UNIVERSITY OF TRENTO - Italy

eit Digital MASTER SCHOOL

Offensive technologies Fall 2016

Lecture 1- General Introduction to Vulnerabilities in Web Applications

Stanislav Dashevskyi

https://securitylab.disi.unitn.it/doku.php?id=course_on_offensive_technologies

About this lecture

- *The whole course is dedicated to the identification, testing and mitigation of various forms of security vulnerabilities*
- *The purpose of this lecture is to briefly introduce the background needed for recognizing some of the vulnerabilities in the source code*
- *We will test this ability using a practical exercise on Wednesday: it is important for the latter part of the course*

Outline

- *Vulnerabilities in web applications*
- *Injection vulnerabilities*
- *Information Disclosure vulnerabilities*
- *Session Fixation vulnerabilities*
- *Denial of Service vulnerabilities*

Vulnerabilities in web applications

- ***Many security holes in corporate IT are not due to worms or viruses, but due to vulnerabilities in the source code of applications***
  - These vulnerabilities are often exploited by attackers for both fun and profit
- ***Differences between web and client-server applications open enterprises to significant risk***
  - JavaScript has diffused boundaries between client and server
  - Easier to deploy, harder to maintain securely
- ***Web application security is critical for businesses***
- ***Finding and fixing web application vulnerabilities is mostly about looking at the source code***

Practical Approaches in Vulnerability Discovery

- ***Software security is a problem that is very hard to define***
- ***"A system is secure if and only if it starts in a secure state and cannot enter an insecure state" – the Bell-LaPadula model***
  - Even if we could define it, it's impossible to formalize:
    - "I do not want my email to be read by others"
  - There is no way to define a desired behavior for a considerably complex system
    - Different stakeholders act according to the "tragedy of commons" dilemma
  - It is nearly impossible to analyze software behavior conclusively
    - A. Turing's halting problem
    - H.G. Rice's theorem
- ***For now, security is largely a non-algorithmic problem***
  - Eventually, security field specialists fall back to set of empirical recipes

Practical Approaches in Vulnerability Discovery (continued)

- ***Plan to have everything compromised***
  - Everything is vulnerable
- ***Rely on tools to detect and correct <u>SPECIFIC</u> problems but not replace everything by tools***
  - Tools can help finding certain vulnerabilities but they are nothing without knowledge
- ***Learn from (preferably) other's mistakes***
  - We can use Open Source Software to learn

Why looking at open source software?

- *There is little difference with commercial software*
- *The source code and development histories are available*
- *Often, open source maintainers are doing a good job in documenting vulnerabilities, so it is possible to reverse-engineer them*
- *Many commercial systems are using open source components, thus the learning effort will be useful*

A quick look at vulnerabilities taxonomy

- ***There are different categories, classifications and databases***
  - Open Web Application Security Project (OWASP) Top 10 list
  - Common Weakness Enumeration (CWE)
  - Common Weakness Scoring System (CWSS)
  - The National Vulnerability Database (NVD)
  - ~~Open-sourced Vulnerability Database (OSVDB)~~
  - IARPA Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP)

- ***Almost all these vulnerabilities are related to problems in the source code***
  - Design errors
  - Implementation errors
  - Many of them are Language/Framework independent

OWASP Top 10 (2013)

- A1: Injection
- A2: Broken Auth. and Session Management
- A3: Cross-site Scripting (XSS)
- A4: Insecure Direct Object References
- A5: Security Misconfiguration
- A6: Sensitive Data Exposure
- A7: Missing Function Level Access Control
- A8:Cross-site Request Forgery (CSRF)
- A9: Using Component With Known Vulns.
- A10: Unvalidated Redirects and Forwards

Common Weakness Enumeration (CWE)

- *https://cwe.mitre.org/*
- ***A formal dictionary of common software bugs/flaws that occur in software architecture, design, and implementation that can lead to exploitable security vulnerabilities (> 800 entries)***
- ***A common language for describing and a standard for measuring such bugs/flaws***
- ***Information about identification/mitigation/prevention efforts***

Common Weakness Enumeration (CWE)

Nature	Type	ID	Name	V	
ChildOf		20	Improper Input Validation	**700**	
ChildOf		74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	**699** **1000** **1003**	
ChildOf		442	Web Problems	699	
ChildOf		712	OWASP Top Ten 2007 Category A1 - Cross Site Scripting (XSS)	**629**	
ChildOf		722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	
ChildOf		725	OWASP Top Ten 2004 Category A4 - Cross-Site Scripting (XSS) Flaws	**711**	
ChildOf		751	2009 Top 25 - Insecure Interaction Between Components	**750**	
ChildOf		801	2010 Top 25 - Insecure Interaction Between Components	**800**	
ChildOf		811	OWASP Top Ten 2010 Category A2 - Cross-Site Scripting (XSS)	**809**	
ChildOf		864	2011 Top 25 - Insecure Interaction Between Components	**900**	
ChildOf		931	OWASP Top Ten 2013 Category A3 - Cross-Site Scripting (XSS)	**928**	
ChildOf		990	SFP Secondary Cluster: Tainted Input to Command	**888**	
CanPrecede		494	Download of Code Without Integrity Check	1000	
PeerOf		352	Cross-Site Request Forgery (CSRF)	1000	
ParentOf		80	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	**699** **1000**	
ParentOf		81	Improper Neutralization of Script in an Error Message Web Page	**699** **1000**	
ParentOf		83	Improper Neutralization of Script in Attributes in a Web Page	**699** **1000**	
ParentOf		84	Improper Neutralization of Encoded URI Schemes in a Web Page	**699** **1000**	
ParentOf		85	Doubled Character XSS Manipulations	**699** **1000**	
ParentOf		86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages	**699** **1000**	
ParentOf		87	Improper Neutralization of Alternate XSS Syntax	**699** **1000**	
MemberOf		635	Weaknesses Used by NVD	**635**	
MemberOf		884	CWE Cross-section	**884**	
CanFollow		113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	1000	
CanFollow		184	Incomplete Blacklist	1000	692

UNIVERSITY OF TRENTO - Italy

Common Weakness Enumeration (CWE)

Observed Examples

Reference	Description
CVE-2008-5080	Chain: protection mechanism failure allows XSS
CVE-2006-4308	Chain: only checks "javascript:" tag
CVE-2007-5727	Chain: only removes SCRIPT tags, enabling XSS
CVE-2008-5770	Reflected XSS using the PATH_INFO in a URL
CVE-2008-4730	Reflected XSS not properly handled when generating an error message
CVE-2008-5734	Reflected XSS sent through email message.
CVE-2008-0971	Stored XSS in a security product.
CVE-2008-5249	Stored XSS using a wiki page.
CVE-2006-3568	Stored XSS in a guestbook application.
CVE-2006-3211	Stored XSS in a guestbook application using a javascript: URI in a bbcode img tag.
CVE-2006-3295	Chain: library file is not protected against a direct request (CWE-425), leading to reflected XSS.

The National Vulnerability Database (NVD)

- ***https://nvd.nist.gov/***
- ***The US Government repository of vulnerability data***
- ***Enables automation of vulnerability management, security measurement and compliance***
- ***Includes databases of security-related software flaws/bugs, product names, and impact metrics***
- ***Supports the Common Vulnerability Scoring System (CVSS) scores***
  - Quantifies characteristics of each vulnerability so that they can be compared

The National Vulnerability Database (NVD)

National Cyber Awareness System

Vulnerability Summary for CVE-2014-0075

Original release date: 05/31/2014

Last revised: 08/22/2016

Source: US-CERT/NIST

Overview

Integer overflow in the parseChunkHeader function in java/org/apache/coyote/http11/filters/ChunkedInputFilter.java in Apache Tomcat before 6.0.40, 7.x before 7.0.53, and 8.x before 8.0.4 allows remote attackers to cause a denial of service (resource consumption) via a malformed chunk size in chunked transfer coding of a request during the streaming of data.

Impact

CVSS Severity (version 2.0):

CVSS v2 Base Score: 5.0 MEDIUM

Vector: (AV:N/AC:L/Au:N/C:N/I:N/A:P) (legend)

Impact Subscore: 2.9

Exploitability Subscore: 10.0

CVSS Version 2 Metrics:

Access Vector: Network exploitable

Access Complexity: Low

Authentication: Not required to exploit

Impact Type: Allows disruption of service

Outline

- ***Vulnerabilities in web applications***
- ***Injection vulnerabilities***
- ***Information Disclosure vulnerabilities***
- ***Session Fixation vulnerabilities***
- ***Denial of Service vulnerabilities***

Injection vulnerabilities

- ***Assume an application is written in multiple languages: Java, JavaScript, HTML, SQL …***
- ***An application accepts any user input without sanitization***
  - Problem: some input that looks like a String in Java can be accepted as a piece of executable code by SQL, JavaScript, or HTML interpreters
  - These are also called ”polyglot” vulnerabilities
- ***Consequences?***
  - Website defacement
  - …
  - Complete control over the machine that hosts the vulnerable application

SQL/NoSQL injection

- ***Description:***
  - Due to insufficient input filtering (or output escaping) attacker-controlled input may be interpreted as code by a database interpreter and executed **[1].** Eventual outcome is code execution.
- ***Related Threats: Information Disclosure, Data Modification/Deletion, Elevation of Privileges.***
- ***Technical Impact: Severe.***

SQL injection: example

```
UserData data = getDataFromUser();
String userId = data.getUserId();
String passwd = data.getPasswd();
SomeDB.executeQuery("SELECT * FROM users WHERE users.userId = ’
              + userId + ”’ AND users.passwd ='” + passwd + “'");
```

```
userid <- ”John Doe”
passwd <- ”qweJk@#4kw”
query <- "SELECT * FROM users WHERE users.userId =
’John Doe’ AND user.passwd = ’qweJk@#4kw’”
```

```
userId <- “Batman’ OR ’1’ == ’1’; DROP TABLE users; --”
passwd <- “”

query <- "SELECT * FROM users WHERE users.userId =
’Batman’ OR ’1’ == ’1’; DROP TABLE users; --’ AND users.passwd= ’’"
```

NoSQL injection: example

```
exports.insecure = function(request, response) {
    var login = request.body.userid;
    var password = request.body.passwd;
    var loginParam = eval("({ _id: '" + login + "', pword : '" + password + "'})");

    server.dbprovider.findOne("users", loginParam, function(error, item) {
        if (error != null) {
            response.send("MongoDB ERROR: " + error);
            return;
        }
        if (item != null) {
            response.send("Hello, " + item._id + "!");
        }
        else {
            response.send("A
        }
    });
}
```

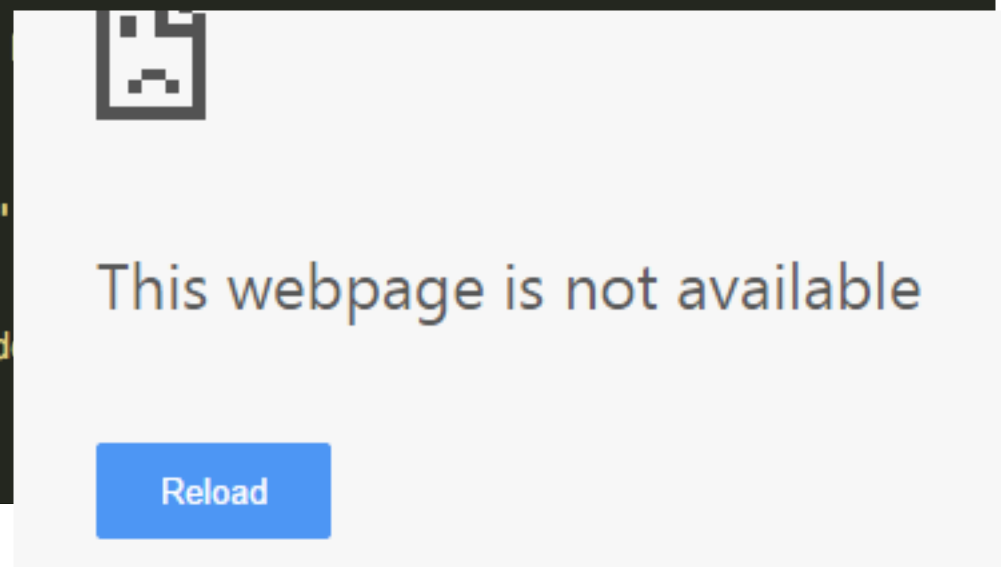
```
Batman'})//
```

*The image is taken from http://www.busanhlf4.org/

NoSQL injection: example

```
exports.insecure = function(request, response) {
    var login = request.body.userid;
    var password = request.body.passwd;
    var loginParam = eval("({ _id: '" + login + "', pword : '" + password + "'})");

    server.dbprovider.findOne("users", loginParam, function(error, item) {
        if (error != null) {
            response.send("MongoDB
            return;
        }
        if (item != null) {
            response.send("Hello, "
        }
        else {
            response.send("Access d
        }
    });
}
```




```
Batman'}); process.exit(); //
```

*The image is taken from http://www.busanhlf4.org/

SQL/NoSQL injection: how to find it?

- ***You should be suspicious if an application***
  - Gets user input
  - Does not check/sanitize the input
  - Uses this input to construct a query to a database
  - Uses string operations (e.g., concatenation, replacement) to build a query

Language	Keywords
Java (+JDBC)	`sql, java.sql`
Python	`pymssql,`
C#	`Sql, SqlClient, OracleClient, SqlDataAdapter`
PHP	`mysql_connect`
Node.js	`require("mysql”), require(”mssql"), require("mongodb")`

Cross-Site Scripting (XSS)

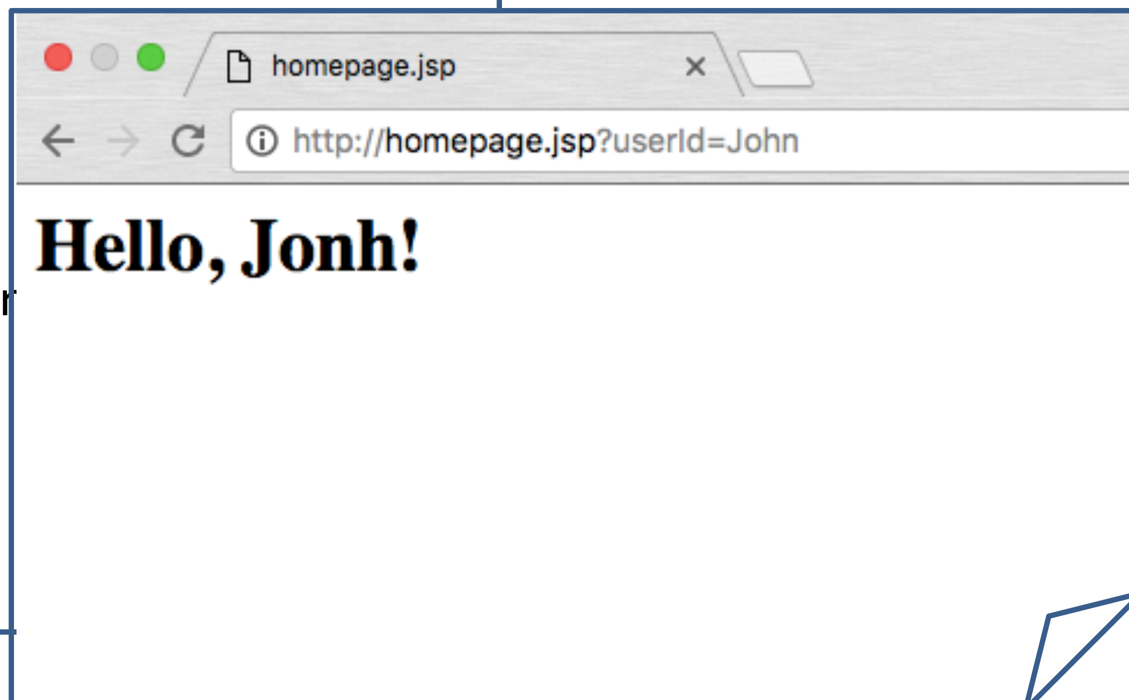
- ***Description:***
  - "Insufficient input validation or output escaping can allow an attacker to plant his own HTML or scripts on a vulnerable site. The injected scripts will have access to the entirety of the targeted web application … " **[2]**.
  - The *reflected* variant takes the advantage when the input is incorrectly echoed back to the browser; the *persistent* variant goes a bit further: it also takes the advantage on the lack of sanitization of the data that goes to a DB.
- ***Related Threats:***
  - Information Disclosure, Elevation of Privileges.
- ***Technical Impact:***
  - Moderate/Severe

Cross-Site Scripting (XSS): reflected

http://homepage.jsp?userId=John

```
…
<% String userId =
request.GetParameter(”userId") %>
…

<html>
    ...
    <h1>
        Hello, <%= user
    </h1>
    ...
</html>
```

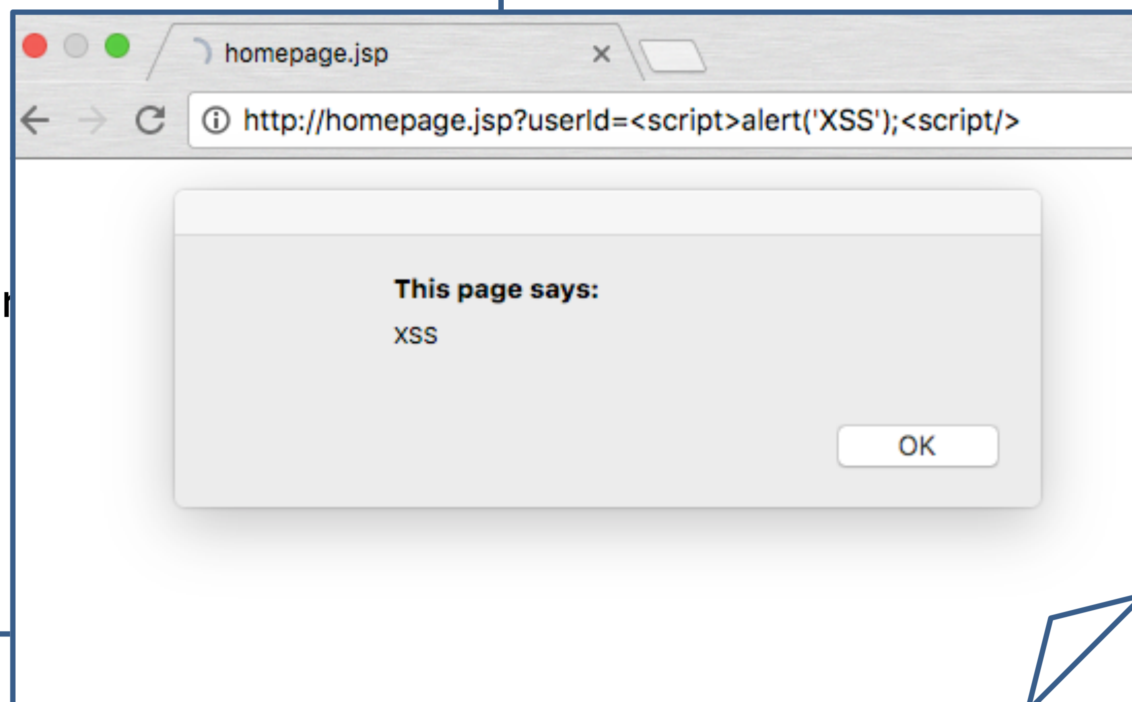


Cross-Site Scripting (XSS): reflected

http://homepage.jsp?userId=<script>alert('XSS');</script>

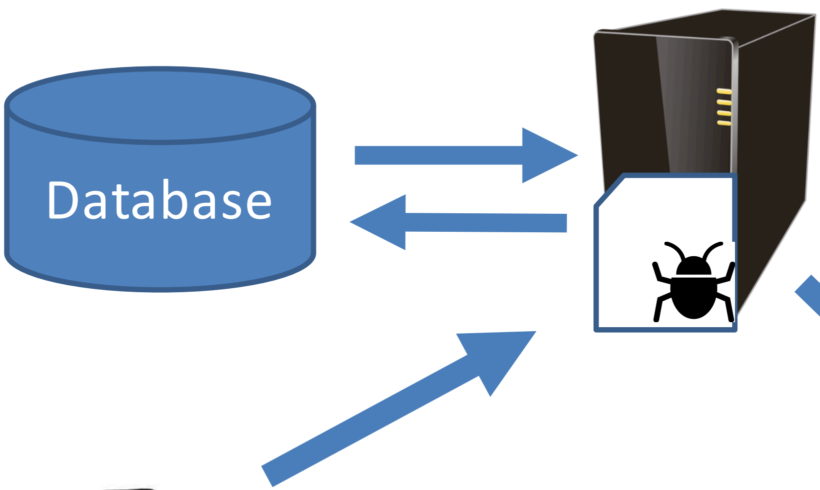
```
…
<% String userId =
request.GetParameter(”userId") %>
…

<html>
    ...
    <h1>
        Hello, <%= user
    </h1>
    ...
</html>
```



Cross-Site Scripting (XSS): stored

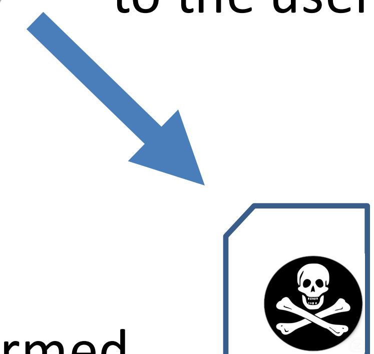
Step 0 -> developer writes vulnerable pages:
1st one stores invalidated input;
2nd one reads it from a database and with no validation.



Step 2 -> User browses the site.

Step 3 -> Web site reads unchecked data and sends it along with attacker's code to the user's browser.

Step 1 ->
Attacker sends malformed input (code) to a vulnerable web page.



Step 4 -> User's browser renders the web page and runs the attacker's code (every time the page is requested!)

*The diagram is adapted from **[3]**.

Cross-Site Scripting (XSS): some examples (reflected)

http://homepage.jsp?page=123

```
public class XSS extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {

        /* ... */
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
                           "The page \"" +
                           request.getParameter("page") +
                           "\" was not found.");
    }
}
```

Cross-Site Scripting (XSS): some examples (reflected)

http://homepage.jsp?page=<script>alert('XSS')</script>

```
public class XSS extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {

        /* ... */
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
                           "The page \"" +
                           request.getParameter("page") +
                           "\" was not found.");
    }
}
```

UNIVERSITY OF TRENTO - Italy

EIT Digital MASTER SCHOOL

Cross-Site Scripting (XSS): some examples (stored)

http://show-employee.jsp?eid=123

```
<%
...
String eid = request.GetParameter(”eid”);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(”select *
                from emp where id='” + eid + ”'”);
if (rs != null) {
   rs.next();
}
String bio = rs.getString(”bio”);

Employee biography: <%= bio %>
…
%>
```

UNIVERSITY OF TRENTO - Italy

Cross-Site Scripting (XSS): some examples (stored)

http://show-employee.jsp?eid=qwe‘ or ’1’ == ’1’; insert into emp (bio) values ('<script>alert(\"XSS\")</script>’) select * from emp; --

```
<%
...
String eid = request.GetParameter(”eid”);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(”select *
             from emp where id='” + eid + ”'”);
if (rs != null) {
   rs.next();
}
String bio = rs.getString(”bio”);

Employee biography: <%= bio %>
…
%>
```

Cross-Site Scripting (XSS): how to find it?

- ***You should be suspicious if an application***
  - Gets an input from an HTTP entity such as query string, header or form, or request object
  - Does not check the input for validity
  - Echoes it back to the browser (either HTML or HTTP headers), saving it to or retrieving from a database unchecked

Cross-Site Scripting (XSS): how to find it?

Language	Keywords
Java (JSP)	`addCookie`, `getRequest`, `request.getParameter` followed by `<jsp:setProperty` or `<%=` or `response.sendRedirect`
Python	`form.getvalue`, `SimpleCookie` when the data is not validated correctly.
C#	`Request.*`, `Response.*`, and <%= when the data is not validated correctly.
PHP	Accessing $_REQUEST, $_GET, $_POST, or $_SERVER followed by `echo`, `print`, `header`, or `printf`.
Node.js	`request`, `response`, …

Outline

- ***Vulnerabilities in web applications***
- ***Injection vulnerabilities***
- ***Information Disclosure vulnerabilities***
- ***Session Fixation vulnerabilities***
- ***Denial of Service vulnerabilities***

Information Disclosure vulnerabilities

- ***Description:***
  - Attacker is able to get data that leads to a breach in security or privacy policy. The data itself could be the goal, or the data can provide information that leads the attacker to the goal.
  - **Intentional**: the design team has a mismatch with the end user as to whether data should be protected (privacy issues).
  - **Accidental**: the data could leak due to an error in the code, or a nonobvious channel.
  - **Mistake**: verbose [error] messages that developers think are safe, but attackers find them helpful, e.g., the name or the ip address of a server
  - **Three main categories**: hardcoded credentials, comments in the source code, and verbose error messages.
- ***Technical impact: could be anything***

Information Disclosure: example 0

```
try {
    /* ... */
}
catch (Exception e) {
    System.out.println(e);
    e.printStackTrace();
}
```

Information Disclosure: example 1

```php
<?php
  $UName = "[illegible]";
  $PWord = "[illegible]";
  $DB="[illegible]";
?>
```

```python
def authenticate(uname, pword):
    if uname == "":
        return False
    elif pword != "[illegible]":
        return False
    else:
        return True
```

```
user name: pb-admin
pword:     [illegible]
```

```python
def authenticate(uname,pword):
    if uname=="[illegible]" and pword=="[illegible]":
        return True
    else:
        return False
```

Information Disclosure: example 2

```
public boolean authenticate(Request req, Response res) {
    /* ... */
    if (config.getRealmName() == null) {
       authenticateCC.append(request.getServerName());
       authenticateCC.append(':');
       authenticateCC.append(Integer.toString(
              request.getServerPort()));
    }
    else {
       authenticateCC.append(config.getRealmName());
    }
    return (false);
}
```

Information Disclosure: example 2

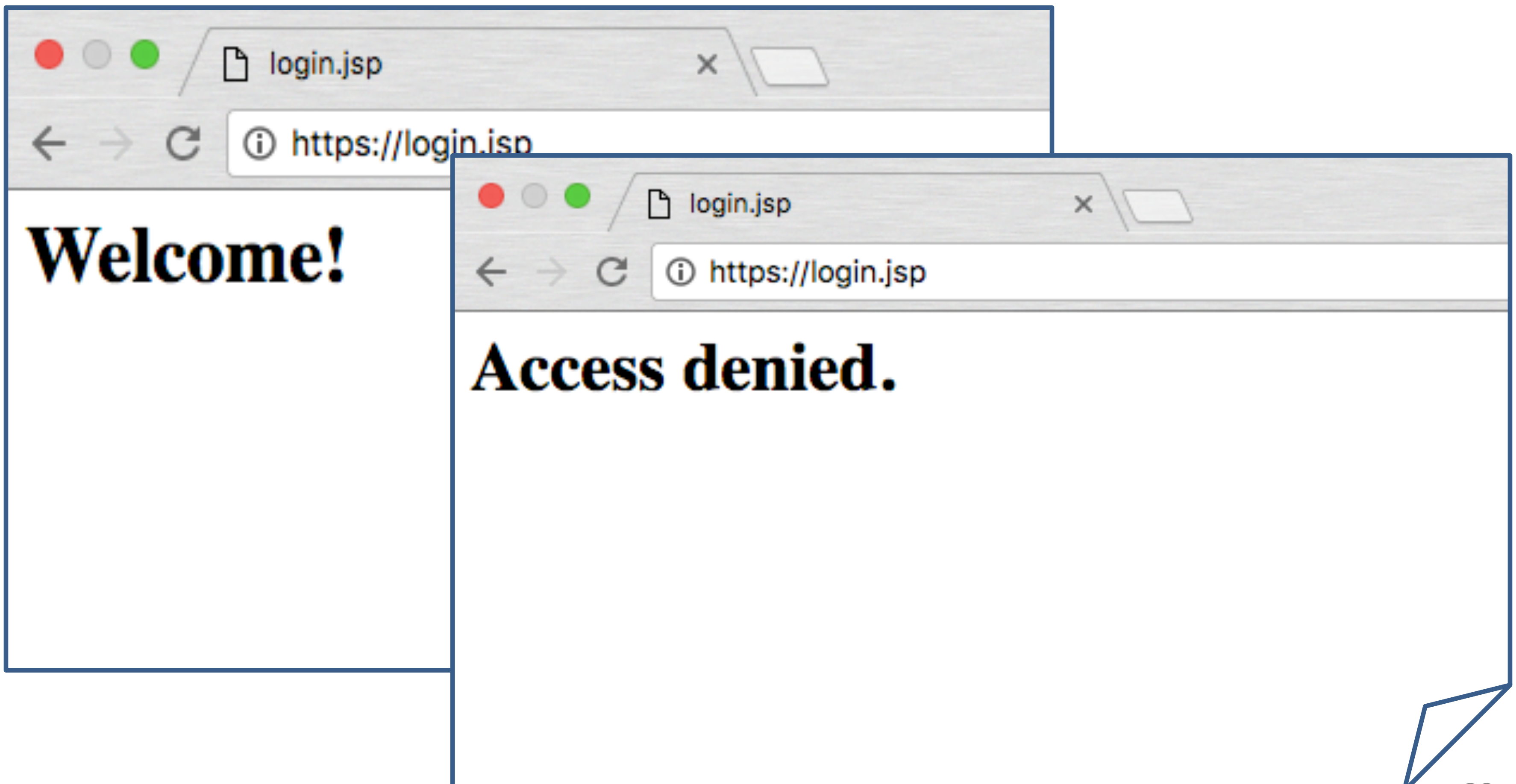
```
public boolean authenticate(Request req, Response res) {
    /* ... */
    if (config.getRealmName() == null) {
       authenticateCC.append(request.getServerName());
       authenticateCC.append(':');
       authenticateCC.append(Integer.toString(
              request.getServerPort()));
    }
    else {
       authenticateCC.append(config.getRealmName());
    }
    return (false);
}
```

Information Disclosure: example 2

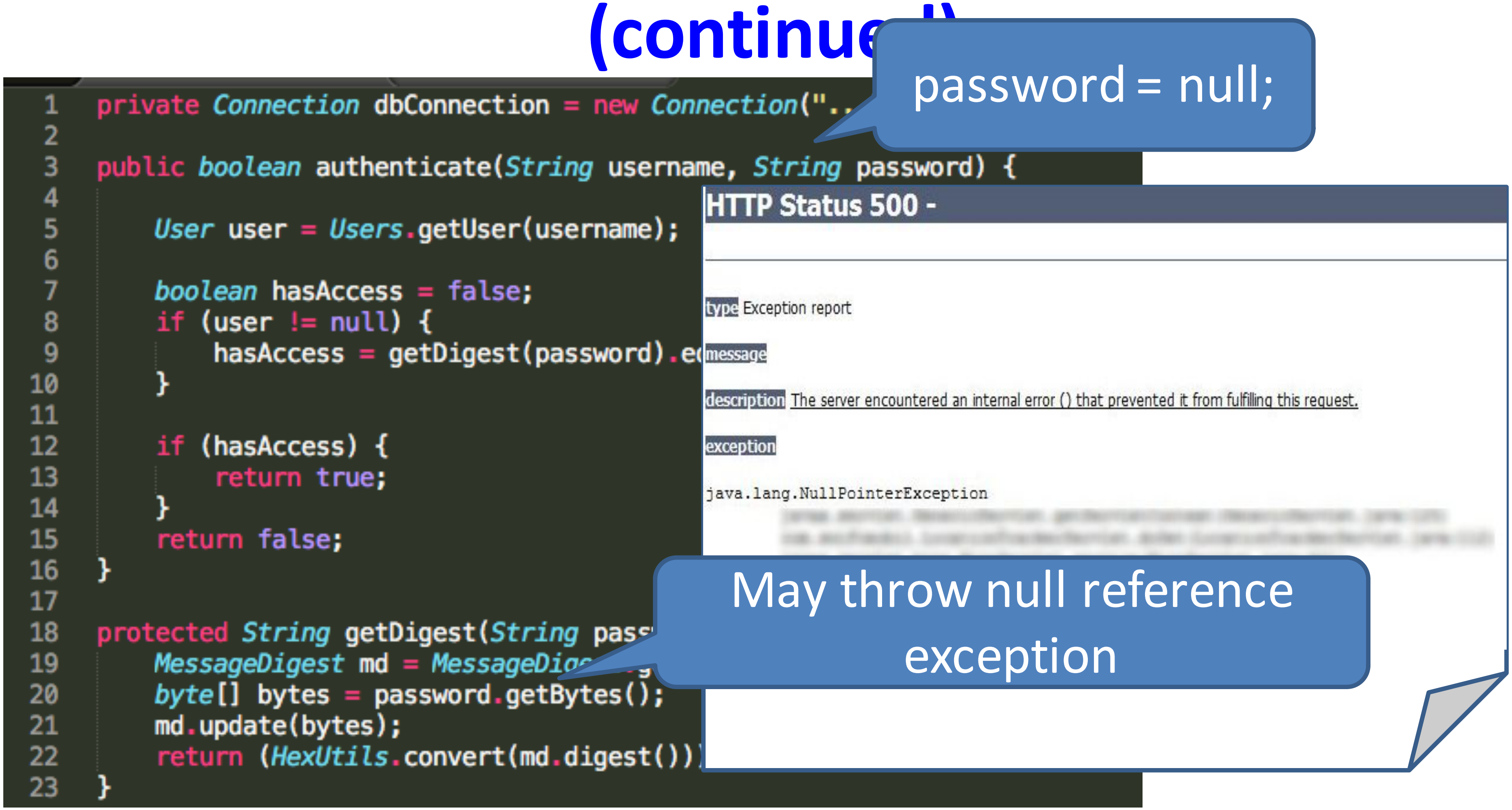
```
public boolean authenticate(Request req, Response res) {
    /* ... */
    if (config.getRealmName() == null) {
       authenticateCC.append(request.getServerName());
       authenticateCC.append(':');
       authenticateCC.append(Integer.toString(
             request.getServerPort()));
    }
    else {
       authenticateCC.append(config.getRealmName());
    }
    return (false);
}
```

Information Disclosure: example 3

Login successful: "authenticate" method returns ”true”



Information Disclosure: example 3 (continued)



Information Disclosure: how to find it?

- ***Application returns "default " information such as server type/ configuration/ip address/hostname.***
- ***Too many details in error messages, unhandled exceptions, stack traces; different error messages when handling user login.***
- ***Look for "password", "credentials", "login" and similar keywords, you might find something quite interesting.***

Path Traversal

- ***Description:***
  - An application can be tricked into reading or writing files at arbitrary locations (often bypassing application-level restrictions). This often happens due to improper recognition of "../" segments in un user-supplied parameters. Unconstrained file writing bugs are often exploited for deploying attacker-controlled code **[2]**.
- ***Related threats: Information disclosure, code injection, denial of service***
- ***Technical impact: Moderate/Severe***

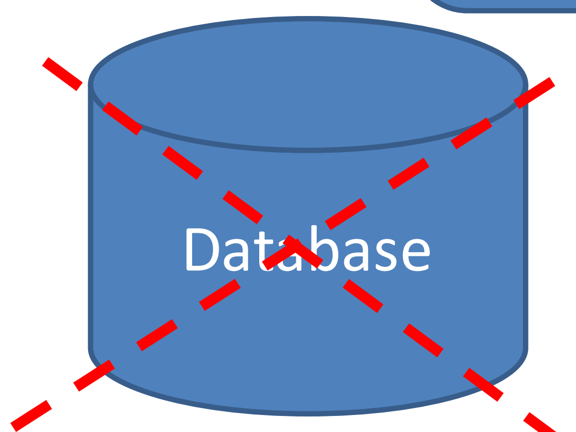
Path Traversal: some examples

```
String path = getInputPath();
if (path.startsWith("/safe_dir/")) {
   File f = new File(path);
   f.delete();
}
```

An attacker could provide an input such as : /safe_dir/../data.db

The code attempts to validate the input by whitelisting.

If the file is within the "/safe_dir/" folder, the file gets deleted.



Path Traversal: some examples (continued)

```
public void sendUserFile(Socket sock, String user) {
    BufferedReader filenameReader = new BufferedReader(
       new InputStreamReader(sock.getInputStream(), "UTF-8"));

    String filename = filenameReader.readLine();
    BufferedReader fileReader =
        new BufferedReader(new FileReader("/home/" + user +
            "/" + filename));

    String fileLine = fileReader.readLine();
    while(fileLine != null) {
      sock.getOutputStream().write(fileLine.getBytes());
      fileLine = fileReader.readLine();
    }
}
```

Path Traversal: some examples (continued)

```
public void sendUserFile(Socket sock, String user) {
    BufferedReader filenameReader = new BufferedReader(
       new InputStreamReader(sock.getInputStream(), "UTF-8"));

    String filename = filenameReader.readLine();
    BufferedReader fileReader =
        new BufferedReader(new FileReader("/home/" + user +
           "/" + filename));

    String fileLine = fileReader.readLine();
    while(fileLine != null) {
      sock.getOutputStream().write(fileLine.getBytes());
      fileLine = fileReader.readLine();
    }
}
```

Path Traversal: how to find it?

- ***You should be suspicious if an application***
  - Gets an input from user
  - The input is used to construct a path for any purpose (downloading/uploading files, redirects, etc.)
  - Even if the input looks like it is sanitized, sanitization functions often contain errors, so you pay close attention to sanitizers
  - Sometimes there are no path constraints at all

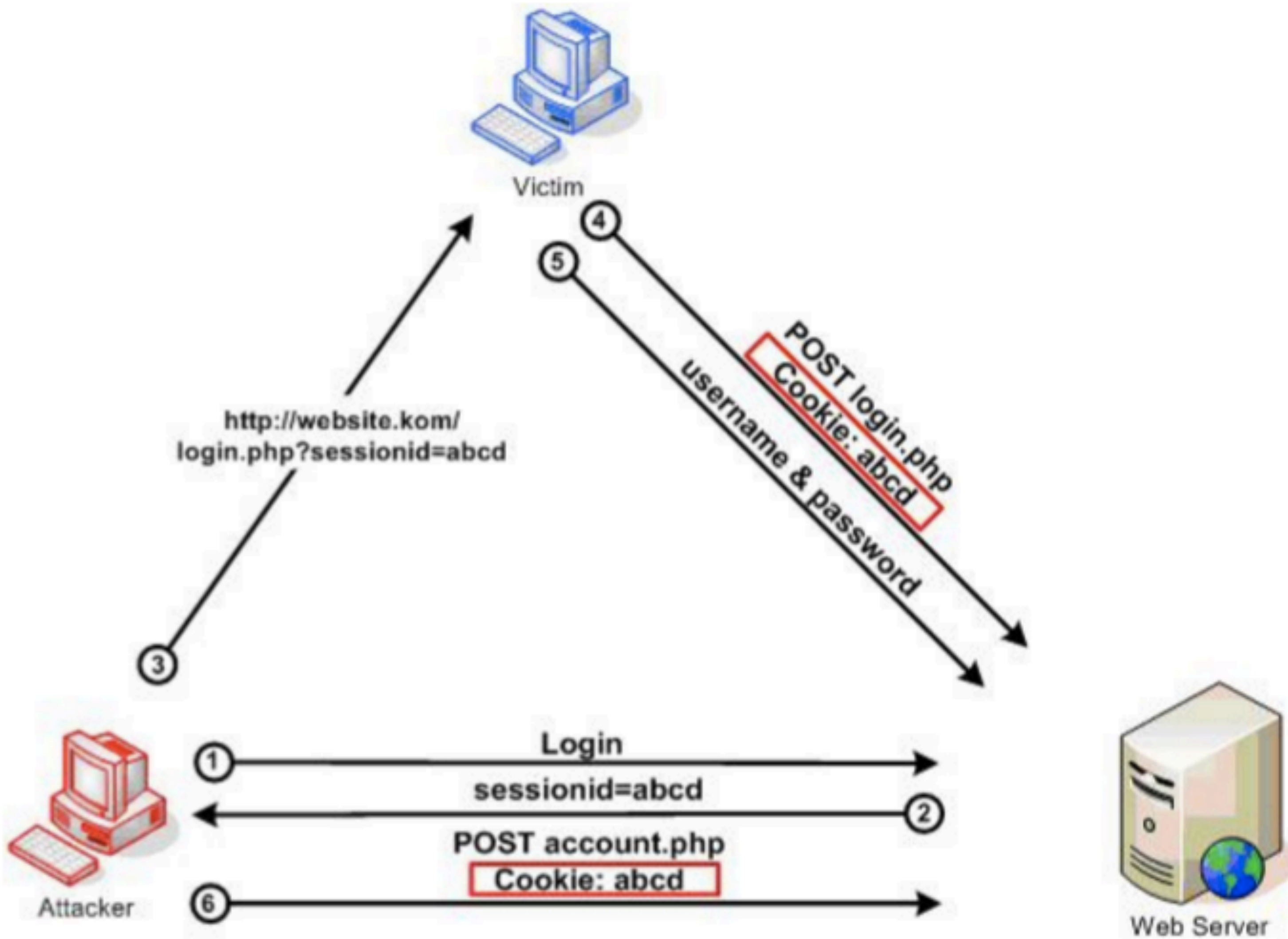
Outline

- ***Vulnerabilities in web applications***
- ***Injection vulnerabilities***
- ***Information Disclosure vulnerabilities***
- ***Session Fixation vulnerabilities***
- ***Denial of Service vulnerabilities***

Session Fixation vulnerabilities

- ***Description:***
  - An attack that allows to hijack a valid user session. When authenticating a user, an app doesn’t assign a new session ID, making it possible to use an existent session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim's browser use it. **[5]**
- ***Technical impact: Severe***

Session Fixation: example*



*This example is taken from **[4]**.

Session Fixation: example

1. *The attacker establishes a legitimate connection with a web server;*
2. *The web server issues a session ID;*
3. *The attacker has to send a link with the established session ID to the victim; she has to click on the link, accessing the site;*
4. *The web server "sees" that the session has been already established (by the attacker), so it doesn't create a new one;*
5. *The victim provides her credentials to the web server; the attacker can access her account knowing the session ID.*

(session ID can be also sent via a cookie or a hidden field in the DOM container)

Session Fixation: example (continued)

```
protected boolean parseRequest(Request req, Response res) {
    if (isURLRewritingDisabled(req)) {
        clearRequestedSessionURL(req);
    }

    /* ... */

    String sessionID =
       req.getPathParameter(Globals.SESSION_PARAMETER_NAME);

    if (sessionID != null) {
        req.setRequestedSessionId(sessionID);
        req.setRequestedSessionURL(true);
    }

    /* ... */
}
```

UNIVERSITY OF TRENTO - Italy

eit Digital MASTER SCHOOL

Session Fixation: example (continued)

```
protected boolean parseRequest(Request req, Response res) {
    if (isURLRewritingDisabled(req)) {
        clearRequestedSessionURL(req);
    }

    /* ... */

    String sessionID =
       req.getPathParameter(Globals.SESSION_PARAMETER_NAME);

    if (sessionID != null) {
        req.setRequestedSessionId(sessionID);
        req.setRequestedSessionURL(true);
    }

    /* ... */
}
```

Session Fixation: example (continued)

```
protected boolean parseRequest(Request req, Response res) {
    if (isURLRewritingDisabled(req)) {
        clearRequestedSessionURL(req);
    }

    /* ... */

    String sessionID =
       req.getPathParameter(Globals.SESSION_PARAMETER_NAME);

    if (sessionID != null) {
        req.setRequestedSessionId(sessionID);
        req.setRequestedSessionURL(true);
    }

    /* ... */
}
```

Session Fixation: example (continued)

```
protected boolean parseRequest(Request req, Response res) {
    if (isURLRewritingDisabled(req)) {
        clearRequestedSessionURL(req);
    }

    /* ... */

    String sessionID =
       req.getPathParameter(Globals.SESSION_PARAMETER_NAME);

    if (sessionID != null && !isURLRewritingDisabled(req)) {
        req.setRequestedSessionId(sessionID);
        req.setRequestedSessionURL(true);
    }

    /* ... */
}
```

Session Fixation: how to find it? [5]

- ***You should be suspicious if the usual flow is broken [6]***
  - User enters correct credentials
  - The application authenticates the user successfully
  - Session information (temporary data) is stored in a temporary location
  - Session is invalidated (*session.invalidate()*)
  - Any temporary data is restored to new session (new session ID)
  - User goes to successful login landing page using new session ID

Session Fixation: how to find it? (continued) [5]

- *Check for session fixation if a user tries to login using a session ID that has been specifically invalidated (requires maintaining this list in some type of URL cache)*
- *Check for session fixation if a user tries to use an existing session ID already in use from another IP address (requires maintaining this data in some type of map)*
- *Some server applications (e.g., JBOSS, Tomcat) have a setting for disabling URL rewriting -> this mitigates the attack when session ID is exposed via GET parameter of a URL (as well as being stored in browser history, proxy servers, etc)*

Outline

- *Vulnerabilities in web applications*
- *Injection vulnerabilities*
- *Information Disclosure vulnerabilities*
- *Session Fixation vulnerabilities*
- *Denial of Service vulnerabilities*

Denial of Service vulnerabilities

- ***Description:***
  - The Denial of Service (DoS) attack is focused on making a resource (site, application, server) unavailable for the purpose it was designed. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses.
- ***Technical impact: Severe***

UNIVERSITY OF TRENTO - Italy

EIT Digital MASTER SCHOOL

Denial of Service: example 1

```
String TotalObjects = request.getParameter("numberofobjects");
int NumOfObjects = Integer.parseInt(TotalObjects);
ComplexObject[] anArray = new ComplexObject[NumOfObjects];
```

We may "kill" the server by filling all of its memory

Denial of Service: example 2

```
public class MyServlet extends ActionServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                        throws ServletException, IOException {
        /*. . .*/
        String [] values = request.getParameterValues("CheckboxField");
        // Process the data without length check for reasonable range – wrong!
        for ( int i=0; i<values.length; i++) {
            // lots of logic to process the request
        }
        /*. . .*/
    }
    /*. . .*/
}
```

The user has control over the loop counter: we may decrease server's performance or even kill it.

Denial of Service: example 3

```java
public class AccountDAO {
    /* ... */
    public void createAccount(AccountInfo acct)
                    throws AcctCreationException {
    /* ... */
        try {
            Connection conn = DAOFactory.getConnection();
            CallableStatement  calStmt = conn.prepareCall(…);
            /* ... */
            calStmt.executeUpdate();
            calStmt.close();
            conn.close();
        }  catch (java.sql.SQLEx
             throw AcctCreati
        }
    }
}
```

Both Connection and CallableStatement objects should be closed in the “finally” block

Denial of Service: how to find it?

- ***You should be suspicious if***
  - User-controlled values define the size of allocated memory, arrays or buffers;
  - User-controlled values influence loop conditions;
  - ”Heavy” resources are never released (file locks/descriptors, database connections, data streams, etc.)
  - There is an "infinite" amount of resources that a single user can allocate (e.g., the number of working processes or server sockets);

References

- *[1] Web Application Vulnerabilities and Avoiding Application Exposure* *https://f5.com/resources/white-papers/web-application-vulnerabilities-and-avoiding-application-exposure*
- *[2] Zalewski, Michal. The tangled Web: A guide to securing modern web applications. No Starch Press, 2012.*
- *[3] Michael Howard, David LeBlanc, and John Viega. 24 deadly sins of software security: programming flaws and how to fix them. McGraw-Hill, Inc., 2009.*
- *[4] OWASP: the free and open software security community* *https://www.owasp.org/index.php/Main_Page*
- *[5] The White Hat Security blog on Session Fixation prevention:* *https://www.whitehatsec.com/blog/session-fixation-prevention-in-java/*
- *[6] The OWASP Enterprise Security API session handling example:* *https://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/reference/DefaultHTTPUtilities.java*
- *[7] Secure Coding Guidelines for Java SE* *http://www.oracle.com/technetwork/java/seccodeguide-139067.html*