CSE 484 / CSE M 584: Computer Security and Privacy

Web Security
[Web Application Security]

Spring 2020

Franziska (Franzi) Roesner
franzi@cs.washington.edu

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

- HW 2 due today
- Lab 2 out
  - Highly recommend the readings on the course schedule
- Late Days +2
  - This quarter is tough
  - Max of 3 per assignment
  - No late days for last final project deadline
XSS Recap

Fundamental issue: data interpreted as code.
Violates the spirit of the same-origin policy
(code is not really from the same origin).

What if ”term” is `<script>alert(document.cookie);</script>` ?

```html
<html> <title>Search results</title> <body>You have searched for `<?php echo $_GET['term'] ?>`... </body>
</html>
```
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 XSS Filters

• Preventing injection of scripts into HTML is hard!
  – Blocking “<” and “>” is not enough
  – Event handlers, stylesheets, encoded inputs (%3C), etc.
  – phpBB allowed simple HTML tags like <b>
    \[ <b c="">" onmouseover="script" x="<b ">Hello<b> \]

• 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
<doc>

/* Autogenerated */
https://samy.pl/myspace/tech.html

MySpace Worm (2)

Resulting code:

```
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 piss anyone off. 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
SQL Injection
Typical Login Prompt

![Login Prompt Image]
$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

- Web browser (Client)
  - Enter Username & Password
  - SELECT passwd
    FROM USERS
    WHERE uname IS ‘$user’

- Web server

- DB
Normal Login

Web browser (Client) → Enter Username & Password → Web server → SELECT passwd FROM USERS WHERE uname IS 'franzi' → DB
Malicious User Input
SQL Injection Attack

Enter Username & Password

Web browser (Client)

Web server

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

Eliminates all user accounts

DB
Exploits of a Mom

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 post malicious form
2. Victim server unintended query
3. Victim SQL DB receive 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
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!
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)
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, ...)

http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html
OWASP Top 10 Web Vulnerabilities

1. Injection
2. Broken Authentication & Session Management
3. Cross-Site Scripting
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery
9. Using Known Vulnerable Components
10. Unvalidated Redirects and Forwards
Cross-Site Request Forgery (CSRF/XSRF)
Cookie-Based Authentication Redux

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

response
Browser Sandbox Redux

• Based on the same origin policy (SOP)
• Active content (scripts) can send anywhere!
  – For example, can submit a POST request
  – Some ports inaccessible -- e.g., SMTP (email)
• Can only read response from the same origin
  – … but you can do a lot with just sending!
Cross-Site Request Forgery

• Users logs into bank.com, forgets to sign off
  – Session cookie remains in browser state
• User then visits a malicious website containing
  
  ```html
  <form name=BillPayForm
    action=http://bank.com/BillPay.php>
    <input name=recipient value=badguy> …
  </form>
  ```

  ```javascript
  document.BillPayForm.submit();
  ```

  • Browser sends cookie, payment request fulfilled!
  • **Lesson**: cookie authentication is not sufficient when side effects can happen
Cookies in Forged Requests

User credentials automatically sent by browser
Impact

• Hijack any ongoing session (if no protection)
  – Netflix: change account settings, Gmail: steal contacts, Amazon: one-click purchase

• Reprogram the user’s home router

• Login to the attacker’s account
Hidden iframes submitted forms that...
- Changed user’s email notification settings
- Linked a new checking account
- Transferred out $5,000
- Unlinked the account
- Restored email notifications