Web Vulernabilities

Daniel Zappala

CS 360 Internet Programming
Brigham Young University

Web Vulnerabilities

- once you put up a site, you *will* be attacked
- web applications make you vulnerable
  - database = opportunity to steal, modify, or add information
    - place an order in your Amazon account
    - add comment/link spam to a web site
    - delete your email
  - Javascript = opportuntiy to run a program on user's machine
    - ▸ redirect you to a site
    - ▸ trick user into entering password
    - steal cookies or login credentials
    - change DNS entry to impersonate your bank
- two common attacks: XSS, CSRF

XSS

XSS

- Cross-Site Scripting (XSS) attack
  - attacker injects client-side script into a web page viewed by someone else
  - relies on browser trusting the scripts given to it by the current web site
- *if I visit Facebook, I should be safe to execute scripts the Facebook site gives me*

Example Vulnerability

```
http://www.google.com/search?q=flowers

<p>Your search for 'flowers'
returned the following results:</p>
```

- if server does not check the input, then an attacker can inject a script

Example Vulnerability

```
http://www.google.com/search?q=flowers+<script>alert(1)
</script>

<p>Your search for 'flowers<script>alert(1)</script>'
returned the following results:</p>
```

- if you can execute a script, then you can
  - redirect to malware
  - deface a web site
  - steal cookies, passwords, clipboard

XSS Statistics

- WhiteHat Web Site Security Statistics Report, 2010
  - 64% of web sites vulnerable to XSS attack
  - 105 days on average to fix it (banking is faster, retail is slower)
- why aren't they fixed?
  - no one at organization understands them or is responsible for fixing them
  - features prioritized ahead of security
  - code owned by an unresponsive third party
  - risk is accepted

XSS Types

- reflected
  - user input read from request parameters in URL and written directly to output
  - attack usually delivered via email or a neutral web site
  - get user to click on URL
- persistent
  - script stored directly on a web site (e.g. a Facebook status or Flickr caption)
  - when victim visits the web page, viewing the page triggers the attack

XSS Vulnerability, Django

```
c = Comment()
c.text = request.POST[’text’]
c.save()
```

- site accepts comments, stores input directly from user
- when comment is displayed, it can include anything, including script

Example

- ▸ list-o-matic
- load the page xss.html in a browser
- use Firefox, compare with Chrome
- works because *templates/index.html* considers user input "safe"

Protection from XSS

- filter input
- escape output
- many web development frameworks do this for you automatically

CSRF

CSRF

- Cross-Site Request Forgery (CSRF) attack
  - attacker tricks victim into executing a script on a site where the victim has an account
  - relies on server trusting the user's identity
- *if the user logs in to my bank and sends me a request to withdraw funds that contains his login cookie, then I can trust that it is really her*

Example Vulnerability

```
<html>
<body>
<p>Welcome!</p>
<img src=http://bank.example.com/transfer?fromaccount=bob&
     amount=1000000&toaccount=mallory">
</html>
```

- if you are currently logged into your bank, then the bank cannot tell that this request isn't coming from you

CSRF Statistics

- WhiteHat Web Site Security Statistics Report, 2010
  - 24% of web sites vulnerable to CSRF attack
  - hard to capture because web site logs make it look like a legitimate user request, may be under-reported
- identified on ING Direct (banking), YouTube, MetaFilter, The NY Times in 2008

Example

- ▸ list-o-matic
- load the page csrf.html in a browser
- use Firefox, compare with Chrome
- works because server uses only the cookie to validate the user's identity

Protection from CSRF

- tokens
  - require a GET request to get a form before accepting a POST request for the form
  - send a token in the GET request that must be echoed back in the POST
  - token should be random and unique to that form
  - expire the token after a short time
- require user authorization for significant transactions

JSON Web Tokens

JSON Web Tokens

1. client sends login request with username and password

```
POST /api/users/login HTTP/1.1
Host: listomatic.com

username: ‘‘emma’’
password: ‘‘emma’’
```

2. server validates username and password, responds with crytographic token

```
200 OK

name: ‘‘Emma’’
token: ‘‘eyJ0e...’’
```

3. client sends token in all subsequent requests

```
GET /api/items
Host: listomatic.com
Authorization:‘‘eyJO...’’
```

Token Format

```
Header.Payload.Signature
```

- Header: { "alg": "HS256", "typ": "JWT" }
- Payload: { username: username }
- Signature: HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
- all are Base64 encoded strings
- ▸ the JSON Web Token library includes additional options such as token expiration

Advantages

- no server state
    - if token decrypts properly, using server secret, then it contains the needed state (e.g. username of user)
- compact – easy to store in cookie or HTML 5 local storage
- every request may be authenticated

Storing the Token

- cookie
  - use the HttpOnly flag
  - prevents XSS since not accessible to JavaScript
  - vulnerable to CSRF, so use CSRF protection
- HTML 5 storage
  - prevents CSRF
  - vulnerable to XSS since any JavaScript you serve can access the token
  - all the libraries you depend on must be secure
- some strong opinions: ▸ Where to store your JWTs