CSE 484 / CSE M 584: Computer Security and Privacy

Web Security
[Web Application Security]

Autumn 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

• Lab 2
  – Granting access starting today
  – Please sign up if you haven’t already

• Final project
  – First checkpoint deadline Friday

• This week
  – No class or office hours Wednesday (Veterans’ Day)
  – Guest lecture Friday (Charlie Reis, Google, web security)
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

SELECT passwd FROM USERS WHERE uname IS ‘$user’

Enter
Username
& Password

Web
server

Web
browser
(Client)
Normal Login

Web browser (Client) → Enter Username & Password

Web server

SELECT passwd FROM USERS WHERE uname IS ‘franzi’

DB
Malicious User Input

![Image of a login form with SQL injection]

The login form contains a field for the user name and password. The user name field has been modified to include a SQL injection attack, attempting to drop the `USER` table.

```
Enter User Name: 'DROP TABLE USERS;' --
Enter Password: ********
```
SQL Injection Attack

Enter Username & Password

Web server

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

DB

Web browser (Client)

Eliminates all user accounts
Exploits of a Mom

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** post malicious form
2. **Victim SQL DB** send unintended query
3. **Victim server** 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

• Web server executes query

```sql
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, …)

Cross-Site Request Forgery (CSRF/XSRF)
Cookie-Based Authentication Redux

Browser

POST/login.cgi

Server

Set-cookie: authenticator

GET...
Cookie: authenticator

response

not forgeable; $secret
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
  <form name=BillPayForm
    action=http://bank.com/BillPay.php>
    <input name=recipient value=badguy> ...
  </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

User credentials sent in forged request to www.bank.com

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!
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? spy on you

[XSRF on login form]
XSRF True Story

[Alex Stamos]

Internet Exploder

GET news.html

HTML and JS

HTML Form POSTs

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

GET news.html

www.cybervillians.com/news.html

Bernanke Really an Alien?

script

ticker.stockbroker.com

Java
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
Referer Validation

- **Lenient** referer checking – header is optional
- **Strict** referer checking – header is required

Referer:
- Referer:
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