CSE 484: Computer Security and Privacy

Web Security

Spring 2021

Tadayoshi Kohno
yoshi@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

• Last Class: Ariana Mirian from UCSD “Hack for Hire: Exploring the Emerging Market for Account Hijacking”

• Upcoming guest lectures (please join during class time, if possible, for Q&A)
  • Friday, May 14: Emily McReynolds from Microsoft re: security, privacy, and the law
  • Monday, May 17: Sunny Consolvo and Kurt Thomas from Google re: recent work on “Hate, Harassment, and the Changing Landscape of Online Abuse”
  • Friday, May 28: Charlie Reis from Google on Chrome Security
Administrivia

• HW2 due on May 14
  • Please see rubric on canvas re: where we want to see work
• Final Project deadlines coming soon
  • May 14: Project formation + brief description
  • May 26: Outline and references
  • June 7: Final submission
Possible Talk of Interest (Langdon Winner, May 20, 5:30pm)

• Decades of enthusiasm for the magic of digital devices has generated a society largely passive as regards democratic participation in the shaping of new technologies that will affect how we live.

• We’ve learned to accept and celebrate whatever flows from the Silicon Valley pipeline, even when the results undermine personal privacy and concentrate wealth and power in the hands of a scant few.

• Initiatives in “technology assessment” from earlier times encouraged popular participation and careful reflection upon choices in this realm. Can this approach be revived?

• RSVP: https://techpolicylab.uw.edu/events/event/distinguished-lecture-with-langdon-winner-technology-innovation-and-the-malaise-of-democracy/
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) → Web server

Enter Username & Password

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 login page with malicious input]

The image shows a login page with a malicious input entered into the username field. The input is `'; DROP TABLE USERS; --`. This is an example of SQL injection, where a malicious user attempts to execute a SQL statement through the application's input validation vulnerabilities.
SQL Injection Attack

Web browser (Client) → Enter Username & Password → Web server → DB

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

Eliminates all user accounts
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. Attack post malicious form
2. Unintended query
3. Receive data from DB
Authentication with Backend DB

```sql
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!
“Blind SQL Injection”  

- 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

Browser

POST/login.cgi

Set-cookie: authenticator

GET... Cookie: authenticator

response

Server
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

www.attacker.com

GET /blog HTTP/1.1

<form action=https://www.bank.com/transfer method=POST target=invisibleframe>
<input name=recipient value=attacker>
<input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>

POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
Recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E

HTTP/1.1 200 OK
Transfer complete!

www.bank.com
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

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)
XSRF Defenses

• Secret validation token

  <input type=hidden value=23a3af01b>

• Referer validation

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

Referer:
- ?

For your security, never enter your Facebook password on sites not located on Facebook.com.
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