Top 10 Web Security Controls
(1) Query Parameterization (PHP PDO)

```php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)" );

$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);
```
SqlConnection objConnection = new SqlConnection(_ConnectionString);
objConnection.Open();
SqlCommand objCommand = new SqlCommand(
    "SELECT * FROM User WHERE Name = @Name AND Password =
    @Password",
    objConnection);
objCommand.Parameters.Add("@Name", NameTextBox.Text);
objCommand.Parameters.Add("@Password", PasswordTextBox.Text);
SqlDataReader objReader = objCommand.ExecuteReader();
if (objReader.Read()) { ...}
double newSalary = request.getParameter("newSalary");
int id = request.getParameter("id");
PreparedStatement pstmt =
    con.prepareStatement("UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setDouble(1, newSalary);
pstmt.setInt(2, id);

Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
Query Parameterization (Ruby)

# Create
Project.create!(:name => 'owasp')

# Read
Project.all(:conditions => "name = ?", name)
Project.all(:conditions => { :name => name })
Project.where("name = :name", :name => name)

# Update
project.update_attributes(:name => 'owasp')

# Delete
Project.delete(:name => 'name')
Query Parameterization (Cold Fusion)

```cfml
<cfquery name="getFirst" dataSource="cfsnippets">
    SELECT * FROM #strDatabasePrefix#_courses WHERE intCourseID =
    <cfqueryparam value=#intCourseID# CFSQLType="CF_SQL_INTEGER">
</cfquery>
```
Query Parameterization (PERL)

```perl
my $sql = "INSERT INTO foo (bar, baz) VALUES ( ?, ? )";
my $sth = $dbh->prepare( $sql );
$sth->execute( $bar, $baz );
```
XSS: Why so Serious?

- Session hijacking
- Site defacement
- Network scanning
- Undermining CSRF defenses
- Site redirection/phishing
- Load of remotely hosted scripts
- Data theft
- Keystroke logging
Danger: Multiple Contexts

Browsers have multiple contexts that must be considered!
XSS in HTML Attributes

<input type="text" name="comments" value="UNTRUSTED DATA">

<input type="text" name="comments" value="hello" onmouseover="/*fire attack*/">

- Attackers can add event handlers:
  - onMouseOver
  - onLoad
  - onUnload
  - etc...
XSS in Source Attribute

- User input often winds up in src attribute
- Tags such as `<img src=""/>`  `Here is your requested image:` `<p/>`
  `<img src="mymap.jpg"/>` `<p/>
- Example Request:
  ```
  http://example.com/viewImage?imagename=mymap.jpg
  ```
- Attackers can use `javascript:*attack*/` in src attributes
URL Parameter Escaping

- Escape **all** non alpha-num characters with the %HH format

`<a href="/search?data=UNTRUSTED DATA">`

- Be careful not to allow untrusted data to drive entire URL’s or URL fragments

- This encoding only protects you from XSS at the time of rendering the link

- Treat DATA as untrusted after submitted
XSS in the Style Tag

- Applications sometimes take user data and use it to generate presentation style

```html
169  body {
170   font-size: 0.8em;
171   color: black;
172   font-family: Geneva, Verdana Arial, Helvetica, sans-serif;
173   background-color: white;
174   margin: 0;
175   padding: 0;
176 }
```

- Consider this example:

http://example.com/viewDocument?background=white
CSS Pwnage Test Case

<div style="width: <%=temp3%>;"> Mouse over </div>

```
temp3 =
    ESAPI.encoder().encodeForCSS("expression(alert(String
    .fromCharCode (88,88,88)))");
```

<div style="width: expression\28 alert\28 String\2e
    fromCharCode\20 \28 88\2c 88\2c 88\29 \29 ;"> Mouse over </div>

- Pops in at least IE6 and IE7.

lists.owasp.org/pipermail/owasp-esapi/2009-February/000405.html
**Javascript Context**

- Escape **all** non alpha-num characters with the \xHH format

```html
<script>
var x='UNTRUSTED DATA';
</script>
```

- You're now protected from XSS at the time data is assigned

- **What happens to x after you assign it?**
Best Practice: DOM Based XSS Defense

- Untrusted data should only be treated as displayable text
- JavaScript encode and delimit untrusted data as quoted strings
- Use `document.createElement("...")`, `element.setAttribute("...","value")`, `element.appendChild(...)`, etc. to build dynamic interfaces
- Avoid use of HTML rendering methods
- Understand the dataflow of untrusted data through your JavaScript code. If you do have to use the methods above remember to HTML and then JavaScript encode the untrusted data
- Avoid passing untrusted data to `eval()`, `setTimeout()` etc.
- Don’t `eval()` JSON to convert it to native JavaScript objects. Instead use `JSON.stringify()` and `JSON.parse()`
- Run untrusted scripts in a sandbox (ECMAScript canopy, HTML 5 frame sandbox, etc)
# (2) XSS Defense by Data Type and Context

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body</td>
<td>HTML Entity Encode</td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute</td>
<td>Minimal Attribute Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid <code>javascript:</code> URL’s, Attribute encoding, safe URL verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>Strict structural validation, CSS Hex encoding, good design</td>
</tr>
<tr>
<td>HTML</td>
<td>HTML Body</td>
<td>HTML Validation (JSoup, AntiSamy, HTML Sanitizer)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>DOM XSS Cheat sheet</td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td>Sandboxing</td>
</tr>
<tr>
<td>JSON</td>
<td>Client parse time</td>
<td>JSON.parse() or json2.js</td>
</tr>
</tbody>
</table>

**Safe HTML Attributes include:** align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width
Attacks on Access Control

- **Vertical Access Control Attacks**
  - A standard user accessing administration functionality
  - “Privilege Escalation”

- **Horizontal Access Control attacks**
  - Same role, but accessing another user's private data

- **Business Logic Access Control Attacks**
  - Abuse of workflow
Best Practice: Code to the Activity

```java
if (AC.hasAccess(ARTICLE_EDIT, NUM)) {
    //execute activity
}
```

- Code it once, never needs to change again
- Implies policy is persisted/centralized in some way
- Requires more design/work up front to get right
Best Practice: Use a Centralized Access Control Center

**In Presentation Layer**

```java
if (ACL.isAuthorized(VIEW_LOG_PANEL))
{
    <h2>Here are the logs</h2>
    <%=getLogs();%>
}
```

**In Controller**

```java
try (ACL.assertAuthorized(DELETE_USER))
{
    deleteUser();
}
```
(3) Access Control Positive Patterns

- Code to the activity, not the role
- Centralize access control logic
- Design access control as a filter
- Fail securely (deny-by-default)
- Apply same core logic to presentation and server-side access control decisions
- Server-side trusted data should drive access control
- Provide privilege and user grouping for better management
- Isolate administrative features and access
Anatomy of an CSRF Attack

Consider a consumer banking application that contains the following form:

```html
<form action="https://bank.com/Transfer.asp" method="POST" id="form1">
<p>Account Num: <input type="text" name="acct" value="13243"/></p>
<p>Transfer Amt: <input type="text" name="amount" value="1000" /></p>
</form>
<script>document.getElementById('form1').submit(); </script>
```
(4) Cross Site Request Forgery Defenses

- **Cryptographic Tokens**
  - Primary and most powerful defense. Randomness is your friend.

- Request that cause side effects should use (and require) the POST method
  - Alone, this is not sufficient

- Require users to re-authenticate
  - Amazon.com does this *really* well

- Double-cookie submit
  - Decent defense, but no based on randomness, based on SOP
Authentication Dangers

- Weak password
- Login Brute Force
- Username Harvesting
- Session Fixation
- Weak or Predictable Session
- Plaintext or poor password storage
- Weak "Forgot Password" feature
- Weak "Change Password" feature
- Credential or session exposure in transit via network sniffing
- Session Hijacking via XSS
(5) Authentication Defenses

- 2FA
- Develop generic failed login messages that do not indicate whether the user-id or password was incorrect
- Enforce account lockout after a pre-determined number of failed login attempts
- Force re-authentication at critical application boundaries
  - edit email, edit profile, edit finance info, ship to new address, change password, etc.
- Implement server-side enforcement of credential syntax and strength
(6) Forgot Password Secure Design

- Require identity and security questions
  - Last name, account number, email, DOB
  - Enforce lockout policy
  - Ask one or more good security questions

- Send the user a randomly generated token via out-of-band method
  - email, SMS or token

- Verify code in same web session
  - Enforce lockout policy

- Change password
  - Enforce password policy
(7) Session Defenses

- Ensure secure session ID’s
  - 20+ bytes, cryptographically random
  - Stored in HTTP Cookies
  - Cookies: Secure, HTTP Only, limited path

- Generate new session ID at login time
  - To avoid *session fixation*

- Session Timeout
  - Idle Timeout
  - Absolute Timeout
  - Logout Functionality
(8) Clickjacking Defense

- **Standard Option: X-FRAME-OPTIONS Header**

  // to prevent all framing of this content
  response.addHeader( "X-FRAME-OPTIONS", "DENY" );

  // to allow framing of this content only by this site
  response.addHeader( "X-FRAME-OPTIONS", "SAMEORIGIN" );

- **Frame-breaking Script defense:**

  <style id="antiClickjack">body{display:none}</style>
  <script type="text/javascript">
    if (self == top) {
      var antiClickjack =
      document.getElementById("antiClickjack");
      antiClickjack.parentNode.removeChild(antiClickjack);
    } else {
      top.location = self.location;
    }
  </script>
public String hash(String plaintext, String salt, int iterations) throws EncryptionException {
    byte[] bytes = null;
    try {
        MessageDigest digest = MessageDigest.getInstance(hashAlgorithm);
        digest.reset();
        digest.update(ESAPI.securityConfiguration().getMasterSalt());
        digest.update(salt.getBytes(encoding));
        digest.update(plaintext.getBytes(encoding));

        // rehash a number of times to help strengthen weak passwords
        bytes = digest.digest();
        for (int i = 0; i < iterations; i++) {
            digest.reset();
            bytes = digest.digest(bytes);
        }

        String encoded = ESAPI.encoder().encodeForBase64(bytes, false);
        return encoded;
    } catch (Exception ex) {
        throw new EncryptionException("Internal error", "Error");
    }
}
(9b) Password Security Defenses

- Disable Browser Autocomplete
  - `<form AUTOCOMPLETE="off"/>`
  - `<input AUTOCOMPLETE="off"/>`

- Password and form fields
  - Input type=password

- Additional password security
  - Do not display passwords in HTML document
  - Only submit passwords over HTTPS
(10) Encryption in Transit (TLS)

- Authentication credentials and session identifiers must be encrypted in transit via HTTPS/SSL
  - Starting when the login form is rendered
  - Until logout is complete
  - All other sensitive data should be protected via HTTPS!

- https://www.ssllabs.com free online assessment of public facing server HTTPS configuration