Web Vulnerabilities/Google Gruyere Debrief
Same-Origin Policy

• Key tenet of the web security model
• Means that a website can only access data related to its current domain
• Any cookies set on one domain cannot be accessed or set on another domain
• For example google.com has no idea what your cookies on facebook.com are, and they can’t change them
• The first 3 types of attacks we’ll see are trying to get around this/use it to their advantage
Cross-Site Scripting (XSS) Attack

- Injecting client-side scripts that users will run on trusted website
- Sites that display user-generated content (search fields, comments, file uploads, etc.) vulnerable
- Gets around the “same-origin policy” by running malicious code in the trusted site
- Non-Persistent/Reflected attacks are more common
  - Require users to visit a specific URL
  - This page will reflect un-sanitized parts of URL
  - Example: https://www.google.ca/?q=<script type="text/javascript">alert(“attack!”)</script> (obviously they sanitize this)
- Persistent attacks can be more devastating
  - Permanently display injection code on normal pages
  - Example: Make a comment on Facebook “<script type="text/javascript">alert(“attack!”)</script>” (again, they sanitize)
Cross-Site Scripting (XSS) Solutions

• Use a separate domain for user-generated content (to prevent access to secure cookies)
• Sanitize/Escape any user-provided text before rendering it
• Ideally a template engine will handle the sanitization/escaping process for web developers because it can be a hard problem
• For example Django templates will automatically escape all content unless they are explicitly told not to
• See https://code.djangoproject.com/wiki/AutoEscaping for why this choice was made (and links to arguments surrounding it)
Cross Site Request Forgery (XSRF/CSRF) Attack

• Unlike XSS, these are run on malicious website
• Malicious website has the user make a malicious request to a legitimate/trusted website
• Takes advantage of the fact that the request is actually coming from the compromised user
• Often identity is verified using a cookie
• User will send all cookies for that domain alongside that request
• Embed malicious request in some element on malicious website
• Example: `<img src="http://trusted_bank_domain.com/withdraw?amount=10000&for=bad_guy">`
Cross-Site Script Inclusion (XSSI) Attack

• Similar to CSRF, but specific to Javascript
• Include a Javascript script from a trusted website on your malicious website
• User will request legitimate script from legitimate website
• Malicious site can then use data from this legitimate script for malicious purposes
• Example: `<script src="http://trusted_bank_domain.com/my_account_details.js"></script>`
CSRF and XSSI Solutions

• Make sure all requests that change state are POSTs (sometimes still exploitable, but makes it harder)
• Generate and use CSRF tokens
  – Govind mentioned this on Friday
  – Many web frameworks automatically do a CSRF check on all POSTs (so you need to include this CSRF token)
  – Django for example, you must include `{% csrf_token %}` on any POSTed form
  – See https://docs.djangoproject.com/en/dev/ref/csrf/ for more details
  – Note that this works because of same origin policy with cookies
Path Traversal Attack/Solution

Attack:
• Web browsers traverse files just like a terminal
• Meaning ../ is a valid path
• If a server’s directories are not secured properly, then users can see files they shouldn’t

Solution:
• Secure directories and files (using chmod)
• Ensure that web server will not serve files outside of the root web directory
Buffer Overflow/Integer Overflow

- Huge topic, not specific to just websites
- Buffer Overflow: Allow user to write data past the end of a buffer’s allocated size
- Integer Overflow: Allow integer computation to result in an integer larger than that integer type can handle
- From attackers standpoint this is beneficial because it can lead to arbitrary server-side code execution
- **Solution**: Unique to each situation, but in general be diligent in bounds- and error-checking
Remote Code Execution

- Refers to an attacker executing remote code on server
- Often performed using buffer overflows
- Again not specific to websites, but often more vulnerable because they are actively listening for incoming connections
- **Solution**: Protect against buffer overflows, never ever run web server (or any network-accessible applications) as root user, and keep system updated with security patches
SQL Injection Attack

• Basic SQL query: SELECT username, password FROM Users WHERE username="Alice"
• If this statement were executed where “Alice” is directly taken from user input, user confidentiality would be at risk
• Other statements modify the database, meaning if we use unsanitized user input the whole database can be destroyed/changed
• Example: http://xkcd.com/327/
SQL Injection Solutions

• Parse user provided data to ensure it is escaped/sanitized, and in the correct format

• Conflicting opinions on how this parsing should be done (as demonstrated in Friday’s guest lecture)

• Use an object-relational mapping (ORM) which will generally sanitize input for you (and may make databases easier to work with!)

• ORM Example (Django): https://docs.djangoproject.com/en/1.7/topics/db/queries/
AJAX (Asynchronous JavaScript and XML)

- Not about attacks specifically, just quick summary of AJAX since Gruyere uses it
- Basically means you’re communicating with the server without a whole new page load
- Google popularized AJAX by using it with Gmail and Google Maps
- Note that in Gmail the whole page never reloads, this is AJAX
- Requests are made and responses processed through Javascript, so could potentially introduce more vulnerabilities