Web Attacks III

Attacking the Client and the Server

SQL Injection: Defense

- Recall SQL injection:
  - `SELECT * FROM users WHERE username="$username" AND password="$password"`
  - Submit username `bob`, password `" OR 1=1; --`
  - Becomes `SELECT * FROM users WHERE username="bob" AND password="" OR 1=1; --"`
- SQL sanitization is hard
  - ...really hard
  - PHP's `mysql_escape_string` and `mysql_real_escape_string`

SQL Injection: Defense

- The answer: *prepared statements*
- Insight: the root of the problem is treating data as code
- Don’t embed data in code; send code and data separately
- Send query string and parameters separately
  - First send `SELECT * FROM users WHERE username=? AND password=?`
  - Then send list of typed parameters:
    - string: `bob`
    - string: `' OR 1=1; --`

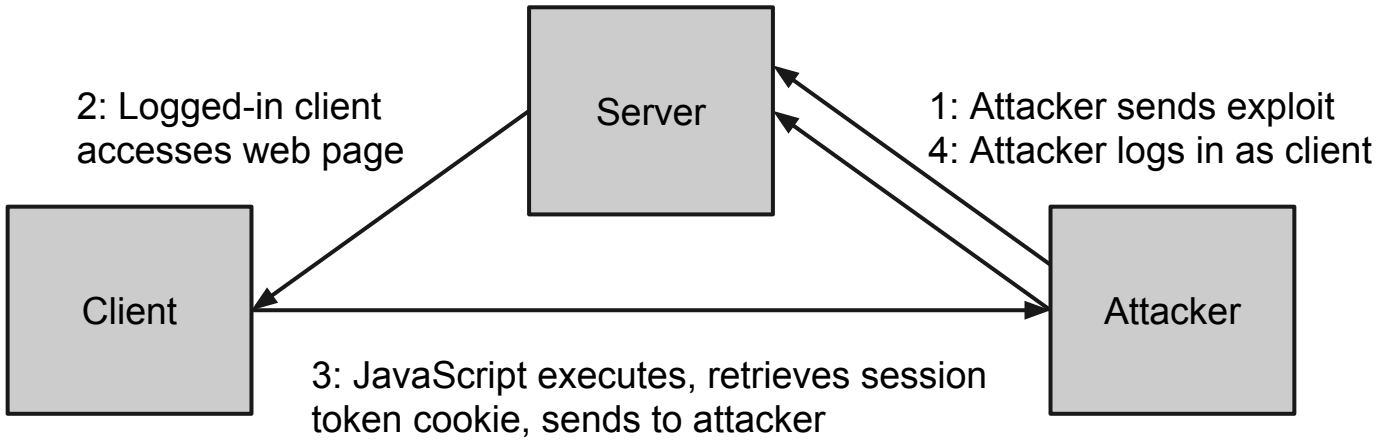
Attacking the Client

Cross-Site Scripting (XSS)

- Problem: users can submit text that will be displayed on webpages
  - Facebook posts
  - Blog comments
  - In-browser chat
  - In-browser email
  - etc
- Browsers interpret everything in HTML pages as HTML
- What could go wrong?

Cross-Site Scripting (XSS)

- Take-home: you can do anything the browser can do
- Classic example: steal session token



Cross-Site Scripting (XSS)

- Defense is *really hard*
- Attempt #1: remove `<script>` tags
  - `<scr<script>ipt>`
- Attempt #2: HTML-encode special characters
  - `<` becomes `&lt;`, `>` becomes `&gt;`, etc
  - In attributes, browsers first HTML-decode
  - User submits a link: `javascript:evilFunc()`
  - `<a href=”javascript&#58;evilFunc()”>`
- “it is preferable to avoid inserting user-controllable data into these locations” (The Web Application Hacker’s Handbook)

Cross-Site Scripting (XSS)

- “it is preferable to avoid inserting user-controllable data into these locations” (The Web Application Hacker’s Handbook)
  - In other words, don’t try this at home
- Luckily, libraries exist to do this sanitization for you
- Use them!
- Many major tech companies have people whose job it is to maintain internal versions of these libraries specific to the product

Cross-Site Scripting (XSS)

- Let's say you've fixed all these "stored XSS" issues
- Still have to worry about "reflected XSS"
- Let's say you have a 404 page:

```
<?php echo "page not found: " . $_SERVER
['REQUEST_URI']; ?>
```

- `foo.com/<script>evilFunc()</script>`
- So a user can attack themselves… so what?

Cross-Site Scripting (XSS)

- So a user can attack themselves… so what?
  - How hard is it to get people to click links?
    - hint: come to lecture on Friday
- So make sure to sanitize *all* data on the backend
- But what about on the front-end?
- Pages could dynamically use the URL from JavaScript

```
<script> url = document.location; </script>
```

Cross-Site Scripting (XSS)

https://www.youtube.com/watch?v=zv0kZKC6GAM

image source: http://i.telegraph.co.uk/multimedia/archive/02939/tweetdeck_2939196b.jpg

Cross-Site Request Forgery (CSRF)

- Problem:
  - any requests to a domain include all cookies
  - sites you visit can have javascript which initiate arbitrary requests
- What could go wrong?

Cross-Site Request Forgery (CSRF)

- Problem:
  - any requests to a domain include all cookies
  - sites you visit can have javascript which initiate arbitrary requests
- What could go wrong?
  - You can request pages with the permissions of the user
  - Access sensitive data
  - Take privileged actions

Cross-Site Request Forgery (CSRF)

- In more detail:
  - The client visits your (malicious) website, `foo.com`
  - `foo.com` contains JavaScript:

```
var req = new XMLHttpRequest();
req.open("GET", "https://bank.com/transfer?to=
<attacker-account>&amount=10000000000000000");
```

- Profit!

Cross-Site Request Forgery (CSRF)

- In more detail:
  - The client visits your (malicious) website, `foo.com`
  - `foo.com` contains JavaScript:

```
var req = new XMLHttpRequest();
req.open("GET", "https://bank.com/transfer?to=
<attacker-account>&amount=10000000000000000");
```

- Profit!
- So how could we prevent this?

Cross-Site Request Forgery (CSRF)

- CSRF tokens
  - Every request includes not just cookies, but extra data (called “tokens”) which is loaded with the page and not stored by the browser
  - In order to know token, must be running JavaScript *on the page*

Cross-Site Request Forgery (CSRF)

- Is this mitigation perfect?

Cross-Site Request Forgery (CSRF)

- Is this mitigation perfect?
  - No!
  - It assumes JavaScript coming from the server is trusted
  - Not true if there's an XSS vulnerability
  - This is how the self-reteweeting tweet worked
    - Would not have worked if a different site had tried to request Tweetdeck's retweet URL (assuming they had proper CSRF protection)

Attacking the Server

Basic Dynamic Execution

- Server has a *web root*
  - ie, `/var/www`
- URL paths are interpreted relative to web root
  - ie, `foo.com/bar` refers to `/var/www/bar`
- Web server determines whether it's static or dynamic
  - static: html, css, javascript - served directly
  - dynamic: code to be executed (ie, PHP) - executed
    - output of the execution is sent as the response
- If the path refers to a directory, serve listing of contents or default page (like `index.html` or `index.php`)

Improper Path Sanitization

- Problem: only some paths are valid; which ones?
- Improper path sanitization can lead to disallowed resources being accessed
- What sorts of resources/paths might we want to make off-limits?

Improper Path Sanitization

- Problem: only some paths are valid; which ones?
- Improper path sanitization can lead to disallowed resources being accessed
- What sorts of resources/paths might we want to make off-limits?
  - configuration files (ie, Apache's `.htaccess`)
  - files outside the web root

Improper Path Sanitization

- Attempt #1: path blacklists
  - ie, "`/foo/bar` is off limits"
- What's wrong with this?

Improper Path Sanitization

- Attempt: path blacklists
  - ie, "`/foo/bar` is off limits"
- What's wrong with this?
  - Multiple paths can refer to the same resource
  - `/foo/bar/`
  - `/foo//bar`
  - `/foo/../foo/bar`
  - `/foo/bar/baz/..`

Improper Path Sanitization

- Attempt: path blacklists
  - ie, "`/foo/bar` is off limits"
- What's wrong with this?
  - What about paths outside the web root?
  - `/../../etc/passwd`
    - Becomes `/var/www/../../etc/passwd`
    - (ie, `/etc/passwd`)
  - (if you don't know where the web root is, use `/../../../../../../../../../../etc/passwd`)

OS Code Injection

- Problem: web site code may invoke OS commands to perform work
  - ie, PHP's `exec` function
- You have to trust *all* OS commands you invoke!
- What could go wrong?

OS Code Injection

- Some commands are more powerful than you'd think
  - ie, `less` allows arbitrary shell commands
    - seriously: run `less` and then type `!<shell command>`
- You have to properly sanitize *all* inputs
  - ...which is *really hard*
  - Example:
    - Users have personal directories named with their usernames
    - Users are allowed to run `ls` on their directories (ie, `ls <username>`)
    - I choose my username to be "-R"
    - When I ask for a listing, server runs `ls -R`, and I get a recursive listing of all files in all users' directories

OS Code Injection

- Also have to worry about the invocation itself
- Since arguments are really a list of strings, many languages take a list of string arguments
  - ie, `exec(command, arg1, arg2, ...)`
- ...but not all languages ;)
- PHP's exec takes one argument and executes it using `/bin/sh`
- This means that you can escape the invocation and talk to `/bin/sh` directly

OS Code Injection

- PHP's exec takes one argument and executes it using `/bin/sh`
- This means that you can escape the invocation and talk to `/bin/sh` directly
  - ie, `exec("echo " . $input)`
  - I supply `"; <literally-anything>"`
  - Becomes `echo; <literally-anything>`
  - `"; rm -rf /"`
  - `"; sleep 10000000000000"`
  - `"; useradd hacker"`

Execution Engine Attacks

- Problem:
  - Web server executes anything it thinks is code
  - Users can affect what files are on the filesystem
- TL;DR: Get the server to execute your code
- How?

Execution Engine Attacks

- If paths are not properly sanitized, could execute scripts outside of the web root
  - ie, `foo.com/../../root/delete-users.php`
- If uploaded files are stored in the web root, could upload script and execute it
  - Let's say uploads are stored in `/uploads`
  - Upload `myscript.php`
  - Request `foo.com/uploads/myscript.php`
- What's the vulnerability? What allows this to happen?

Execution Engine Attacks

- Problem: any `.php` files are executed
- Solutions?

Execution Engine Attacks

- Problem: any `.php` files are executed
- Solutions?
  - Don’t let users upload `.php` files
    - Better hope your sanitization is good!

Execution Engine Attacks

- It gets worse
- Sometimes PHP is embedded in HTML:

```
<title><?php echo $SITE_TITLE; ?></title>
```

- So servers don't look for `.php` extension, but rather for `<?php` and `?>` tags.
- What can we do about this?

Execution Engine Attacks

- What can we do about this?
  - Attempt #1: has to be well-formed file
    - ie, if uploading a .jpg, has to be valid .jpg file
    - but .jpg file type (and many others) allow arbitrary embedded comments!
    - even if it didn't, could still craft sequence of pixels whose bytes were `\x3C\x3F\x70\x68\x70` ("`<?php`") and `\x3F\x3E` ("`?>`")

Execution Engine Attacks

- Solution: don't allow direct file access
  - Instead of `foo.com/uploads/foo.pdf`, `foo.com/get.php?file=foo.pdf`
  - Preferably store uploaded files *outside of the web root*
  - But be careful! `get.php` could introduce its own path traversal vulnerabilities
    - `foo.com/get.php?file=/../../../../../etc/passwd`