The OWASP Top Ten

- The Top Ten Vulnerabilities for 2010
- Mitigating the Vulnerabilites

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is OWASP?

- From http://www.owasp.org:
- The Open Web Application Security Project (OWASP) is a 501c3 not-for-profit worldwide charitable organization focused on improving the security of application software. Our mission is to make application security visible, so that people and organizations can make informed decisions about true application security risks. Everyone is free to participate in OWASP and all of our materials are available under a free and open software license



You can read more about the OWASP organization at:

`http://www.owasp.org/index.php/OWASP:About`

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The OWASP Top Ten for 2010

- OWASP publishes a list of the top-ten current application vulnerabilities, most of which are geared toward Web applications:

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

The Top Ten for 2010 supercedes the previous list, which was released in 2007.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Injection

- Injection flaws occur when untrusted data is sent to an interpreter (e.g. SQL database) as part of a command or query
- An attacker can trick the interpreter into executing unintended commands for accessing unauthorized data

```
HTML Form                Program
---------                -------
                       String s =
                          req.getParameter("serial");

<input type="text"     String SQL =
  name="serial">         "SELECT FROM Users WHERE
                         serial='" + s + "'";

                       ResultSet rs =
                          stmt.executeQuery(SQL);
```


The most common form of injection is SQL injection, but the problem's not limited to databases. For example, LDAP directories could be at risk.

For a good explanation of how SQL injection works, see:

http://unixwiz.net/techtips/sql-injection.html

In the code shown here, an attacker could "inject" unintended SQL by entering a string that contained SQL "escape" syntax.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Injection in Java

- Your primary protection against SQL injection is to use JDBC *PreparedStatements* to construct SQL from user input

```
HTML Form                Program
---------                -------
                        String s =
                           req.getParameter("serial");

<input type="text"      String SQL =
  name="serial">          "SELECT FROM Users WHERE
                          serial=?";

                        PreparedStatement stmt =
                           conn.prepareStatement(SQL);
                        stmt.setString(1, serial);
                        ResultSet rs =
                           stmt.executeQuery();
```


This technique is effective since the JDBC driver validates SQL before executing it, thus reducing the likelihood of an injection.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Cross-Site Scripting

- XSS flaws occur when a Web application passes untrusted data to a Web browser
- Attackers use XSS to execute scripts in the user's browser to hijack session, deface Web sites or redirect the user to malicious sites

```
JSP With Input Form
-------------------
<form action="output.jsp">
  Enter your name: <input type="text" name="name">
  <input type="submit" value="Submit">
</form>

Output JSP
----------

You entered: ${param.name}
```

User Input:

<script type="text/javascript">alert('Got you!');</script>

Note that the user input contains a simple JavaScript. This script is harmless (it displays an alert box), but attackers can write much more malicious scripts to hijack sessions and so forth.

The problem here is that the JSP expression (${...}) simply displays the input text within a generated output page.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating XSS in Java Web Apps

- If your Web application accepts input from the user and then redisplays it in a Web page, you should ensure that *XML escaping* is enabled
- Note that the JSP *expression language* does NOT escape XML by default

```
Output JSP
----------

Bad:  ${param.name}
Good: ${fn:escapeXml(param.name)}
Good: <c:out value="${param.name}" />
```

A JSP can minimize the likelihood of XSS by "escaping" untrusted input before outputting it. Note that the "fn:escapeXml" and "c:out" are functions from the JSP Standard Tag Library (JSTL) and that "c:out" escapes content by default. Be aware, however, that this is not complete protection, since an attacker could submit the input encoded in hexadecimal. So before escaping, you should first decode any hexadecimal text in the user input. See the following article for details:

http://today.java.net/article/2005/09/19/handling-java-web-application-input-part-2

Also note that if you are using JavaServer Faces (JSF) with the Facelets view technology, it automatically escapes output, too.

You can also encounter XSS if your application has a servlet that directly outputs untrusted user input.

See:

http://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet a

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

for more details.

Broken Authentication and Sessions

- This vulnerability occurs when applications do not correctly implement authentication and session management
- An attacker can compromise passwords or session tokens

```
http://www.goliath.com/myServlet;jsessionid=863F373633D316
```


HTTP is a "stateless" protocol, so Web applications that need to maintain state (e.g. a shopping cart) typically use browser cookies to maintain state between requests. But some developers use also "URL rewriting" in case the user disables browser cookies; if so, then the session information appears in the URL itself.

This potentially opens the application to attack since an attacker can see the session ID via a network sniffer or XSS. An attacker can use the sessionID to mount an attack. For more details in the Java world, see:

```
http://www.javapractices.com/topic/TopicAction.do?Id=226
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Broken Sessions in Java Web Apps

- One possible vulnerability is Web applications that use *URL rewriting* to maintain session state
- Other vulnerabilities relate to not properly invalidating sessions at the end of a flow or using long session timeouts
- Another attack involves applications that store user passwords in the clear on a server-side database - once an attacker compromises the database, they have access to all users' passwords

---

See:

`http://www.owasp.org/index.php/ASVS`

for more details.

Java Web developers should avoid using URL rewriting and instead use browser cookies to maintain state.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Insecure Direct Object References

- This vulnerability occurs when a program exposes a reference to an internal "object" such as a file, directory or database key
- Without proper access protection, an attacker can use the "object" to access unauthorized data

```
// URL from authorized access

http://www.goliath.com/myServlet;accountID=1234

// Attack to access unauthorized account

http://www.goliath.com/myServlet;accountID=7777
```


The idea here is that an attacker discovers a reference and then attempts access using forged reference, attempting to access unauthorized information.

In the example shown here, after the attacker learns that '1234' is a legal account, the attacker attempts to access acount number 7777.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Insecure Object Vulnerabilities

- Applications should not use explicit database keys in user interfaces, but should use "magic numbers" that the application maps to the actual keys
- Developers and administrators should ensure that any file or directory access from an application refers to a properly secured file system

---

See

http://www.owasp.org/index.php/ESAPI

for more details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Cross-Site Request Forgery

- A CRSF attack forces a logged-in victim's browser to use a forged request
- The attacker can supply parameters on the forged request to cause the vulnerable application to perform unintended actions

This attack often occurs on Internet "forum" sites where users can enter "img" tags but not script, or via spam email.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

CRSF Example

Step 1: Sue logs into bank Web application, which establishes a session

Step 2: Sue browses to a Web page containing a form to transfer money to an account

Step 3: Sue submits the form

```
POST  http://mybank.com/transfer
. . .
acct=1234
amt=100
```

Web app uses Sue's session to authenticate and performs transfer

Step 4: Evil Sam sends Sue an HTML email with the following content:

```
<img width=1" height="1" src="http://mybank.com/transfer?acct=7777&amt=1000">
```

Browser in HTML email client sends a GET request using Sue's session
Insecure Web application transfers $1000 to Evil Sam's account

---

The email contains a virtually invisible picture that will not render in the browser, but the browser WILL make the request specified by the "src" attribute, triggering the bank transfer.

Note for this attack to work, Evil Sam must know a valid account number and must trick Sue into opening the email before she logs out of the bank application or the session times out.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating CRSF

- Don't process GET requests when POST is expected (but recognize that attackers can forge POSTs, too)
- To effectively mitigate CRSF, applications must ensure that all forms contain a unique security token that that the application validates when the request arrives
- Some Java application servers, e.g. Tomcat version 7 and IBM WebSphere (via the *ProjectZero* initiative) provide automatic, configurable CRSF protection

---

See

http://en.wikipedia.org/wiki/Cross-site_request_forgery
http://www.owasp.org/index.php/ESAPI
http://www.projectzero.org/
http://www.developer.com/java/ent/article.php/3891171/Tomcat-7-Debuts-for-Java.htm

for more details.

The OWASP group provides a CRSF testing tool at

http://www.owasp.org/index.php/CSRFTester

to help detect CRSF vulnerabilities.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating CRSF, cont'd

Step 1: Sue logs into bank Web application, which establishes a session

Step 2: Sue browses to a Web page containing a form to transfer money to an account

Application generates a unique "token" and stores it in Sue's session on the server and as a hidden field in the form sent back to Sue's browser

Step 3: Sue submits the form

```
POST  http://mybank.com/transfer
. . .
acct=1234
amt=100
token=7hjfsadf834343
```

Web app first validates token against the value stored in Sue's session and only then performs transfer

Step 4: Evil Sam sends Sue an HTML email with the following content:

```
<img width=1" height="1" src="http://mybank.com/transfer?acct=7777&amt=1000">
```

Browser in HTML email client sends a GET request using Sue's session
Web application rejects request since it doesn't contain the required token

---

By requiring a unique security token associated with the form and the legitimate user's session, we make it difficult for Evil Sam to forge a request (he would also need to guess the security token).

To further reduce risk, the application can "age" the token to minimize the time window in which the application is vulnerable to CRSF.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Security Misconfiguration

- This vulnerability occurs when applications use frameworks, libraries, application servers and databases that contain vulnerabilities

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Security Misconfiguration

- Ensure that all accessory software is patched and up to date and that there's a well defined process for updating
- Ensure that unnecessary services, ports and unused pages are disabled
- Ensure that security settings in your application server are configured properly

See:

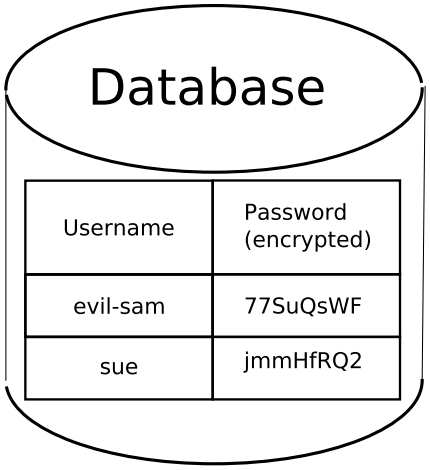
http://www.owasp.org/index.php/Configuration

for more details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Insecure Cryptographic Storage

- This vulnerability occurs when applications do not properly protect sensitive data, such as credit card numbers and passwords



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Insecure Storage

- Ensure that sensitive data is encrypted using strong encryption algorithms
- Ensure that access to sensitive data is limited to authorized users

See

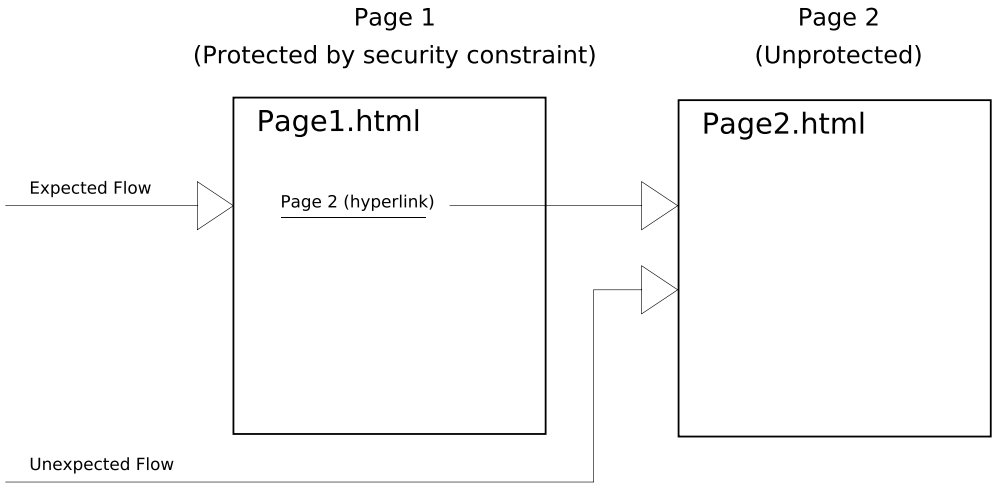
http://www.owasp.org/index.php/ASVS

for more details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Failure to Restrict URL Access

- This vulnerability occurs when applications fail to perform access control checks whenever protected resources are accessed, but only protect links and submit buttons
- An attacker can forge URLs that access restricted pages and bypass access control checks



Many Web server infrastructures, including Java Enterprise Edition (JEE), let you configure security constraints to restrict access to pages in a Web application.

This vulnerability results from the failure to protect all sensitive pages, but to instead only protect the "landing" pages of the application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Failure to Restrict URL Access

- Deny all access to protected pages by default and use explicit grants for specific users on specific pages
- Use role-based security to make restriction easier to maintain when user population is dynamic

---

See

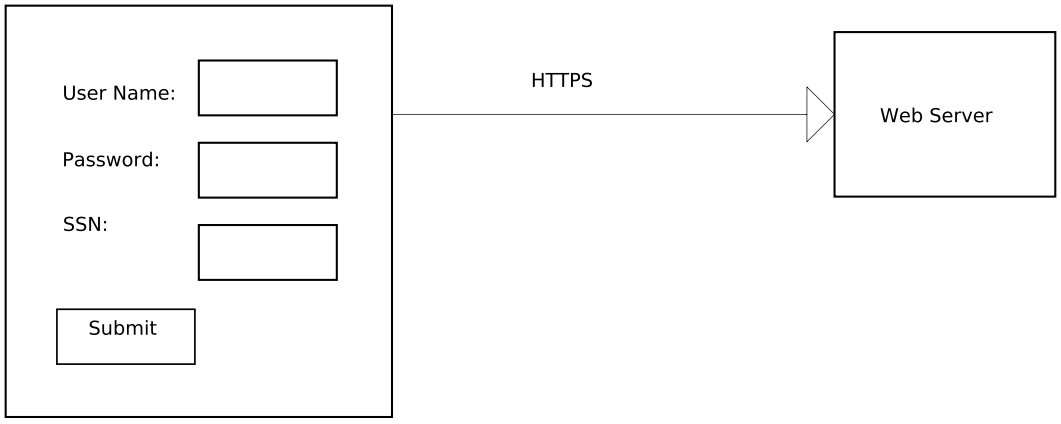
http://www.owasp.org/index.php/Guide_to_Authorization

for more details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Insufficient Transport Layer Protection

- This vulnerability occurs when applications fail to use strong encryption (e.g. SSL) as sensitive data passes over an insecure network



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Insufficient Transport Layer Protection

- Use SSL whenever sensitive data is transmitted over an insecure network, especially for login credentials
- Avoid using mixed secure and insecure content on a given page
- Ensure that the SSL certificates are up to date, use strong encryption and passwords and are properly configured

---

See

http://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

for details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unvalidated Redirects and Forwards

- This vulnerability occurs when a Web application redirects or forwards to another page using untrusted data to determine the destination
- An attacker can redirect to phishing or malware sites or forward to unauthorized pages

```
Java Servlet Code
_________________

String dest = req.getParameter("form_field");

. . .

resp.sendRedirect(dest);
```


Forwarding and redirecting are Web application techniques to transfer control programmatically. Forwarding is generally limited to within a Web application, while redirection is part of the HTTP specification and can transfer to any URL.

In the example shown here, an attacker could provide a URL in a form parameter that would cause the servlet to redirect the user to a potentially malicious site.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mitigating Unvalidated Redirects and Forwards

- Avoid using forwarding and redirecting unless absolutely necessary
- If an application uses input from the user to determine the destination, validate the destination address

---

See

http://www.owasp.org/index.php/Open_redirect

for more details.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The OWASP Java Project

- The OWASP group has a *Java Project* whose goal is to enable Java and J2EE developers to build secure applications efficiently
- This project has the following subcategories:
    - J2EE Security for Architects
    - J2EE Security for Developers
    - J2EE Security for Deployers
    - J2EE Security for Security Analysts and Testers

This OWASP project has specific details for security in the JEE and JSE environments. See

`http://www.owasp.org/index.php/Category:OWASP_Java_Project`

for more information.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About The Open Web Application Security Project
- The OWASP Top Ten Vulnerabilities
- How to mitigate the vulnerabilities

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum