links to perform further actions**
 - When performing these actions, no additional check is done as the **developer (wrongly) assumes** these actions can only be done when having access to the links in the protected area, i.e. when having “passed” the first check
 - This assumption is wrong, as a **user / attacker can always perform any action** provided he knows the correct request

Programmatic Security – Discussion (3)

- We could have solved this **with declarative security only**
 - Have two different admin areas /admin/sales/ and /admin/marketing/ that are accessible by the respective roles
 - Have appropriate servlets and JSPs for the two areas
 - But this also means more code and duplicate code (the servlets and JSPs vary only little)
- In this example, using programmatic security was a **reasonable decision**, because:
 - Implementing this was easy, only **little additional code** was required
 - If you know that access rights must be checked with every action performed by the user, it's not really difficult to implement this in a **secure way**
- But in general, **don't use programmatic access control checks more than necessary** and use declarative security wherever reasonable

Programmatic Security Features in Java EE

Since [Java EE 6](#), 3 further methods for programmatic security are provided, also part of the [HttpServletRequest](#) interface:

- `boolean authenticate(HttpServletRequest response)`
 - Initiates a user authentication using the configured mechanism in web.xml (<login-config>, e.g. BASIC or FORM authentication)
 - Allows to start authentication programmatically anytime, independent of any security constraints in web.xml
- `void login(String username, String password)`
 - Performs a login using the configured Realm (e.g. JDBCRealm)
 - Provides the basis to implement alternative authentication mechanisms than what is provided by Java EE
- `void logout()`
 - Performs a logout, the authenticated user is “forgotten” by the web application
 - In contrast to `HttpSession.invalidate()`, other session attributes won't be affected, e.g. the cart would be preserved

Marketplace – Custom Login (1)

- So far, the Marketplace application only uses hashed passwords but no **salt** – we will change this by using the `login` method
- We add two new columns to the **UserPass table**: Salt and Digest2

Username	Salt	Digest2
alice	c5402e5139e8634f	5115db008bb28afe8bd333e38e4148dc43b2a8f0a6787c9e74891bc47
john	a83b0197cf24cb63	fff9fd19eb27e2ef6fcc9bce286751ed9bf827c788f287ab164172dd302f
robin	33c043d560d059a8	7d9298e7e38598696f71d1a2907199f26fc550377c83d554e6c39c358

- Salt is a 64 bit value, stored in hex format
- **During login**, the following should happen:
 - The user submits username and password
 - The salt of the user is retrieved from the UserPass table
 - The digest is computed as `SHA-256(password|salt)`
 - If the computed digest matches the digest in the table, login is successful

Marketplace – Custom Login (2)

- So far, we used **declarative security to handle logins**
 - Can easily be used by configuring the `<login-config>` element in web.xml
 - But this only supports plaintext or directly hashed password
- To support salt to compute password hashes, we have to perform login programmatically
 - We therefore **remove the `<login-config>` element** from web.xml

```
<login-config>  
  <auth-method>FORM</auth-method>  
  <form-login-config>  
    <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>  
  </form-login-config>  
</login-config>
```



Marketplace – Custom Login (3)

- We also adapt the **JDBCRealm configuration** in context.xml
 - Tomcat should no longer perform any digest computation (we do that programmatically in the code including the salt)
 - The new column (Digest2) is used to check the correctness of the credentials

```
<Realm className="org.apache.catalina.realm.JDBCRealm" digest="SHA"  
  driverName="com.mysql.jdbc.Driver"  
  connectionURL="jdbc:mysql://localhost:3306/marketplace"  
  connectionName="marketplace" connectionPassword="marketplace"  
  userTable="UserPass" userNameCol="Username" userCredCol="Digest2"  
  userRoleTable="UserRole" roleNameCol="Rolename" />
```

- When calling `login(String username, String password)`, the following happens:
 - Tomcat checks whether the table **UserPass** contains an entry with the passed username / password
 - Columns **Username** and **Digest2** are used to perform the check
 - Which means **we must pass the hash** (computed over password|salt) as the second parameter

Marketplace – Custom Login (4)

- Until now, the  button directly called `ListPurchasesServlet`
 - As we used declarative security to handle logins (`<login-config>`), the user was automatically redirected to the login form when needed
- As we handle the login programmatically now, this no longer works
 - Clicking  therefore calls a `new LoginServlet`
- We therefore change the action of the button

```
<form action="<c:url value='/admin/login' />" method="get">
  <input type="submit" value="Admin area">
</form>
```

- `Servlet and mapping` in `web.xml`

```
<servlet>
  <servlet-name>LoginServlet</servlet-name>
  <servlet-class>servlets.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>LoginServlet</servlet-name>
  <url-pattern>/admin/login</url-pattern>
</servlet-mapping>
```

Marketplace – Custom Login (5) – LoginServlet.java (part 1)

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

    // Check if the request contains parameters
    String url = null;
    if (request.getParameterNames().hasMoreElements() == false) {

        // If there are no parameters, the request comes from clicking
        // the "Admin area" button. Check if the user is already logged in.
        if (request.isUserInRole("sales") || request.isUserInRole("marketing")) {

            // Logged in, Forward to servlet
            url = "/admin/listpurchases";
        } else {

            // Not logged in, forward to login form
            url = "/WEB-INF/pages/admin/login.jsp";
        }
        RequestDispatcher dispatcher =
            getServletContext().getRequestDispatcher(url);
        dispatcher.forward(request, response);
    }
}
```

The user is not logged in
(or has at least not the
right role), so forward to
the login form

Marketplace – Custom Login (6) – Login Form

- This form is basically the same as before, but as we do the login programmatically, we **no longer use the identifiers** j_security_check, j_username, and j_password

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

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

<form action="<c:url value=' /admin/login' />" method="get">
<table cellpadding="5" border="0">
  <tr>
    <td align="right">Username</td>
    <td><input type="text" name="username"></td>
  </tr>
  <tr>
    <td align="right">Password</td>
    <td><input type="password" name="password"></td>
  </tr>
  <tr>
    <td><input type="submit" value="Login"></td>
  </tr>
</table>
</form>
```

Send the inserted values
to the LoginServlet

Marketplace – Custom Login (7) – LoginDB

- We must **read the salt of a user from the DB**, this done with the method `getSalt` in a new class `LoginDB`

```
public static String getSalt(String username) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;

    String query = "SELECT * FROM UserPass WHERE Username = ?";
    try {
        ps = connection.prepareStatement(query);
        ps.setString(1, username);
        rs = ps.executeQuery();
        if (rs.next()) {
            return rs.getString("Salt");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        DBUtil.closeResultSet(rs);
        DBUtil.closePreparedStatement(ps);
        pool.freeConnection(connection);
    }
    return "";
}
```

Marketplace – Custom Login (8) – LoginServlet.java (part 2)

```
} else {  
  
    // Parameters have been submitted, the request comes from the  
    // admin form. Check submitted username and password  
    String username = request.getParameter("username");  
    String password = request.getParameter("password");  
  
    // Authenticate the user  
    try {  
  
        // Read the salt from the DB and compute the digest  
        String salt = LoginDB.getSalt(username);  
        String digest = Crypto.printHex(Crypto.computeSHA256(password + salt));  
        request.login(username, digest);  
        url = "/admin/listpurchases";  
    } catch (Exception e) {  
        request.setAttribute("message", "Login failed.");  
        url = "/index.jsp";  
    }  
    RequestDispatcher dispatcher =  
        getServletContext().getRequestDispatcher(url);  
    dispatcher.forward(request, response);  
}  
}
```

Perform the actual login, which results in checking whether an entry with username and "password" digest exists in the UserPass table (according to the JDBCRealm configuration in context.xml). If the login fails, an Exception is thrown.

Marketplace – Custom Login (9)

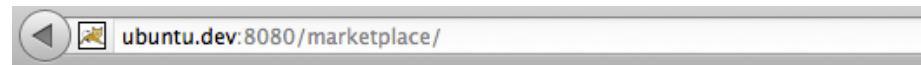


- One detail remains:
 - The [LoginServlet](#) is reached via [/admin/login](#)
 - But access to [/admin/*](#) is only allow for authenticated user – chicken-egg-problem?
- This can easily be solved: Add a security constraint that [explicitly allows access to /admin/login](#)
 - This wont affect any other resources below [/admin/*](#)
 - There's no `<auth-constraint>` element, so any user has access
 - When accessing [/admin/login](#), this won't be combined with the existing [/admin/*](#) constraint, as the url-pattern in this constraint is the "better match"

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Login page</web-resource-name>
    <url-pattern>/admin/login</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Marketplace – Custom Login (10)

- Testing with user john works as expected:



Welcome to the Marketplace

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

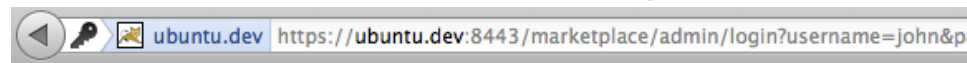
 

Admin Login Form

Please enter your username and password to continue.

Username

Password



Purchases

First Name	Last Name	CC Number	Total Price	
Script	Lover	5555 6666 7777 8888	\$10.95	Complete Purchase
I buy	everything	0000 0000 9999 9999	\$250,461.85	Complete Purchase

Annotations in Java EE 6

Annotations in Java EE 6

- Since Java EE 5 **annotations** are supported
- This was extended in Java EE 6
 - In particular, **access control annotations** were added, which can be used instead of `<security-constraint>` in `web.xml`
- Today, using security constraints in `web.xml` still dominates in practice
 - But using access control annotation has its advantages and is **likely to become the future best practice**
- Note that the annotations **do not introduce any new functionality**, it's just another way to do the same
 - And not everything that you can do with `web.xml` can also be done with annotations (e.g. `<login-config>`, `<session-config>`...)
- There are other **(non security-related) annotations** that can be used to replace `web.xml` elements
 - E.g. `<servlet>`, `<servlet-mapping>`, `<filter>`...
 - Here, we focus on the access control annotations

Access Control Annotations

There are 3 access control annotations:

- **@ServletSecurity** with the following elements

- **value**: an @HttpConstraint that defines the access rights for all HTTP methods that are NOT included in httpMethodConstraints
- **httpMethodConstraints**: an array of @HttpMethodConstraint, each of which defines the access rights for a specific HTTP method

- **@HttpConstraint** with the following elements

- **value**: the authorization if rolesAllowed is not used (PERMIT or DENY)
- **rolesAllowed**: the authorized roles (other roles have no access)
- **transportGuarantee**: the data protection requirements (HTTPS)

- **@HttpMethodConstraint** with the following elements

- **value**: the HTTP method (e.g. GET or POST)
- **emptyRoleSemantic**: the authorization if rolesAllowed is not used (PERMIT or DENY)
- **rolesAllowed**: the authorized roles (other roles have no access)
- **transportGuarantee**: the data protection requirements (HTTPS)

Marketplace – with Annotations (1)

- We first look at the **servlets that don't have specific access control restrictions** (besides allowing only GET and POST)
 - ListProductsServlet and CartServlet

```
@ServletSecurity(  
    value = @HttpConstraint(value = EmptyRoleSemantic.DENY),  
    httpMethodConstraints = {@HttpMethodConstraint(value = "GET"),  
                             @HttpMethodConstraint(value = "POST")})  
public class ListProductsServlet extends HttpServlet {...}
```

- Explanation:
 - The two **@HttpMethodConstraint** annotations **allow access with GET and POST** requests for everyone over HTTP
 - rolesAllowed is not used, so everyone is allowed access
 - transportGuarantee is not used, so access with HTTP is allowed
 - emptyRoleSemantic is not used, so the default value PERMIT is used
 - The **@HttpConstraint** annotation **denies access for all other methods**
 - value is set to DENY, which denies access to all users

Marketplace – with Annotations (2)

- Now, we look at the servlets that are used to **make a purchase and to log in**
 - CheckoutServlet, PurchaseServlet and LoginServlet
 - Everyone has access to these resources using GET and POST, but in addition, **HTTPS must be used**
- This can be configured by additionally specifying the **transportGuarantee** element in @HttpMethodConstraint

```
@ServletSecurity(  
    value = @HttpConstraint(value = EmptyRoleSemantic.DENY),  
    httpMethodConstraints = {  
        @HttpMethodConstraint(value = "GET",  
            transportGuarantee = TransportGuarantee.CONFIDENTIAL),  
        @HttpMethodConstraint(value = "POST",  
            transportGuarantee = TransportGuarantee.CONFIDENTIAL)})  
public class CheckoutServlet extends HttpServlet {...}
```

Marketplace – with Annotations (3)

- Finally, we look at the servlets that are only accessible by users with roles **marketing or sales** (and only over HTTPS)
 - DeletePurchaseServlet, ListPurchasesServlet and LogoutServlet
- This can be configured by additionally specifying the **rolesAllowed** element in @HttpMethodConstraint

```
@ServletSecurity(  
    value = @HttpConstraint(value = EmptyRoleSemantic.DENY),  
    httpMethodConstraints = {  
        @HttpMethodConstraint(value = "GET",  
            rolesAllowed = {"marketing", "sales"},  
            transportGuarantee = TransportGuarantee.CONFIDENTIAL),  
        @HttpMethodConstraint(value = "POST",  
            rolesAllowed = {"marketing", "sales"},  
            transportGuarantee = TransportGuarantee.CONFIDENTIAL)})  
public class DeletePurchaseServlet extends HttpServlet {...}
```

Marketplace – with Annotations (4)

- Note that **ALL** `<security-constraints>` can now be removed from `web.xml` – as we have everything covered by annotations
- The application behaves exactly the same as before
- One can certainly say that **using the annotations is easier** than having (possibly overlapping) security constraints in `web.xml`
 - Especially as the restrictions for a particular servlet are configured “at one place” – in the servlet
- But there are also **drawbacks**:
 - Annotations must be **duplicated** across servlets with the same restrictions (which is cumbersome when something must be changed)
 - The probability that something “is forgotten” is likely to be higher than when configurations take place at a central place and for entire directories (e.g. `/admin/*`)

Marketplace – with Annotations (5)

One final remark:

- Access control annotations **can only be used to protect access to servlets**, but not to JSPs
- In a “real” servlet/JSP-based application, this is not a problem as the **browser primarily calls servlets**
- If you must control access to JSPs (or other resources such as HTML pages), this can still be done with **security constraints** in web.xml
 - One reason why we placed the JSPs that should not be directly accessible below WEB-INF is that now – having switched to annotations – we don’t need security constraints at all

Input Validation with ESAPI

Input Validation

- Input validation is important for web applications as it helps **protecting from several attacks** (injection attacks, XSS, parameter tampering...)
 - But it's also important for functional aspects to make sure, e.g., that users only submit correct credit card numbers or e-mail addresses
- Java EE provides support to perform input validation
 - **Java Server Faces (JSF)**, which is part of Java EE, provides input validation functionality
 - Since Java EE 6, the **Bean Validation Framework** is provided, which allows specifying validation rules in managed beans
 - See the appendix to learn more about them
- But both technologies cannot be used with servlets/JSPs
 - We therefore have two choices: Implement our **homegrown approach** or use a **3rd party input validation library**
 - As we shouldn't reinvent the wheel, we choose the 2nd approach

Input Validation Libraries

Several libraries have been developed to specifically support input validation, some examples:

- **Apache Commons Validator**
 - Originally developed as an add-on to Struts, but can be used with any Java web application
 - Easy to use, popular, but no new release since 2006
- **OWASP Stinger**
 - Works as servlet filter, configurable via XML files
 - No new version since 2009, project appears to be dead
- **OWASP Enterprise Security API (ESAPI)**
 - A web application security library that offers much more than “just” input validation (e.g. data sanitation, CSRF prevention)
 - Intended for many platforms beyond Java (.NET, PHP...), but Java is by far most mature
 - Frequently used and is still actively developed further

Input Validation and Normalization (1)

- One important aspect of input validation is **normalization**
- Normalization means that any data that is analyzed **should first be decoded into it's most basic form** (i.e. "text", encoded as ASCII or Unicode)
- **Why is this important?**
 - Assume you write a (naïve) filter that tries to detect XSS attacks
 - You scan received user data for occurrences of the string <script>
 - If the attacker encodes (parts of) his injected script, the filter most likely fails → evading input validation is relatively easily possible
- Of course, **for the attack to be successful**, the following would have to happen:
 - The web application decodes the encoded string before sending it to the browser or the browser of the victim interprets the encoded text as a script
 - Both has happened before and corresponding attacks have occurred

Input Validation and Normalization (2)

- As an example, we use **different valid hex or decimal encodings** of the string `<script>`
 - `%3c%73%63%72%69%70%74%3e`
 - `<script>`
 - `\x3c\x73\x63\x72\x69\x70\x74\x3e`
 - `<script>`
 - → Unless these strings are first decoded, there's no way to detect they all represent the string `<script>`
- One can also do **double / multiple encoding**
 - `%25%33%63%25%37%33%25%36%33%25%37%32%25%36%39%25%37%30%25%37%34%25%33%65`
 - Decoding this leads to `%3c%73%63%72%69%70%74%3e`
 - If a filter does not check the resulting string again, it again won't detect that this represents the string `<script>`

Input Validation and Normalization (3)

- In general, **don't try to think about which encoding may be dangerous**
- The reason is that there's a **wide range of programs** (browsers, mobile apps...) that may be used to access your application
 - And you usually don't know exactly how all these programs interpret the encoded data
 - **Future features / bugs** of these programs may "transform" an encoded string – which so far was harmless – a problematic one
- The **best strategy** to avoid problems is therefore to always use normalization to decode the received data into its most basic form
 - Apply filtering always to the decoded data
- Once the data has been decoded, keep it decoded (e.g. don't store the encoded version in the DB)
 - This minimizes work for possible further filters (e.g. data sanitation) as normalization must not be repeated

Whitelisting

- With input validation, experience has shown that the **whitelisting approach** is best suited to perform the task correctly
- Whitelisting means: **you specify what is allowed for specific data**, e.g.:
 - Minimum and maximum number of characters
 - Allowed characters
- The opposite is **blacklisting**, e.g. defining what is forbidden
- **Whitelisting has several advantages:**
 - It is **more natural** because when asking yourself what input data is legitimate, you usually think about what characters are allowed
 - A name of a person has at most 30 characters and consists of letters...
 - It's usually **more secure** as with blacklisting, it's easy to forget something, which may have negative security impacts

OWASP ESAPI

- We only look at the **input validation and normalization features** of ESAPI here
- When performing input validation with ESAPI, **normalization is always done first** – unless you explicitly disable it
 - This means that you don't have to deal with normalization , it "simply happens"
 - ESAPI can deal with a wide range of common encodings
 - If you have special requirements, you can specify exactly the decoders you desire, but usually the default settings are well suited
- ESAPI offers some **methods to check some typical input formats**:
 - Date, file name, floating point number, credit card number...
 - In addition, you can do any custom filtering using the `getValidInput` methods of the `Validator` interface
 - Filtering is done with regular expressions

Marketplace – Input Validation (1)

- We extend the [Marketplace application](#) such that it provides input validation
- As an example, we do this on the [checkout page](#) → user must insert valid first and last names and a valid credit card number
- For usability reasons, it would [be reasonable to check the inserted data also in the browser \(using JavaScript\)](#) when the user clicks “Purchase”
 - This saves round-trips in case the user accidentally provided incorrect information
 - But we won’t do this here, as security-wise, this helps nothing at all as client-side filtering can easily be circumvented by an attacker

Checkout

Please insert the following information to complete your purchase:

First name:

Last name:

Credit card number:

Marketplace – Input Validation (2) – Validation.java (part 1)

- We write a Validation class that contains all validation methods used in the Marketplace application

```
package util;

import org.owasp.esapi.ESAPI;
import org.owasp.esapi.Validator;

public class Validation {

    public static String validatePersonName(String input) {
        Validator validator = ESAPI.validator();
        try {
            String clean = validator.getValidInput("Name", input.trim(),
                "PersonName", 32, false);

            return clean;
        } catch (Exception e) {
            return null;
        }
    }
}
```

Used ESAPI
classes

One usually uses the standard implementation of the Validator interface, which is provided by ESAPI

- This object will be used to perform all validation operations

Returns the normalized string of input if validation is successful or an exception otherwise

- "Name" is an identifier used in error messages produced during validation
- input.trim() removes leading and trailing spaces
- The regular expression defined as "PersonName" in ESAPI.properties is used to validate the input
- Max length after normalization is 32
- The input must not be null (false)

Marketplace – Input Validation (3) – Validation.java (part 2)

```
public static String validateCCNumber(String input) {  
  
    Validator validator = ESAPI.validator();  
    try {  
        String clean = validator.getValidCreditCard("CreditCard",  
                                                    input.trim(), false);  
        return clean;  
    } catch (Exception e) {  
        return null;  
    }  
  
    ...  
  
}
```

Returns the normalized string of input (trimmed) if validation is successful or an exception otherwise. The input must not be null (false).

Marketplace – Input Validation (4)

- In `validatePersonName`, we specified to use the regular expression identified as “**PersonName**” in the configuration file `ESAPI.properties`
- We **define a valid person name** as follows:
 - Minimum 2, maximum 32 characters
 - Allowed are lower- and uppercase letters and the single quote ('), as O’Neil and O’Connor want to shop as well
- Needed entry in **ESAPI.properties**:

```
Validator.PersonName=[a-zA-Z']{2,32}$
```

Marketplace – Input Validation (5) – PurchaseServlet.java

- Extend PurchaseServlet.java to validate all three fields:

```
// Get parameters from the request
String firstName = request.getParameter("firstName");
String lastName = request.getParameter("lastName");
String ccNumber = request.getParameter("ccNumber");

// Validate the received data
if ((firstName = Validation.validatePersonName(firstName)) == null) {
    message = "Please insert a valid first name";
    url = "/checkout";
} else if ((lastName = Validation.validatePersonName(lastName)) == null) {
    message = "Please insert a valid last name";
    url = "/checkout";
} else if ((ccNumber = Validation.validateCCNumber(ccNumber)) == null) {
    message = "Please insert a valid credit card number";
    url = "/checkout";
} else {

    // Store purchase in DB
```

If any validation fails, forward the user again to the checkout page and display an appropriate message

Marketplace – Input Validation (6)

- Testing works as expected:

First name:	<input "="" type="text" value="John="/>	→ Please insert a valid first name
Last name:	<input type="text" value="Wayne"/>	
Credit card number:	<input type="text" value="1111 2222 3333 4444"/>	
First name:	<input type="text" value="John"/>	<div style="border: 1px solid black; background-color: #e0ffe0; padding: 10px; text-align: center;"> This shows, that the credit card number not only has to contain 16 digits, but that the checksum is also tested! </div>
Last name:	<input type="text" value="Wayne"/>	
Credit card number:	<input type="text" value="1111 2222 3333 4445"/>	
		↑
		→ Please insert a valid credit card number
First name:	<input type="text" value="John"/>	→ Your purchase has been completed, thank you for shopping with us.
Last name:	<input type="text" value="O'Connor"/>	
Credit card number:	<input type="text" value="1111 2222 3333 4444"/>	

Normalization in Detail (1)

- To test normalization, we use **%4a** instead of J:

First name: → Please insert a valid first name

Last name:

Credit card number:

- Interestingly, %4a is not translated to J, but the **first name is rejected**
 - It appears that encoded strings are only accepted if the characters used for encoding (e.g. %) are allowed by the specified filter
- In fact, **validation happens as follows**:
 - If the submitted string does not match the regular expression, it is rejected
 - If it passes this test, the string is normalized
 - The normalized string is then again checked whether it matches the regular expression

So an encoded string can only pass validation if both the encoded and normalized string match the regular expression → This makes it “as difficult as possible” for an attacker to circumvent input validation, which is good!

Normalization in Detail (2)

- To see the actual effect of normalization, we adapt the regular expression for PersonName to **allow digits and the % character**

```
Validator.PersonName=^[a-zA-Z0-9'%]{2,32}$
```

- This time, normalization happens correctly, the first name is accepted and **the DB contains the first name as "standard text"**

First name:

Last name:

Credit card number:

→ Your purchase has been completed, thank you for shopping with us.

Purchases

First Name	Last Name	CC Number	Total Price	
John	Wayne	1111 2222 3333 4444	\$250,000.00	Complete Purchase

Input Validation / Normalization – Final Remarks

- You can argue that in our case, **normalization is not really needed** as only a few characters are permitted, which do not allow any encoding
- This is basically true, but the key is that you **shouldn't think too much** about what encodings may be possible with the allowed character set
 - It's very easy to miss something – unless you are a true encoding expert
 - In general, it's much more secure to simply use the `getValidXY()` methods of **ESAPI**, **which will handle encodings correctly** (most likely)
- Always try to identify all possible ways where data can be inserted into your application and **perform input validation everywhere**
 - For instance, perform input validation also in these cases:
 - A small application used by product managers to manipulate products
 - A tool that imports products from an Excel sheet into the DB
 - This will significantly reduce the number of vulnerabilities associated with a lack of input validation

Cross-Site Request Forgery Protection

CSRF Protection (1)

- Remember Cross-Site Request Forgery (CSRF)?
 - In a CSRF attack, an attacker attempts to force another user to **execute unwanted actions** in a web application in which that user is currently authenticated
- CSRF attacks **exploits “normal” web application features**
 - An authenticated user means the browser has a cookie that is associated by the web application with the authenticated session
 - Whenever the browser sends a request to the target web application, the cookie is sent as part of the request
 - It doesn't matter from where the link to trigger the requests stems: from the actual web application or from an attacker
- This implies that unless you explicitly implement measures to protect from CSRF, a web application **is likely vulnerable** to CSRF attacks

CSRF Protection (2)

- As we haven't done anything so far, the **Marketplace application is indeed vulnerable**, for instance:
 - A user with the role sales is currently logged in
 - He receives a message that contains a link to complete a purchase
 - If the user clicks the link, the purchase will be deleted

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

[Click me](#) if you can!

`https://ubuntu.dev:8443/marketplace/
admin/completepurchase?purchaseId=1`

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

CSRF Protection (3)

- The typical solution to prevent CSRF attacks works as follows:
 - For an authenticated user, create a random, **non-predictable value**, which is usually identified as **CSRF token**
 - Adapt the web pages sent to the user such **that any subsequent request includes the CSRF token**, in a GET or POST parameter
 - Either directly add the token to links (<a> tags)
 - Or use hidden fields with forms
 - When receiving a request, check whether the **received token matches** the one associated with the authenticated user
 - The request is only accepted if the tokens match
- Why does this work?
 - An attacker **would have to guess the CSRF token** to create a valid request, which is considered not feasible if the token is random and long enough

CSRF Protection (4)

- Today, many **mature web frameworks** include CSRF protection mechanisms that follow the CSRF token approach
 - E.g. Spring or JSF (see appendix)
- When using servlets/JSPs, Java EE does **not offer a specific feature** to prevent CSRF attacks
- This could be solved by using one of the specific security libraries
 - **OWASP CSRFGuard** (“stuck” in version 3 alpha, future unclear)
 - **OWASP Enterprise Security API (ESAPI)**
 - Requires that one also uses the login/authentication mechanisms of ESAPI
- Another option is implementing an **own CSRF protection** mechanism – which in fact is quite simple
 - Which is what we are going to do here – and which helps you to truly understand what exactly must be done to protect from CSRF attacks

Marketplace – CSRF Protection (1) – CSRF.java

- We extend the [Marketplace application](#) such that it provides CSRF protection
- We first write an [utility class CSRF](#) that provides the functionality to create and validate tokens
 - For the tokens, we use random 128-bit long values, encoded as hex string

```
public class CSRF {  
    private static SecureRandom prng = new SecureRandom();  
  
    public static String createToken() {  
        byte[] token = new byte[16];  
        prng.nextBytes(token);  
        return Crypto.printHex(token);  
    }  
  
    public static boolean validateToken(String token1, String token2) {  
        return token1.equals(token2);  
    }  
}
```

Create a random, 128-bit long token, encoded as hex string

Compare two tokens

Marketplace – CSRF Protection (2) – LoginServlet.java

- After successful login, **create a CSRF token** and store it in the session
 - We do this by extending LoginServlet.java

```
...  
  
// Read the salt from the DB and compute the digest  
String salt = LoginDB.getSalt(username);  
String digest = Crypto.printHex(Crypto.computeSHA1(password + salt));  
request.login(username, digest);  
  
// Create the CSRF token and store it in the session  
String csrfToken = CSRF.createToken();  
HttpSession session = request.getSession();  
session.setAttribute("csrftoken", csrfToken);  
  
url = "/admin/listpurchases";
```

Marketplace – CSRF Protection (3) – purchases.jsp

- We want to **protect the completion of a purchase**
 - Completing the purchase is done by clicking a link in purchases.jsp
 - The **link is extended with the CSRF token** by adding a GET parameter with the name csrftoken

```
<table cellpadding="5" border=1>
...

<c:forEach var="item" items="${purchases}">
  <tr valign="top">
    <td><c:out value="${item.firstName}" /></td>
    ...
    <td><c:out value="${item.totalPriceCurrencyFormat}" /></td>
    <c:if test="${completeAllowed}">
      <td><c:url var="url"
        value='/admin/completepurchase?purchaseId=${item.id}
          &csrftoken=${csrftoken}' />
      <a href="<c:out value="${url}" />">Complete Purchase</a></td>
    </c:if>
  </tr>
</c:forEach>
</table>
```

Accessing a session attribute is done in the same way as accessing a request attribute: `${csrftoken}` gets the token stored in the session attribute with name `csrftoken`

Marketplace – CSRF Protection (4) – DeletePurchaseServlet.java

- The request to complete a purchase is received by DeletePurchaseServlet, which must **check if the correct CSRF token is included**
 - In case of a mismatch, the action is not completed and the expected and received tokens are displayed for illustrative reasons
 - In a real application, it's probably best to terminate the session

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

    // Check the CSRF token
    String tokenReceived = request.getParameter("csrftoken");
    HttpSession session = request.getSession();
    String tokenExpected = (String) session.getAttribute("csrftoken");
    if (tokenReceived == null ||
        !CSRF.validateToken(tokenReceived, tokenExpected)) {
        message = "Token mismatch, expected: " + tokenExpected +
            ", received: " + tokenReceived;
    } else {
        // Token is valid, check the rights of the user
        ...
        // Get the purchaseId to delete and delete it
        ...
    }
}
```

Marketplace – CSRF Protection (5) – LogoutServlet.java

- For completeness, we **remove the CSRF token from the session** when the user is logged out

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

    // Log out
    request.logout();
    HttpSession session = request.getSession();
    session.removeAttribute("csrftoken");
    request.setAttribute("message", "You have been logged off.");

    // Forward to JSP
    String url = "/index.jsp";
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(url);
    dispatcher.forward(request, response);
}
```


Marketplace – CSRF Protection (6)

- These measures **effectively protect** from CSRF attacks
 - Submitting no token or a wrong token is detected
 - Only submitting the correct token will complete the purchase

Token mismatch, expected: 03db4bd3cfb7baaeeb104b29c4def67e, received: null

Click me if you can! →

<https://ubuntu.dev:8443/marketplace/admin/completepurchase?purchaseId=1>

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

Token mismatch, expected: 03db4bd3cfb7baaeeb104b29c4def67e, received: 4cd589ef100430cf554ed2c1c49b7ee1

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

ubuntu.dev:8443/marketplace/admin/completepurchase?purchaseId=1&csrftoken=03db4bd3cfb7baaeeb104b29c4def67e

Purchase completed

First Name	Last Name	CC Number	Total Price	
C64	Freak	1234 5678 9012 3456	\$444.95	Complete Purchase
Sprint	Love	5555 6666 7777 8888	\$10.05	Complete Purchase

Marketplace – CSRF Protection (7) – Further Remarks

- So far, we have **only protected one action** – completing a purchase
 - Full CSRF prevention requires protecting also all other actions (at least the sensitive ones) that can be performed by authenticated users
- If the action is triggered by clicking a link (<a> tag), protecting is done by **adding a csrftoken parameter to the link**
- If the action is triggered by submitting a **form**, a **hidden field** is used
 - No matter whether the form is submitted as GET or POST request
- Example: The following adds a hidden field to a form in a JSP
 - With **name csrftoken** and **value of the token stored in the session**
 - When submitting the form, a parameter `csrftoken=a72...` is included

```
...  
<input type="hidden" name="csrftoken" value=${csrftoken}>  
...
```

Summary

- From a security standpoint, very **many things can go wrong** when developing a Java web application
- **Java EE, Java SE, and 3rd party libraries** offer the necessary components to secure web applications
- Securing Java web applications requires a combination of **declarative and programmatic security** measures
 - If possible, use declarative security whenever possible and supplement it with programmatic security when needed
- Compared to Java EE 5, Java EE 6 introduced additional methods for **programmatic security and access control annotations**
- Servlets/JSP don't offer specific features to perform **input validation**, OWASP ESAPI is a good 3rd party library to fill this gap
- Servlets/JSPs also don't offer **CSRF protection** mechanisms, but this can easily be implemented "manually" or by using a 3rd party library
- When developing web applications, **consider about defense-in-depth**: E.g. perform input validation *and* use prepared statements to protect from SQL injection