Teaching Secure Programming to Information Systems Students via OWASP Techniques and Libraries

Carey Cole
James Madison University, Harrisonburg, VA, United States., colecb@jmu.edu

Michel Mitri
James Madison University, Harrisonburg, VA, United States., mitrimx@jmu.edu

Teaching Secure Programming to Information Systems Students via OWASP Techniques and Libraries

Carey Cole
James Madison University
colecb@jmu.edu

Michel Mitri
James Madison University
Mitrimx@jmu.edu

ABSTRACT

Current and future Information Systems (IS) personnel and management need to understand SQL Injection, cross-site scripting (XSS), and other web-originating information security vulnerabilities. These can have severe negative impacts, and minimizing these threats is an important consideration for application developers. There are many resources on the Internet and in books to help educate people about these and similar intrusions. The Open Web Application Security Project (OWASP) includes a robust amount of information on this subject and is an excellent starting point in the creation of lecture, demonstration, and student practice on the subject. Using OWASP resources and active software examples is an effective and efficient method to teach IS students on potential security breaches and their prevention.

Keywords

SQL Injection, cross-site scripting (XSS), enterprise security API, OWASP, IT security education.

INTRODUCTION

IT Security is a major concern in many organizations, especially those providing web-based access to their systems. Although security measures like SSL and firewalls provide important protection measures, these do not go far enough in addressing vulnerabilities that exist via inputs from users on browsers. The two most common such vulnerabilities are SQL Injection and Cross Site Scripting (XSS).

SQL Injection and cross site scripting attacks happen frequently and can be very damaging. From 2005 to 2007, Albert Gonzales and two Russians were involved in stealing about 130 million credit card numbers from several companies of which one was 7-11 (Bhushan, 2009). The first Cross Scripting attack on October 4, 2005, was called the "Samy Worm". The target was the personal profiles for 32 million users in MySpace. MySpace had to shut down to fix the attack (Grossman 2007).

These and other security attacks and vulnerabilities have made it increasingly important to improve educational coverage of IT security issues in core Information Systems curricula. This is true in the managerially oriented coursework, in which best practices and security policy formation would be discussed, and also in the technical coursework such as software development courses, in which students should learn about effective coding techniques for preventing SQL injection, XSS, and other security threats.

Traditionally, computer programming classes that include security components have concentrated on the development of security algorithms. But for an Information Systems curriculum, unlike a computer science curriculum, students typically do not gain sufficient coding skills to be able to implement these algorithms; nor should this be necessary from a practical perspective. Rather, a more appropriate venue for these students would involve training in how to utilize existing class libraries and methods for securing their software applications.

In this paper, we explore the use of the Open Web Application Security Project (OWASP) and it Enterprise Security API (ESAPI) for teaching students about the best practices for secure coding as well as the use of pre-existing security classes and methods.

THE OPEN WEB APPLICATION SECURITY PROJECT (OWASP)

Many different types of potential security issues exist for any business application. The Open Web Application Security Project (OWASP) is an open source community that targets web application security issues and provides resources to help people prevent them from occurring.

Periodically, OWASP documents a Top 10 list of web application risks (OWASP 2010), based on a rating methodology centered on two main components: the *threat agent* and the system *vulnerability*. Threat agent factors include skill level, motive, opportunity and size. Vulnerability factors include ease of discovery, ease of exploit, awareness, and intrusion detection. Figure 1 illustrates the metrics particularly as they are applied to vulnerabilities. In this figure, we see that there is much unknown about the threat agents themselves; as for the business impacts, these depend on the particular needs and circumstances of the organization involved. So, the primary considerations that come into play are the vulnerabilities themselves, which include the attack vectors (e.g. input tags on HTML forms), the system weaknesses (e.g. injection flaws, coding holes, insufficient authentication, etc.), and the technical impacts of exploitation. Indeed some of these elements tie into the threat agents themselves. For example, the prevalence and detectability of system weakness impacts the opportunity and motive of the agent; the easier the system is to exploit, the greater the likelihood that the agent will continue to pursue his or her exploitation attempt.

Threat Agent	Attack Vector	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impact
?	Easy	Widespread	Easy	Severe	?
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

Figure 1: OWASP metrics for evaluating web application security risks (from OWASP, Top 10 – 2010)

According to the OWASP 2010 Top 10 ratings (OWASP, Top 10 – 2010), SQL Injection Flaws are the number one risk, most notably because they are so easy to exploit and the technical impact of exploitation is so severe. Cross site scripting vulnerabilities are second on the list largely because they are extremely prevalent in web applications and they are also very easy to detect.

OWASP provides cheat sheets for each of the various web risks. The details in each cheat sheet vary but some of the items that may be included in each are an introduction that describes the risk, risk defenses, prevention measures that do not work, rules, and general recommendations. Cheat Sheets may include best practices to minimize a risk from occurring and include excellent links to related articles.

The SQL Injection prevention cheat sheet is particularly good (OWASP, SQL Injection Prevention Cheat Sheet). This cheat sheet includes an introduction, primary defenses, and additional defenses. The introduction describes SQL injection prevention methods that are available to minimize potential issues with end-users inputting invalid data and the use of dynamic queries by programmers. The following are the defenses and examples listed on the web site:

Primary Defenses
- Option #1: Use of Prepared Statements (Parameterized Queries)
- Option #2: Use of Stored Procedures
- Option #3: Escaping all User Supplied Input

Additional Defenses:
- Also Enforce: Least Privilege
- Also Perform: White List Input Validation

The various defenses from the SQL Injection Prevention cheat sheet are presented using several different programming languages. For the purposes of this paper, the focus will be on the Java examples. Not all of the Cheat Sheet Java code

required is included in the examples below. For all of the Java code needed, refer to the OWASP SQL Injection Prevention web site. The following is a basic dynamic query in Java.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "  +
request.getParameter("customerName");
```

The variable called customerName is where the information entered by an end-user on web page is stored and used by the query. If the user enters something like Tom' or '1'='1 the statement will executed as

SELECT account_balance FROM user_data WHERE user_name = 'Tom' or '1' = '1' which will always work since 1 is always equal to 1. Of course this is not what the programmer intended to happen.

The use of parameterized of queries is the first option that should be used to minimize SQL injection. A parameterized query will treat what is entered by an end-user like it has quotes around it and literally look for a customerName of 'Tom' or '1' = '1' . Of course there will not be a customer with that name so the query will not process as SQL injection but as a query looking for a customer name of 'Tom' or '1' = '1'. An example of a Java parameterized query that revises the dynamic query on the website follows where the ? is where the parameter will be added:

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
```

The use of stored procedures is the second option that should be used to minimize SQL injection. Like a parameterized query, a stored procedure query will treat what is entered by an end-user like it has quotes around it and literally look for a customerName of 'Tom' or '1' = '1' (as long as the stored procedure is implemented correctly). Of course there will not be a customer with that name so the query will not process as SQL injection but as a query looking for a customer name of 'tom' or '1' = '1'. An example of a Java stored procedure query that revises the original dynamic query on the website follows where the stored procedure will be used. The stored procedure must be already created on the database server and accept the customerName as a parameter.

The use of escaping is the third option that should be used to minimize SQL injection. OWASP provides a free API that you can download called ESAPI (The OWASP Enterprise Security API). This API will encapsulate anything passed to it with a single quote on each side of it and like a parameterized query and stored procedure, the query will literally look for a customerName of 'Tom' or '1' = '1' . Again, there will not be a customer with that name so the query will not process as SQL injection but as a query looking for a customer name of 'Tom' or '1' = '1'.

In addition to either using parameterized queries, stored procedures, or escaping, validation and white list input validation should be incorporated into a web application *Validation* is where you verify that the data entered into a form is reasonable. For example when expecting a zip code only numbers should be allowed. *White list input validation* is where you explicitly state what is acceptable in a field in a form. For example, a last name may only include upper and lower cases letters, hyphens, and single apostrophe. If anything else is input, an error message would be displayed.

The Cross Site Scripting (XSS) Cheat Sheet is also very good. This cheat sheet includes an introduction and eight prevention rules. The introduction includes information on XSS untrusted data, escaping or encoding, Injection theory and escaping up or down, prevention model. The following eight prevention rules are listed on the web site:

- Rule 0 - Never Insert Untrusted Data Except in Allowed Locations Option #2: Use of Stored Procedures
- Rule 1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
- Rule 2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
- Rule 3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
- Rule 4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
- Rule 5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
- Rule 6 - Use an HTML Policy engine to validate or clean user-driven HTML in an outbound way
- Rule 7 - Prevent DOM-based XSS

In addition, OWASP provide excellent PowerPoint slides in their Presentation web page (OWASP, Category: OWASP Presentations). There are numerous presentations on SQL Injection and other topics

OWASP'S ESAPI AS AN EDUCATIONAL TOOL FOR TEACHING SECURE CODING PRACTICES

There have been many attempts to incorporate secure coding practices into information systems and computer programming curricula. Ralevich and Martinovic (2010) describe an IS security curriculum for which one of the major goals is that students will be able to develop secure programming solutions in object-oriented and procedural programming languages. Taylor and Kaza (2011) presented a coding project specifically geared to various security injections, with the idea to instill a "security mindset" in the student population, and found that student awareness of security coding concepts improved significantly as a result. Although security is an increasing focus in many undergraduate IS programs, most secure *coding* content is done via computer science courses (Perez et al, 2011), which tend to focus on higher-level programming skills for actually implementing security-related algorithms. This material may be too technically demanding for the average information systems student.

For most IS students, the important technical question is not how to *implement* a security algorithm, but how to recognize and *use* the appropriate algorithms for mitigating an organization's IT security vulnerabilities within their software applications. This is where the OWASP Enterprise Security Application Programmer Interface (ESAPI) can be of great assistance as a pedagogical tool.

The ESAPI is a library of classes and functions developed by members of the OWASP community that address the major vulnerabilities typically found in web-based applications. Figure 2 shows the overall ESAPI architecture. While the ESAPI library supports a wide variety of security needs, in this paper we focus on the top two vulnerabilities identified by OWASP: SQL Injection and Cross Site Scripting.

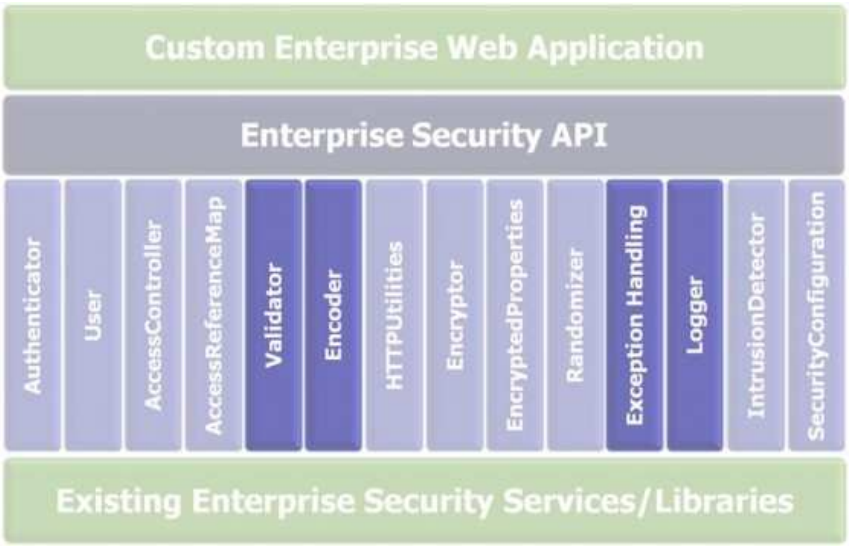

Figure 2: OWASP's enterprise security API (Williams 2008)

As you can see, the ESAPI includes classes that address a wide variety of security concerns, including authentication, access control, input validation and encoding (which will be discussed in detail below), encryption, user logging, and intrusion detection. This library is not intended as a substitute for a particular framework's existing security features, but instead as a supplement. The library is currently available for Java and PHP, and there are efforts by the OWASP community to extend it to .NET.

Because it is free, is accompanied by a strong supporting contingent of OWASP methodologies and best practices, and is very to use, the ESAPI serves as an ideal pedagogical framework for teaching secure programming practices. In the following two sections, we address OWASP and ESAPI approaches to the top two web security risks: SQL injection and cross site scripting.

From a pedagogical perspective, an initial presentation on the OWASP definitions and information is a great way to start people's education. The concepts can be strongly reinforced by a demonstration that shows on how easy it is to make a successful attack. It is one thing to describe how easy SQL Injection can be used in a negative way; it is another to actually demonstrate that fact. The students' awareness of the ease of something like SQL Injection increased significantly based on the classroom response after they saw SQL Injection for the first time. Lastly, a hands-on piece would be ideal to reinforce the concepts even more.

SQL INECTION EXAMPLE

SQL injection flaws are currently ranked as #1 on OWASP's Top 10 risk list. Figure 3 illustrates OWASP's ratings on the important vulnerability indicators for SQL injection.

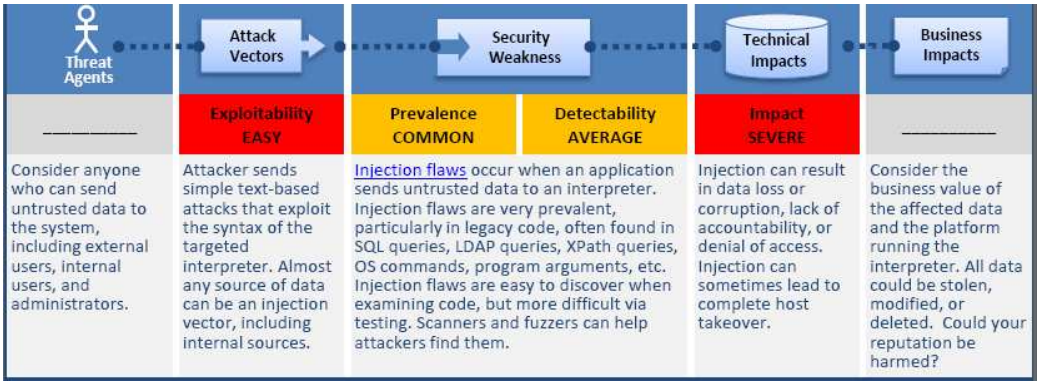

Figure 3: OWASP's assessment of SQL Injection risk (from OWASP, Top 10 – 2010)

Before showing a SQL Injection example it is important to spend some time reviewing the concept of SQL and how injection can occur. SQL is the mechanism needed to select, update, insert, and delete records in a Relational database such Microsoft's SQL Server or Oracle. In general, the term injection refers to inserting characters into a string in the attempt to break out of a *data* context and into a *code* context. In the case of SQL injection, this involves characters with special meaning to a SQL database engine. In the case of XSS, it involves characters with special meaning to web browsers. SQL Injection can occur when an application requires end-users to complete a form where the input from the form is used to dynamically generate a SQL statement. When an end-user enters data in the way that it was intended (i.e., the user name includes only valid last user name type characters), the SQL statement will perform as desired and expected. However, SQL Injection occurs when an end-user enters special characters in a form in an attempt to cause a SQL statement to perform in a manner that was not intended (notice in the examples below, an extra quote and hyphens were added). There are several other considerations, such as permissions, that will impact of the effect of the SQL statement but this illustrates the main concern: SQL Injection can cause an application to execute SQL that was not intended.

For the purposes of the following example a typical login screen will be used with Java as the programming language and SQL Server as the database. Initially, the login screen will appear like the following figure.

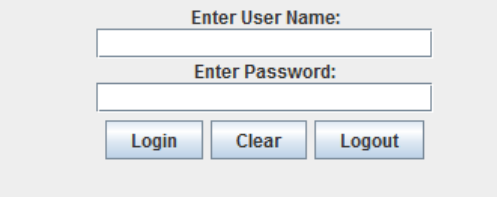

Figure 4: Typical user login screen

Initally the Java code will use a dynamic query that is written as follows where LoginCredentials was table created to emmulate the storage of a username and password, and EmployeeID a link to an Employee table. None of the data is encrypted for the purposes of the example in order to show the actual query results.

```
String SQL = "SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials] where USERNAME = '"
+ userName + "' and PASSWORD = '" + passWord + """;
```

If a valid username and password is entered as shown in the following figure:

Figure 5: Legitimate user login

The dynamic query will run as SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials] where USERNAME = 'colecb' and PASSWORD = 'colecb'.

However, if SQL Injection is attempted as shown in the following figure

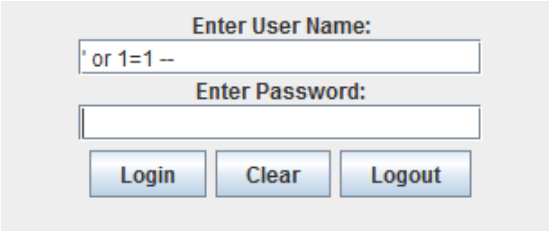

Figure 6: SQL injection attempt

The dynamic query will run as SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials] where USERNAME = '' or 1=1 --' and PASSWORD = ''. Notice in the code above that everthing is commented after the 1=1 statement thus the statement is really processed as SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials] where USERNAME = '' or 1=1. The results are that anyone could login to the system this way.

If we change the dynamic query to a parametrized query the query will run as SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials] where USERNAME = ? and PASSWORD = ?. The login used in this attempt will not work since the query looks for a user name of ' or 1=1 and one does not exist.

An invalid login will also occur if a stored procedure is called from Java that looks like the following:

```
PROCEDURE [dbo].[sp_getUserName] @UserName varchar(25), @Password varchar(25) AS BEGIN
```

```
SELECT USERNAME, PASSWORD, EMP_ID FROM [LoginCredentials]
        where USERNAME = @UserName
        and PASSWORD = @Password
END
```

Escaping is the third option to consider as a way to minimize the possibility of SQL Injection, but according to OWASP this should be a last resort (OWASP, SQL Injection Prevention Cheat Sheet). The term escaping (also called encoding) refers to a programmatic means of converting code-triggering characters in strings to harmless (i.e. data context) characters. If the effort to update a legacy system to parameterized queries or stored procedures is too time consuming or risky, then escaping should be considered. Remember that OWASP provides an API for these purposes. The following figure shows a possible input of the string ' or 1=1 --. The first System.out.println example uses escaping and the second one does not.

```
public class esapiSqlInjectionPrevention {

    public static void main(String[] args){


        String badString = "' or 1 = 1;--'";
        Codec ORACLE_CODEC = new OracleCodec();


        // need to have the ESAPI.properties file in a readable location
        String encoded = ESAPI.encoder().encodeForSQL( ORACLE_CODEC, badString);


        System.out.println(encoded);
        System.out.println(badString);
```

Figure 7: Java code containing OWASP ESAPI library call for encoding SQL

Notice that the figure below displays the output of the initial string in two ways. The first example below shows the SQL with the API. Notice that this code cannot be used for SQL Injection. The second example shows the SQL without escaping. This condition allows for the possibility of SQL Injection.

```
SecurityConfiguration for Logger.LogServerIP not found in ESAPI.properties. Using default: true
SecurityConfiguration for Logger.ApplicationName not found in ESAPI.properties. Using default: DefaultName
'' or 1 = 1;--''
' or 1 = 1;--'
```

Figure 8: Output of encoded and unencoded strings from Figure 7's code

It is worth noting that the SQL presented above does not include sophisticated or advanced statements; they are basic ones that are taught in any introductory database class. People abusing SQL Injection can be performed by anyone who knows basic SQL.

CROSS SITE SCRIPTING EXAMPLE

Cross site scripting (XSS) is currently #2 on OWASP's Top 10 risk list. Figure 9 illustrates OWASP's ratings on the important vulnerability indicators for XSS.

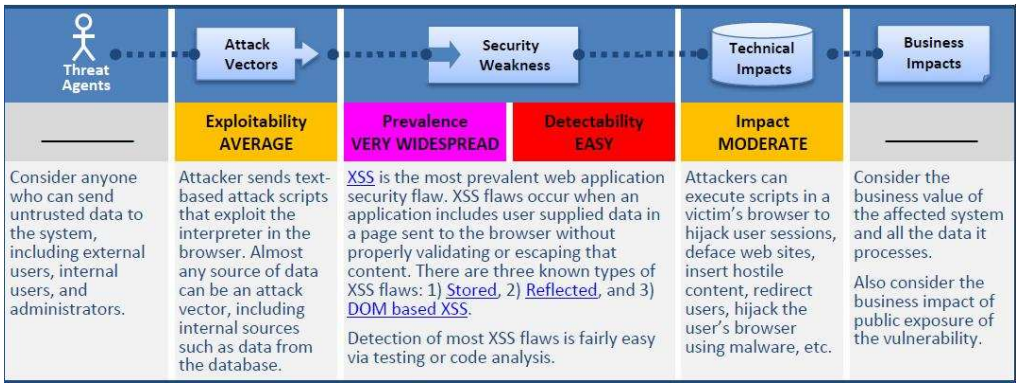

Figure 9: OWASP's assessment of XSS risk (from OWASP, Top 10 – 2010)

XSS vulnerabilities are especially prevalent in web applications that allow HTML and/or JavaScript to be input by end users. Social networking sites and email applications are two common examples. In this section, we consider an example of a job posting site (like a monster.com) in which job applicants submit information about themselves including a resume, and employers search for and read the information input by the applicants. In this scenario, we imagine that a job applicant (the threat agent) inserts HTML and JavaScript code into his or her posted resume, and we explore the potential impact that this action can have. We follow with a simple OWASP ESAPI object reference and method call that mitigates this intrusion attempt.

Consider a job applicant page in which the user enters a name, email address and "About Me" blurb, which could be a resume and/or other personal information about the job applicant. Assuming the About Me posting is a large enough text area on the web page, a user might enter quite a bit of text. Figure 10 illustrates such a page. Note that the user has entered HTML and JavaScript code into the About Me area.

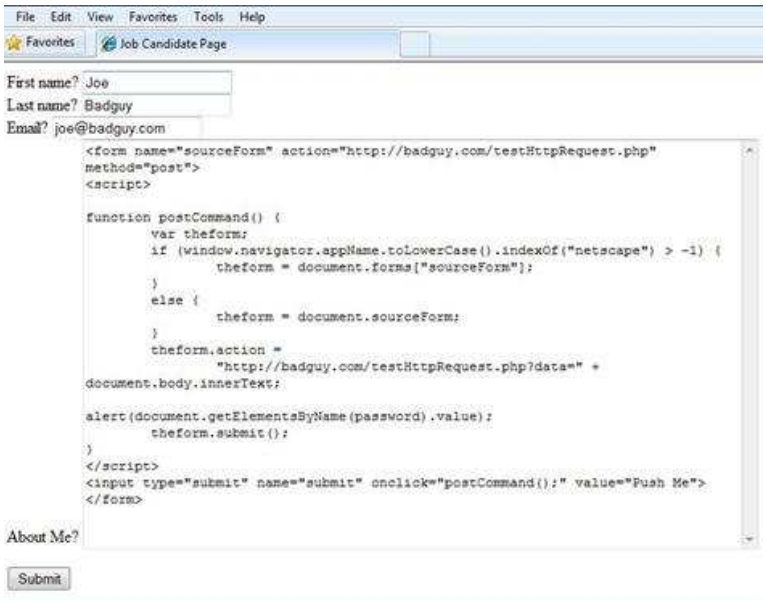

Figure 10: Example XSS code attempt by evil job candidate (HTML form with JavaScript)

In the above web page, the user has inserted an HTML form with a Submit <input> tag, along with a <script> tag with a JavaScript function that will be invoked when an unsuspecting user (a potential employer in this example) clicks the button. When the user submits this information, the web application will insert it into the database. As a result of this input, and in the absence of preventative validation and/or encoding measures in the web application itself, the host's database will store the About Me information exactly as entered.

Note that none of the items in this entry involve SQL injection attempts. Unlike the examples from the previous section, there is no danger of doing damage to the web application host environment by injecting into the SQL engine itself. Therefore, the precautions mentioned above are of no use. Parameterized queries are important for preventing SQL injection, but are useless for preventing XSS.

An employer who subsequently logs in may be presented with a list of job candidates, as shown below.

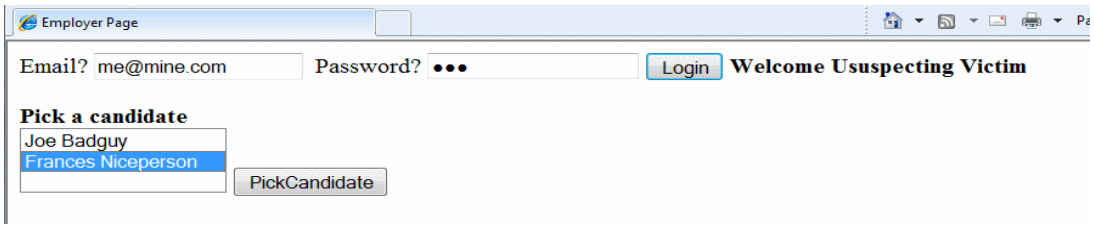

Figure 11: Unsuspecting employer selects a job candidate to look for

Selecting an innocent job candidate may result in display of that person's information:

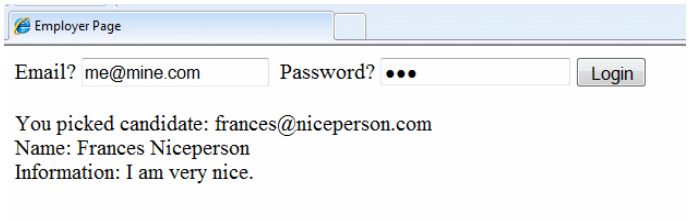

Figure 12: Selecting a legitimate job candidate results in information display from database query

But selecting the hacker who entered the HTML and JavaScript would produce this result:

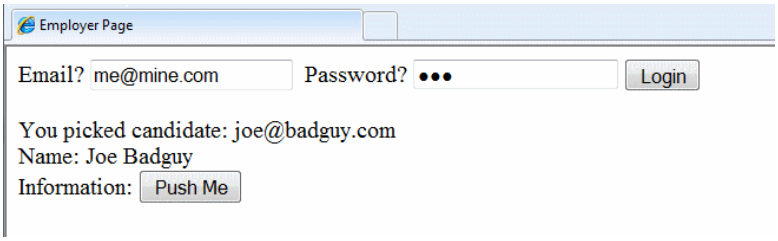

Figure 13: Selecting the evil candidate results in XSS injection code displaying the form and button; clicking the button sends sensitive data to the bad guy's web site

The Push Me button is a classic example of *social engineering*, an attempt to manipulate the employer's behavior (Workman, 2007). If the employer pushes that button, his or her email and password will be sent to the hacker's web site, where it could be stored into a database and/or used for further nefarious activities. Note that social engineering is not necessary for XSS; it was only used in this example for illustrative purposes.

The solution for this particular example is to perform what is commonly called *escaping*, or to use another term, *encoding*. For example, OWASP's Java-based ESAPI includes an Encoder interface (implemented by a **DefaultEncoder** class). This interface includes methods for a wide variety of encodings, including SQL, HTML tags and attributes, JavaScript, CSS, LDAP, URL, XML tags and attributes, and XPath. (OWASP, Java API documentation). In general, these encoders work by converting the symbols that could be used for injection of the relevant interpreters into harmless substitutes; i.e., mitigating the threat agent's attempt at converting a data context into a code context. For example, key HTML injection symbols include the open and close angle brackets "<" and ">". HTML encoding converts these symbols into "<" and ">" respectively (http://www.w3schools.com/html/html_entities.asp).

For, example, assume this was the original code for inserting a job applicant's data into the database via a parameterized SQL statement and parameter settings:

```
// 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();
```

Figure 14: Unprotected code setting SQL parameters directly from form fields

The following change to the code mitigates the injection attempt by encoding the HTML symbols:

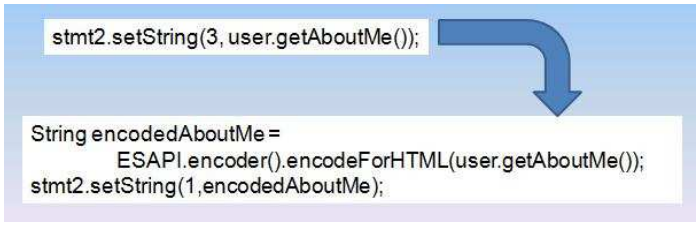

Figure 15: Utilizing OWASP ESAPI HTML encoding functionality

The resulting HTML encoding will result in the database containing this data instead of the original data entered by the job applicant:

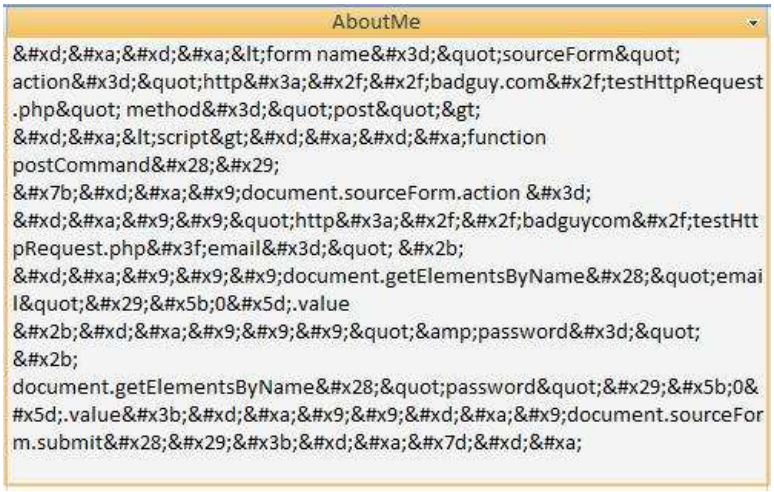

Figure 16: Resulting string stored to database replaces HTML symbols with their encoded values

Therefore, the display that will occur to the employer when selecting the hacker will look like this:

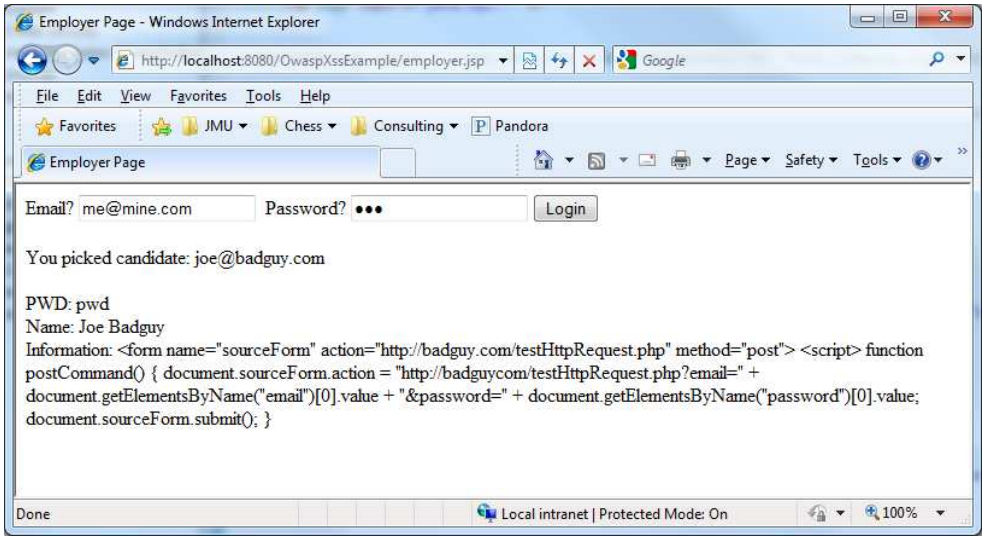

Figure 17: Bud guy's injection attempt becomes a harmless display on employers screen

In this simple case, the solution was to perform HTML encoding, which is what the **encodeforHTML** method does. According to OWASP the best practices recommendations, HTML entity encoding by itself is not sufficient for guarding against XSS risks (OWASP, XSS Cross site Scripting Prevention Cheat Sheet). But it is one tool in a web developer's arsenal, and when used in conjunction with other tools and techniques, it can help minimize these kinds of threats. It is also important to consider that sometimes a web developer may *want* users to be able to enter certain (but not all) HTML and even JavaScript code. For these purposes, OWASP's ESAPI includes an HTML policy engine and sanitizer that can be used to selectively allow a subset of HTML symbols and/or JavaScript functions (Weinberger et al 2011). Once students learn the

methodologies espoused by OWASP and gain practice in using the simpler classes of the ESAPI library, they can get more advanced experience by utilizing the policy engine and sanitizer provided for more refined encoding and validation.

CONCLUSION

The OWASP website has a significant amount of information that includes guidelines and recommendations related to security risks, and software tools and libraries which enable programmers to put these guidelines into practice. Our paper has only scratched the surface on the types of possible security attacks and the information available. Using the OWASP is an excellent resource and should be considered when training students about best practices with regard to securing web applications. Additional training using demonstrations and hands-on exercises will enhance the learning experience. As you can see SQL Injection and Cross Site Scripting are easy to take advantage of but equally as easy to minimize or eliminate. It will be less costly if people understand how to write code correctly so that they will not have to rewrite them or allow the business to suffer as a result of web attacks. Integrating the knowledge and skills OWASP provides, along with the security features of the ESAPI, into a school's IS curriculum helps prepare students to respond effectively to the information security threats they will encounter throughout their professional lives.

REFERENCES

1. Bhushan, A. (2009) Review on master hacker Albert Gonzalez: SQL injection attacks- led to Heartland, Hannaford, 7-Eleven breaches, *CEOWorld Magazine*, available at http://ceoworld.biz/ceo/2009/08/19/review-on-master-hacker-albert-gonzalez-sql-injection-attacks-led-to-heartland-hannaford-7-eleven-breaches (checked 5/4/2012).
2. Grossman, J. (2007). Cross-site scripting worms and viruses: the impending thread & the best defense, WhiteHat Security Whitepaper, available at https://www.whitehatsec.com/assets/WP5CSS0607.pdf (checked 5/4/2012).
3. Open Web Application Security Project (checked 5/4/2012) OWASP presentations, https://www.owasp.org/index.php/Category:OWASP_Presentations.
4. Open Web Application Security Project (checked 5/4/2012) OWASP enterprise security API, https://www.owasp.org/index.php/Esapi.
5. Open Web Application Security Project (checked 5/4/2012) OWASP Java API, http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/index.html.
6. Open Web Application Security Project (checked 5/4/2012) SQL injection prevention Cheat Sheet, https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.
7. Open Web Application Security Project (checked 5/4/2012) Top 10 – 2010: the ten most critical web application security risks, http://owasptop10.googlecode.com/files/OWASP Top 10 - 2010.pdf.
8. Open Web Application Security Project (checked 5/4/2012) XSS (cross site scripting) prevention cheat sheet, https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.
9. Open Web Application Security Project (checked 5/4/2012) OWASP Risk Rating Methodology https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
10. Perez, L, Cooper, S., Hawthorne, E., Wetzel, S, Brynielsson, J, Gokce, A, Impagliazzo, J, Khmelevsky, Y, Klee, K, Leary, M, Philips, A, Pohlmann, N, Taylor, B, Upadyaya, S.(2011), Information assurance education in two- and four-year institutions, in Liz Adams, Justin Joseph Jurgens (Eds.) *ITiCSE '11 Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - working group reports,* June 27-29, Darmstadt, Germany, 39-53.
11. Ralevich V and Martinovic, D. (2010) Designing and implementing an undergraduate program in information systems security *Education and Information Technologies: Special Issue: Information Systems Curriculum* 15,4 293-315.

12. Taylor, B. and Kaza, S. (2011) Security injections: modules to help students remember, understand, and apply secure coding techniques. *ITiCSE '11 Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. June 27-29, Darmstadt, Germany, 3-7.
13. Weinberger, J., Saxeena, P., Akhawe, D., Finifter, M., Shin, R., Song, D. (2011) A systematic analysis of XSS sanitation in web application frameworks. *ESORICS'11 Proceedings of the 16th European conference on Research in computer security*. Sep. 12-14, Leuven, Belgium, 150-171.
14. Williams, J. (2008) Establishing an enterprise security API to reduce application security costs, Aspect Security, http://owasp-esapi-java.googlecode.com/files/OWASP%20ESAPI.ppt
15. Workman, M. (2007) Gaining access with social engineering: an empirical study of the threat. *Information Systems Security*, 16, 315-331.