CSE 484: Computer Security and Privacy

Web Security

Fall 2021

David Kohlbrenner
dkohlbre@cs

Thanks to Franzi Roesner, Dan Boneh, Dieter Gollmann, Dan Halperin, David Kohlbrenner, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Administrivia

• HW2 Due Today!

• Final Project deadline 1 next Friday
  • Take a look at the syllabus for more information
Cross-Site Scripting (XSS)

Victim’s browser

Access some web page

GET/steal.cgi?cookie=

Forces victim’s browser to call hello.cgi on naive.com with this script as “name”

GET/hello.cgi?name=


GET/ hello.cgi?name=


Hello, dear


Welcome!</HTML>

Interpreted as JavaScript by victim’s browser: opens window and calls steal.cgi on evil.com

evil.com

naive.com

hello.cgi
Basic Pattern for Reflected XSS

Injected script can manipulate website to show bogus information, leak sensitive data, cause user’s browser to attack other websites. This violates the “spirit” of the same origin policy.
Reflected XSS

• User is tricked into visiting an honest website
  • Phishing email, link in a banner ad

• Bug in website code causes it to echo to the user’s browser an arbitrary attack script
  • The origin of this script is now the website itself!

• Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  • This violates the “spirit” of the same origin policy
Stored XSS

1. Attack server
   - Inject malicious script
   - Store bad stuff

2. User victim
   - Request content

3. User victim
   - Receive malicious script

4. User victim
   - Users view or download content

   - Steal valuable data
Where Malicious Scripts Lurk

• User-created content
  • Social sites, blogs, forums, wikis

• When visitor loads the page, website displays the content and visitor’s browser executes the script
  • Many sites try to filter out scripts from user content, but this is difficult!
In all XSS there are 3 actors

• Adversary

• Server victim

• User victim
Canvas time
Preventing Cross-Site Scripting

• Any user input and client-side data must be preprocessed before it is used inside HTML

• Remove / encode HTML special characters
  • Use a good escaping library
    • OWASP ESAPI (Enterprise Security API)
    • Microsoft’s AntiXSS
  • In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    • ‘ becomes &\#039; “ becomes &quot; & becomes &amp;
  • In ASP.NET, Server.HtmlEncode(string)
Evading Ad Hoc XSS Filters

• Preventing injection of scripts into HTML is hard! → Use standard APIs
  • Blocking “<” and “>” is not enough
  • Event handlers, stylesheets, encoded inputs (%3C), etc.
  • phpBB allowed simple HTML tags like <b>
    <b c="">" onmouseover="script" x="&lt;b &gt;Hello&lt;b&gt;
• Beware of filter evasion tricks (XSS Cheat Sheet)
  • If filter allows quoting (of <script>, etc.), beware of malformed quoting:
    <IMG """"><SCRIPT>alert("XSS")</SCRIPT>"
  • Long UTF-8 encoding
  • Scripts are not only in <script>:
    <iframe src='https://bank.com/login’ onload=‘steal()’>
MySpace Worm (1)

• Users can post HTML on their MySpace pages
• MySpace does not allow scripts in users’ HTML
  • No <script>, <body>, onclick, <a href=javascript://>
• ... but does allow <div> tags for CSS.
  • <div style="background:url('javascript:alert(1)')">
• But MySpace will strip out “javascript”
  • Use “java<NEWLINE>script” instead
• But MySpace will strip out quotes
  • Convert from decimal instead:
    alert('double quote: ' + String.fromCharCode(34))
MySpace Worm (2)

Resulting code:

```html
< div id="mycode" style="BACKGROUND: url(/images/1200x0.png);" >
<script>
 eval(document.all.mycode.expr) 
 </script>
</div>
```

11/4/2011  
CSE 484 - Fall 2021
MySpace Worm (3)

• “There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or [make anyone angry]. This was in the interest of.. interest. It was interesting and fun!”

• Started on “samy” MySpace page

• Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero

• 5 hours later “samy” has 1,005,831 friends
  • Was adding 1,000 friends per second at its peak
Twitter Worm (2009)

• Can save URL-encoded data into Twitter profile
• Data **not** escaped when profile is displayed
• Result: StalkDaily XSS exploit
  • If view an infected profile, script infects your own profile

```javascript
var update = urlencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter but with pictures, videos, and so much more! ");

var ajaxConn = new XHConn();
ajaxConn.connect("/status/update", "POST", "authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST", "authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update")
```

SQL Injection
Typical Login Prompt
Typical Query Generation Code

```php
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key " . "WHERE Username='".$selecteduser."';
$rs = $db->executeQuery($sql);
```

What if ‘user’ is a malicious string that changes the meaning of the query?
User Input Becomes Part of Query

```sql
SELECT passwd
FROM USERS
WHERE uname IS 'user'
```
Normal Login

Web browser (Client) → Web server

Enter Username & Password

Web server → DB

SELECT passwd FROM USERS WHERE uname IS 'alicebob'
Malicious User Input
SQL Injection Attack

```
SELECT passwd
FROM USERS
WHERE uname IS '';
DROP TABLE USERS; -- '
```

Eliminates all user accounts
Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something? In a way -

Did you really name your son Robert’; DROP TABLE Students; -- ?

Oh, yes. Little Bobby Tables, we call him.

Well, we've lost this year's student records. I hope you're happy.

And I hope you've learned to sanitize your database inputs.

http://xkcd.com/327/
SQL Injection: Basic Idea

- This is an **input validation vulnerability**
  - Unsanitized user input in SQL query to back-end database changes the meaning of query
- Special case of command injection

1. Attacker posts malicious form
2. Unintended query received
3. Data from DB
Authentication with Backend DB

set UserFound = execute(
    "SELECT * FROM UserTable WHERE
    username=' ' & form("user") & ' ' AND
    password=' ' & form("pwd") & ' '");

User supplies username and password, this SQL query checks if user/password combination is in the database

If not UserFound.EOF
    Authentication correct
else Fail

Only true if the result of SQL query is not empty, i.e., user/pwd is in the database

(*) remember to hash passwords for real authentication scheme
Using SQL Injection to Log In

- User gives username ‘ OR 1=1 --
- Web server executes query
  
  ```
  set UserFound=execute(
      SELECT * FROM UserTable WHERE
      username=‘’ OR 1=1 -- ... );
  ```

- Now all records match the query, so the result is not empty ⇒ correct “authentication”!

Always true!  Everything after -- is ignored!

• SQL injection attack where attacker asks database series of true or false questions

• Used when
  • the database does not output data to the web page
  • the web shows generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

• SQL Injection vulnerability more difficult to exploit, but not impossible.
Preventing SQL Injection

• Validate all inputs
  • Filter out any character that has special meaning
    • Apostrophes, semicolons, percent, hyphens, underscores, ...
    • Use escape characters to prevent special characters from becoming part of the query code
      • E.g.: escape(O’Connor) = O\’Connor

• Check the data type (e.g., input must be an integer)

• Same issue as with XSS: is there anything accidentally not checked / escaped?
Prepared Statements

```java
PreparedStatement ps =
    db.prepareStatement("SELECT pizza, toppings, quantity, order_day "+ "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```

- **Bind variables**: placeholders guaranteed to be data (not code)
- Query is parsed without data parameters
- Bind variables are typed (int, string, ...)

Wait, why not do that for XSS?

- “Prepared statements for HTML”?
Data-as-code

• XSS

• SQL Injection

• (Like buffer overflows)
Cross-Site Request Forgery (CSRF/XSRF)