Security

Fall 2018 – Introduction to Full-Stack Development

Instructor: Matthew Fritter

matt@frit.me

Stepping Back: Lab 3

- In Lab 3, we focused on several additional modules within the Laravel framework:
  - Sessions
  - QueryBuilder
  - Validation
- We implemented all three of these modules as part of the SimpleChat website
  - Storing username and color data as a session variable, so it persists across pages
  - Querying our database to produce some basic API calls
  - Validating user input when creating a new chat room

SimpleChat: Storing Data in Session

- Recall that Laravel provides us with a special configurable session driver that allows us to store data in sessions
- Each session identifies a unique user across pages, allowing us to store data for an individual user between HTTP requests
- We used this session to save a user's username and color, instead of assigning them a new username and color every time they accessed a chatroom

SimpleChat: Querying the Database

- We used the QueryBuilder to access our database and dump out data for our API:
  - A basic query that dumps the entirety of the chat table
  - A search query that allows us to look for a user (or any other term) using a where 'LIKE' query
  - A query that allows us to dump the currently created chatrooms, without including their unique hash
- We dumped this data out in JSON format, which is pretty standard these days for passing API data

SimpleChat: Form Validation

- Finally, we added validation to our create() function, ensuring a couple things about our user input for our new chatroom name:
  - It is required
  - It has a max length of 127 characters
  - It is unique within the chatroom table
- By requiring that the name be unique within the chatroom table, we actually do away with the need for a unique hash to identify chatrooms, as there would be no potential for overlap between chatroom names
  - Can you think of a reason that it might be good to keep the unique hash though?

Questions

- Are there any remaining questions about Lab 3?
- Reminder: Lab 3 deadline is not extended – still due tomorrow night
- A note about marking: I'm aiming to have your quizzes back to you next class, in addition to your Lab 1 through 3 marks
- So far, initial looks at the quiz suggest the class did fairly well

Security – An Overview

Full Stack Development and Security

- As full-stack developers, we have to think about security at multiple different levels of the system:
  - Security of our back-end programming
  - Security of our database
  - Security of our server itself
- We are *not* security professionals. Entire fields of research, development, and practice exist to cover this: penetration testing, encryption, cryptanalysis, and security consulting
  - However, that doesn't mean that we should disregard security in our web applications

☆ Basic Concepts: The Bad Actor

- When we talk about "Bad Actors", we are talking about attackers – people, programs, or systems that seek to circumvent our security
- Bad Actors can have various motivations and fall within a variety of categories, including but not limited to the following:
  - Organized Crime – Motivated by profit, organized crime groups focus on data that permits fraud and identity theft: credit card data, banking information, personal information
  - Opportunistic – May be motivated by profit, notoriety, or political reasons, usually performing mass drive-by attacks in hopes of finding a vulnerable service
  - Hacktivism – Motivated by political, ideological, and personal reasons, hacktivists may attack sites of perceived opponents to capture sensitive information, or disable resources
  - Government Sponsored – Motivated by national, political, and military goals. Well-funded and usually employing security specialists, an extreme but unlikely threat

Basic Concepts: The Bad Actor

- What kind of bad actor do you think is responsible for the following HTTP request?

```
  158.69.241.103 - - [14/Sep/2018:13:23:03 +0000] "GET
/w00tw00t.at.blackhats.romanian.anti-sec:) HTTP/1.1" 404
                    10239 "-" "ZmEu“
```

- This request originated from an IP associated with a server in Quebec
- One of thousands of attacks on this server, with an attack occurring approximately every 5 seconds
  - Note that this server doesn't have a domain name and hosts absolutely nothing of value – don't assume that low-traffic or IP-addressed web servers won't be a target

☆ Basic Concepts: Vulnerabilities

- Vulnerabilities are weaknesses in a system that may be exploited by a bad actor
- We can inadvertently introduce vulnerabilities into our systems through bad code, or vulnerabilities may be introduced through code patches, modules, etc
  - Remember, the majority of the code on our servers is written by third parties: CentOS, Apache, PHP, etc – these can all introduce vulnerabilities
- Zero-day vulnerabilities are a special class of vulnerabilities; ones that are unknown to the developer. "Day Zero" is the day that the developer finds out (and often when the vulnerability becomes public knowledge)
  - Zero-day vulnerabilities in major web libraries (JS, Flash, etc) are a serious problem and must be patched promptly
  - In the time between a zero-day vulnerability becoming public knowledge and a patch being released to fix the exploit, your web services are vulnerable to bad actors

☆ Basic Concepts: Attacks

- An attack is when a bad actor attempts to exploit a vulnerability to gain access to a system or data that they otherwise wouldn't
  - An attack may focus on any number of system components:
    - Injection of malicious data within front-end web scripting components like HTML or JS
    - Accessing data in a database such as user logins or private information
    - Complete subversion of the server – acquiring shell access
- Attacks may be carried out by an individual, but more often are automated, allowing attacks to be performed on many targets at once
  - Often, subverted servers are used as a platform for launching additional attacks or furthering an attackers agenda – mailing out spam or fishing emails, attacking other servers, or redirecting users to other websites or content

Basic Concepts: Types of Attacks

- The Open Web Application Security Project (OWASP) publishes yearly reports on security risks that are faced by modern servers
- This covers a variety of potential vulnerabilities and issues, including:
  - Cross-Site Scripting attacks (XSS)
  - Injection attacks
  - Authentication vulnerabilities
  - Components with Known Vulnerabilities
  - File Upload Vulnerabilities

☆ Basic Concepts: Obscurity

- In 1883, cryptographer Auguste Kerckhoffs developed a set of rules to apply to military encryption, which includes the following rule:

  "*It should not require secrecy, and it should not be a problem if it falls into enemy hands*"

- The takeaway: You should not rely on the obscurity of your system components to maintain security
- When you write back-end code, assume that your code is already known by your potential attacker, and plan accordingly
  - For example, much of modern encryption technology is open source, but this doesn't have a negative impact on its security, because it doesn't rely on the algorithm being secret
- While you should not and cannot rely on obscurity as a security measure, the use of purposeful camouflage and subterfuge is a well-understood concept
  - Assume your attacker knows your system inside-and-out, but there's no point in giving them more information about your system

Basic Concepts: Obscurity

- Can you see some information that we're providing a potential attacker?

Basic Concepts: Total Security

- Barring special scenarios, such as air-gapped machines, the idea of "total security" is for the most part a myth
- The architectures that underpin the internet, software package development, management, and distribution always leave the potential for a vulnerability being introduced
  - Even if a vulnerability isn't introduced, there are things beyond our control that can compromise security – for example, a wiretap at the Internet Service Provider (ISP) which intercepts securely transmitted data
- The law of diminishing returns applies to security – at some point, you're preparing for scenarios that are either extremely unlikely, or beyond your control

An Honest Word on Web Security

- Movies, TV shows, and video games often misrepresent what a typical security attack actually looks like
- The type of web attacks that you're most likely to get hit by aren't targeted human hackers using cutting-edge technology
  - Most bad actors are looking for easy targets – consider a burglar looking for the one house in the neighborhood that left the front door unlocked
- 99.99% of the attacks you'll face are zombie machines probing for common vulnerabilities they can exploit
  - Conversely, that 0.01% of attacks, we can't do much about – dedicated hackers and state-sponsored bad actors have enough knowledge and computing power to circumvent most security measures

Security – Cross-Site Scripting Attacks

☆ Cross-Site Scripting (XSS)

- Cross-Site Scripting remains one of the most, if not the most, prominent security vulnerability in web applications today
- The concept is pretty simple: inject arbitrary client side script (JS) into a web request so that it is executed on client machines
- XSS vulnerabilities are split into two major categories, *persistent*, and *reflected:*
  - A Persistent XSS vulnerability is stored server-side. For example, a comment containing a malicious JavaScript string is stored in the database, and executed for all clients that visit the page the comment is rendered on
  - A Reflected XSS vulnerability includes the malicious script in a non-persistent form, such as a GET request parameter

An Example of Reflected XSS

- Consider the following URL:

  www.foo.bar/search?query=matthew

- Lets say for example the site echoes back "Results found for 'matthew'" and a list of search results
- Now consider the following URL:

  www.foo.bar/search?query=%3Cscript%3Ealert%28%27hello%27%29%3C%2Fscript%3E

- This is the URL encoded version of <script>alert('hello')</script>
- If the page echoes this out as the search parameter without escaping the script, the script can be executed
- This allows arbitrary script injection from what looks like a 'safe' URL

An Example of Persistent XSS

- Say for example your blog has a comment box at the bottom
- Someone leaves a comment that is <script>alert('hello')</script>
- If you don't escape script in comments, this script will be stored in your database and rendered (executed) for anyone who visits the page that the comment is made on
- This applies to pretty much any time that you are rendering user-generated content to other users on your website
  - Your SimpleChat application escapes potential scripts using the htmlspecialchars() function in PHP. If it didn't people could arbitrarily run JavaScript on other chat user's machines

What's the Big Deal?

- The question is: how much damage can you do with a client-side JavaScript execution? The answer: a lot!
- A keylogger can be written in ~20 lines of JS that will record all keystrokes you make on the page and send them back to a remote server via AJAX request
- Depending on how data is handled by the website, JavaScript may have access to session identification tokens and cookies, allowing login-related data (such as a valid user session identifier) to be dumped and sent back to a remote server
- Fudging web traffic and ad statistics – You could redirect users to an alternative website to drive traffic up, or place ads in an otherwise ad-free page to generate revenue for the attacker
- Once you have arbitrary script execution, the sky is the limit

☆ How to Defend against XSS

- We've already discussed a couple strategies to help defeat XSS:

NEVER TRUST USER GENERATED CONTENT

- Sanitize all user-generated inputs using the htmlspecialchars() function, and make use of safe echoing procedures. Recall that in Laravel, the {{ $var }} blade function will echo safely (rendering script as plain text), while the {!! $var !!} blade function will echo unsafely, rendering script as script
- These are primarily server-sided defenses, but they're not the only options. You can configure your system to enable client-side defense against XSS attacks as well.

☆ Using the Content Security Policy

- The Content Security Policy (CSP) is a header that can be passed along with an HTTP request that instructs the user's browser how to handle certain kinds of content
- The CSP can be configured in Apache's config file, or using a <meta> header element.
- The Content Security Policy specifies from which domains content can be loaded
- CSP is supported by most major modern web browsers, although the degree of support is dependent on the browser

Example Content Security Policy

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src ‘self’ ‘unsafe-inline’;">
```

- The meta http-equiv tag allows you to add data as if it was an HTTP header. In this case, it would be a CSP header that looks like this:

```
Content-Security-Policy: default-src ‘self’; script-src ‘self’ ‘unsafe-inline’;
```

- The default-src defines what is allowed as a default source, and ‘self’ states that only sources from this same domain are allowed
- Script-src defines where JS scripts may be called from. ‘unsafe-inline’ allows inline JS scripting – if you remove ‘unsafe-inline’, you may only include scripts as JS files.
- Preventing unsafe-inline scripts from executing, and restricting scripts to only JS files from within the same domain largely protects against XSS exploits by preventing execution at the client-side

Dealing with XSS Vulnerabilities

- So, we have two different options for dealing with the potential for XSS vulnerabilities – server side sanitization of user input, and client-side content security policy that operates at the browser level
- An additional level of protection: using the X-XSS-Protection HTTP header, which looks like this:

```
X-XSS-Protection: 1; mode=block
```

  - The X-XSS-Protection header tells the browser to detect and automatically block reflected XSS attempts
  - Largely unnecessary if a good CSP is implemented, but it can't hurt to implement it
- Takeaway: XSS protection should be enabled at both the server and client side to provide as much protection as possible

Demonstration of XSS

```
Route::get('/search', function () {
    $var = $_GET['term'];
    return "<h1>Error 404: Sorry, resource " . $var . " not found.</h1>";
});
```

Security – SQL Injection

☆ SQL Injection (SQLi)

- SQL injection is in a similar vein to XSS in that it involves injecting additional code, but in some ways is more devious
- While XSS targets clients (users), SQL Injection targets the server itself, specifically the database
- A bad actor performing an SQL Injection could potentially obtain dumps of sensitive information (login details, emails, credit card information, etc), or they could simply destroy your databases – dropping tables, deleting records, etc
- Like XSS, SQLi is dependent on user input

Anatomy of an SQL Injection

- SQL Injection is the result of echoing user input directly into an SQL statement without sanitizing it first
- Consider the following SQL statement:

```
SELECT * from users WHERE password = '$var'
```

- Now, consider what happens if the user input $var that we receive isn't a text string (password) as we expected, but looks more like this:

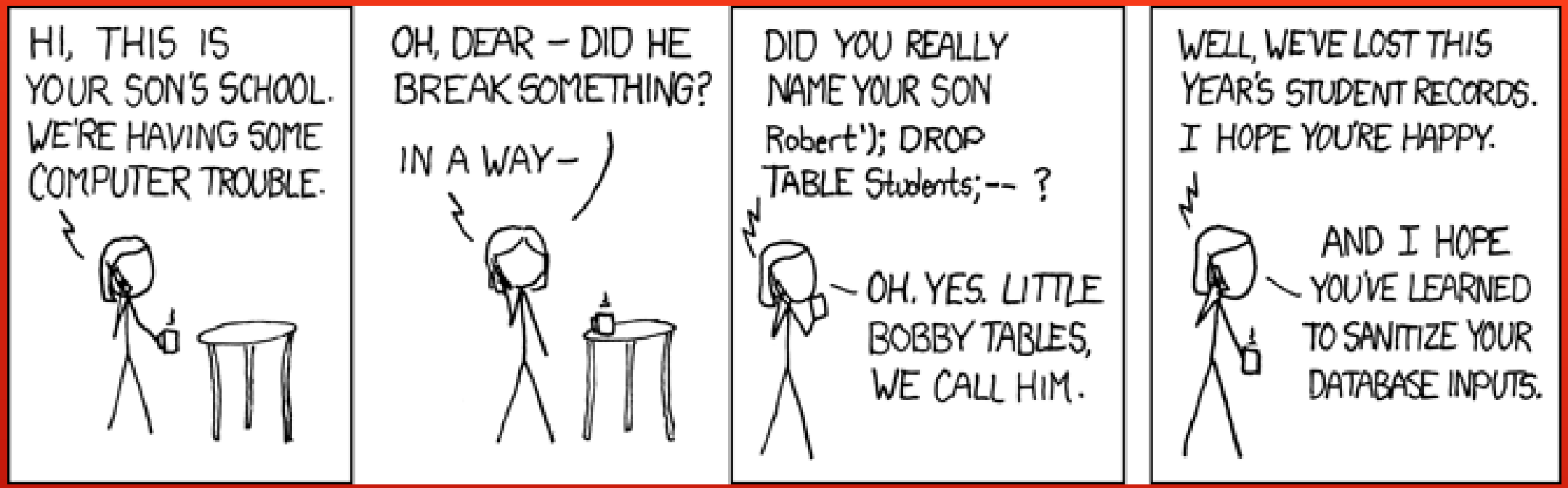
```
1' OR username = 'admin
```

- The result? An SQL query that looks like this:

```
SELECT * from users WHERE password = '1' OR username = 'admin'
```

SQL Injection Methods

- There are several ways of designing SQLi inputs to manipulate the results of a query:
  - In the previous example, we used an OR statement to tack on an additional 'where' clause with our own conditions
  - Use a ';' character to end the current SQL statement and then add an additional statement such as DROP, UPDATE, or INSERT
  - Abuse the SQL comment string '--' to escape part of an SQL statement that comes after the insertion
- There are a great number of ways to perform an SQL injection – for this reason, it's usually better to use a pre-built solution for sanitizing input (Like Laravel QueryBuilder, or PDO prepared statements).
  - However, even these solutions can miss certain edge cases of SQLi



With credit to Randall Munroe of XKCD

An In-Class Exercise

- The following code exists on my server:

```
Route::get('/findUser', function () {
    $var = $_GET['user'];
    $results = DB::select(DB::raw("SELECT * FROM
users WHERE password = '$var'"$
    return json_encode($results);
});
```

- Visit **http://167.99.183.179/findUser?user=**
- See what data you can get out of this query through SQL injection on the *user ($var)* variable

☆ The Statistics

- A March 2012 study pegged XSS as responsible for 37% of vulnerabilities, and SQL injection responsible for another 16%, for a total of 53% of vulnerabilities
- This means you can protect yourself from more than half of all vulnerabilities by protecting yourself from XSS and SQLi exploits
- How do we do that?

NEVER TRUST USER INPUT

- Sanitize all user input, no exceptions. Ensure that there is no chance that a user can insert malicious script or SQL into your queries or your HTML
  - Use server-side sanitization, client-side CSPs, escaped SQL queries, and validation to ensure input is safe

Security – SQLi: Minimizing Damage

Minimizing Damage

- Despite all our precautions, there's always the very slim possibility of getting sideswiped by an edge case – an extremely complex injection that manages to circumvent our sanitization
  - The chances are slim, especially for basic queries, but DBMS systems are extremely complex, and through messing around with character sets and encoding, it *may* be possible to get past some forms of sanitization
- This means that we should prepare for the worst: someone has a complete dump of our database, and has total control over it to do as they please
- How can we minimize the damage in this scenario?

Protecting Sensitive Data

- Ideally, we never store sensitive data within our database. However, reality dictates that sometimes we have to
- If it's data that only needs to be compared against an input (such as a password), hash it
  - Hashes are for the most part one-way functions: you cannot derive the input password from the output hash
  - Since we only need to compare, we can simply compare the hashed user input against the stored hashed password to see if a user has entered the correct password
  - If our database is compromised, it's much better that the bad actor receives a list of usernames and hashed passwords, rather than a list of usernames and plaintext passwords (which could be used in an attempt to break into accounts on another site)
- If it's data that needs to be read, encrypt it before insertion into the database. Assuming that your encryption keys remain secure (an SQL injection by itself shouldn't enable viewing or handling of system files), the encrypted data should be safe, if a good encryption algorithm is used

A General Thought

- In web security, it is best to be proactive, rather than reactive
- Prepare your security the best you can, but have a backup plan. Make sure that if you do have a security breach, that you protect your customers data and your application
  - Hash, Encrypt, Backup
- Don't wait for something to happen and then be caught trying to pick up the pieces. A good mitigation and recovery plan will save a lot of trouble down the road

That's All Folks

Next Week – More Security + Intro to Flask