CSE 484 / CSE M 584: Computer Security and Privacy

Web Security
[Web Application Security]

Spring 2019

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

• Homework 2 due this Friday 5pm
  – Do the encrypted email part now!

• Lab 2 out now
  – Due Friday, May 24, 5pm
  – More today and in section and next Monday
  – Please see readings on course schedule!

• Final Project Checkpoint #1 due next Friday

• My office hours this week: Th 9:30-10:30am
Dynamic Web Application

GET / HTTP/1.1
HTTP/1.1 200 OK
index.php

Web server

Database server
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 Scripting (XSS)
PHP: Hypertext Processor

- Server scripting language with C-like syntax
- Can intermingle static HTML and code
  
  ```
  <input value=<?php echo $myvalue; ?>>
  ```

- Can embed variables in double-quote strings
  ```
  $user = "world"; echo "Hello $user!";
  or $user = "world"; echo "Hello" . $user . "!";
  ```

- Form data in global arrays $$_GET, $$_POST, ...
Echoing / “Reflecting” User Input

Classic mistake in server-side applications


search.php responds with
<html> <title>Search results</title> <body>You have searched for <?php echo $_GET[term] ?>... </body>
Echoing / “Reflecting” User Input

\[ \text{naive.com/hello.php?name} = \text{Bob} \]

Welcome, dear Bob

Welcome, dear
XSS – Quick Demo

```php
<?php
setcookie("SECRET_COOKIE", "12345");
header("X-XSS-Protection: 0");
?>
<html><body><br><br>
<form action="vulnerable.php" method="get">
Name: <input type="text" name="name" size="80">
<input type="submit" value="submit"></form>
<br><br><br>
<div id="greeting">
<?php
$name = $_GET['name'];
if($name) { echo "Welcome ". $_GET['name'];}
?>
</div></body></html>
```

Need to explicitly disable XSS protection – newer browsers try to help web developers avoid these vulnerabilities!
Cross-Site Scripting (XSS)

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

GET/hello.cgi?name=


Hello, dear
Welcome!"</HTML>

naive.com

evil.com

victim’s browser
Reflected XSS

• User is tricked into visiting an honest website
  – Phishing email, link in a banner ad, comment in a blog
• Bug in website code causes it to echo to the user’s browser an arbitrary attack script
  – The origin of this script is now the website itself!
• Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  – This violates the “spirit” of the same origin policy
Basic Pattern for Reflected XSS

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. send valuable data
Where Malicious Scripts Lurk

• User-created content
  – Social sites, blogs, forums, wikis

• When visitor loads the page, website displays the content and visitor’s browser executes the script
  – Many sites try to filter out scripts from user content, but this is difficult!
Stored XSS

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

Users view or download content

Attack server

Server victim

Store bad stuff
Twitter Worm (2009)

- Can save URL-encoded data into Twitter profile
- Data not escaped when profile is displayed
- Result: StalkDaily XSS exploit
  - If view an infected profile, script infects your own profile

```javascript
var update = urlencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter but with pictures, videos, and so much more! ");
var ajaxConn = new XHConn();
ajaxConn.connect("/status/update", "POST", "authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST", "authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update")

```
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.
• 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()'>`
    `<b c="">" onmouseover="script" x="<b >Hello<b>`
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:

```
<http://namb.la/popular/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

![Typical Login Prompt Image]
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’

Database (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

Web browser (Client) → Web server

Enter Username & Password →

Web server → DB

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

Eliminates all user accounts
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
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

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