HOW NOT TO BUILD A WEB APP

LESSONS LEARNED FROM THE MOST VULNERABLE APP IN THE WORLD

WHO AM I?

- Robert Babaev
- 2nd Year Computer and Internet Security
- Started Cybersec journey in October 2019
- Decided to make a cybersec startup when I had no idea how any of it worked
- H4TT, Northsec 2020
- VP CTF Affairs - Ravens

WHY SHOULD YOU CARE?

- Less secure site = more headaches
- Not many know how to break
  - That is, on dev side
- Security breaches common
  - Breaks trust, lose customers, lose money
- 2019, 66% of SMEs reported cyberattacks globally
  - Over 50% reported data breaches

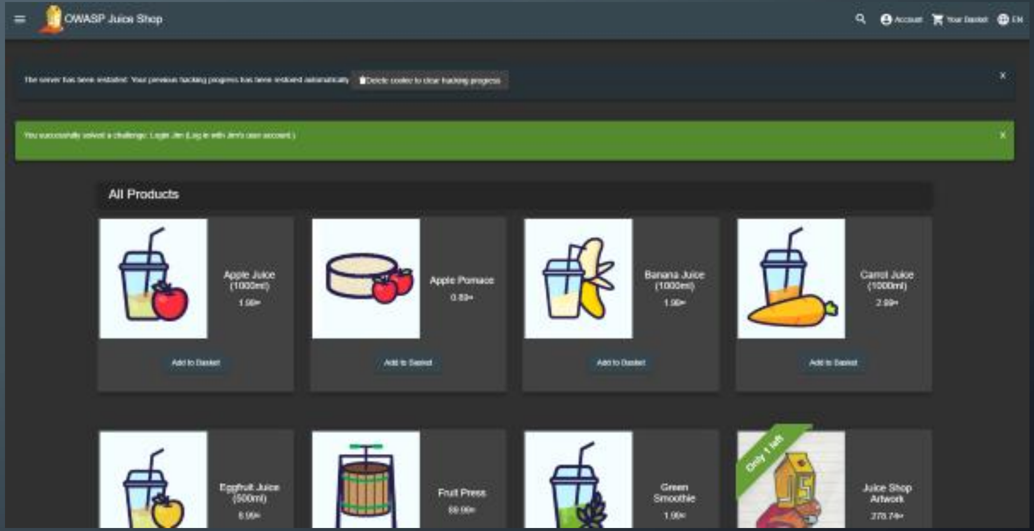
SOURCE: PONEMON INSTITUTE

PREREQUISITE KNOWLEDGE

- To get the most out of this:
  - How web app requests work
  - Some HTML/CSS/JS
  - SQL is handy

WHAT ARE WE ATTACKING?

- OWASP Juice Shop
- Extremely vulnerable web app
- Lots of opportunities
- Freely available



The server has been restarted: Your previous hacking progress has been restored automatically. Delete cookie to clear hacking progress

You successfully solved a challenge: Login Jim (Log in with Jim's user account.)

THINGS TO KEEP IN MIND

- User input is EEEEVILLLLLLLL
  - Sanitize and make sure you don’t have input going anywhere sensitive
- Assume users can and will modify requests
  - This can bypass some non-request measures
  - Avoid http, use https
- Don’t assume just because something is hidden that people won’t find it
- Keep verification server-side if you can

HOW DO ANY OF THESE ATTACKS WORK?

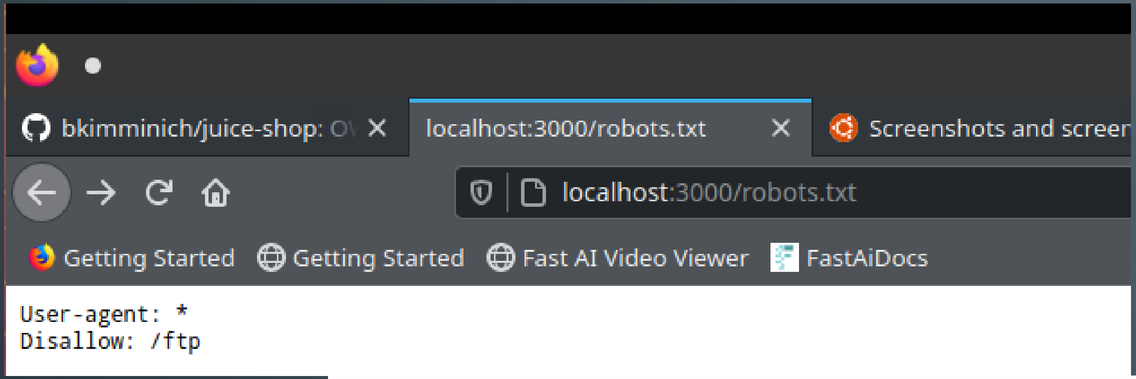
- Based on one or more of OWASP Top 10
  - Most common vulnerabilities found year to year
- Sampling of vulnerabilities
  - Injection – SQL, shoving code to override server-side
  - XSS – Shove JS into input fields
  - Broken Authentication – Bad verification, weak password checks, etc.
  - Sensitive Data Exposure – Self Explanatory
  - And more!

LET’S GET STARTED

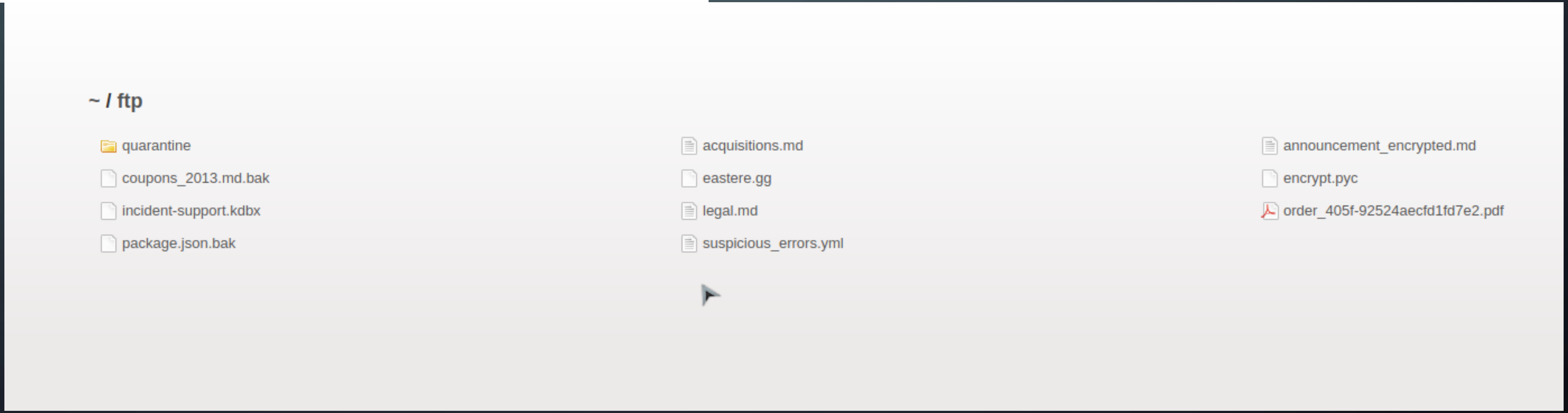
FIRST VULNERABILITY: ROBOTS.TXT

- Handles webcrawlers
  - Also gives potential insight into hidden directories
  - If page doesn’t have protection, can access those hidden directories

ROBOTS.TXT



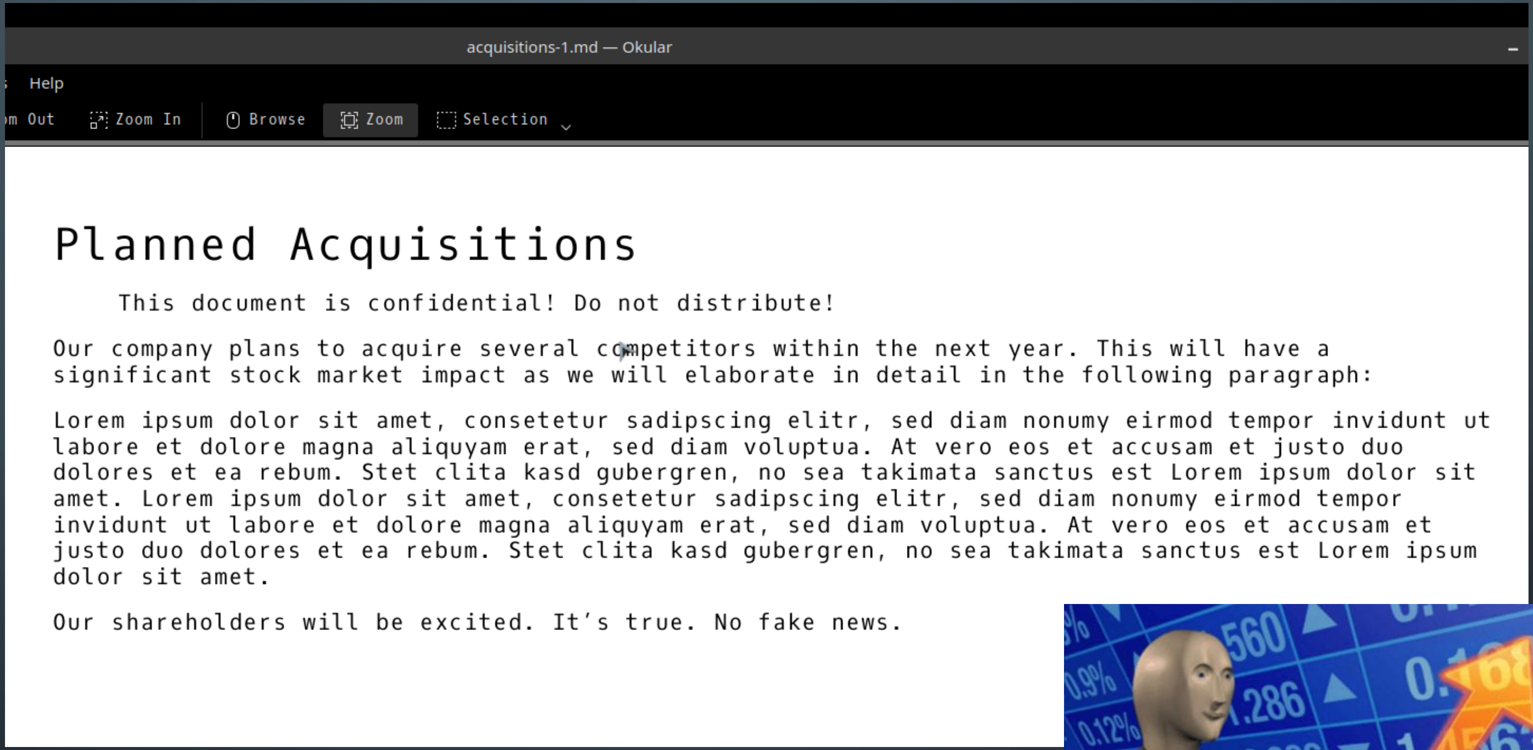
```
User-agent: *
Disallow: /ftp
```



~ / ftp

```
quarantine           acquisitions.md          announcement_encrypted.md
coupons_2013.md.bak  eastere.gg               encrypt.pyc
incident-support.kdbx legal.md                order_405f-92524aecfd1fd7e2.pdf
package.json.bak     suspicious_errors.yml
```

ROBOTS.TXT



Planned Acquisitions

This document is confidential! Do not distribute!

Our company plans to acquire several competitors within the next year. This will have a significant stock market impact as we will elaborate in detail in the following paragraph:

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Our shareholders will be excited. It's true. No fake news.

Sensitive data exposure!

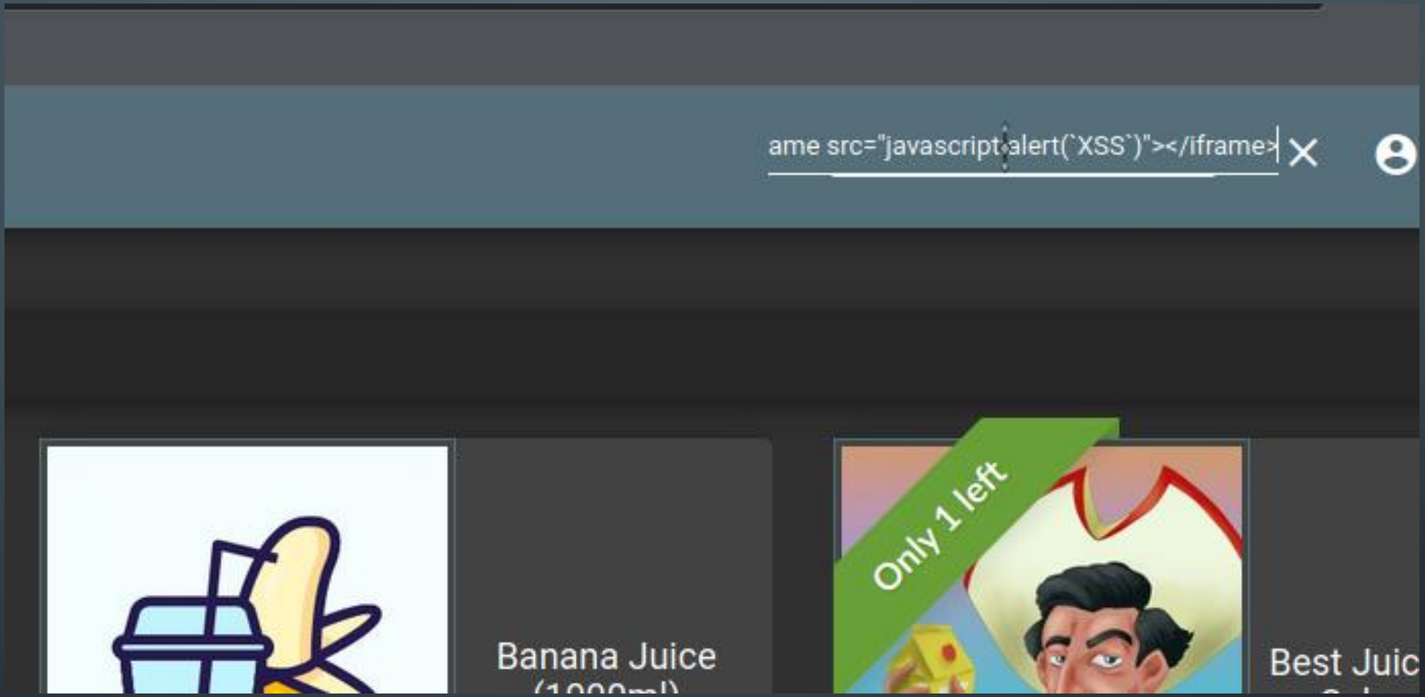
ROBOTS.TXT

- How to defend against it?
  - Make sure that you have a way of handling requests to every possible location in the site
  - Make sure these requests have proper authentication and authorization

SECOND VULNERABILITY: XSS

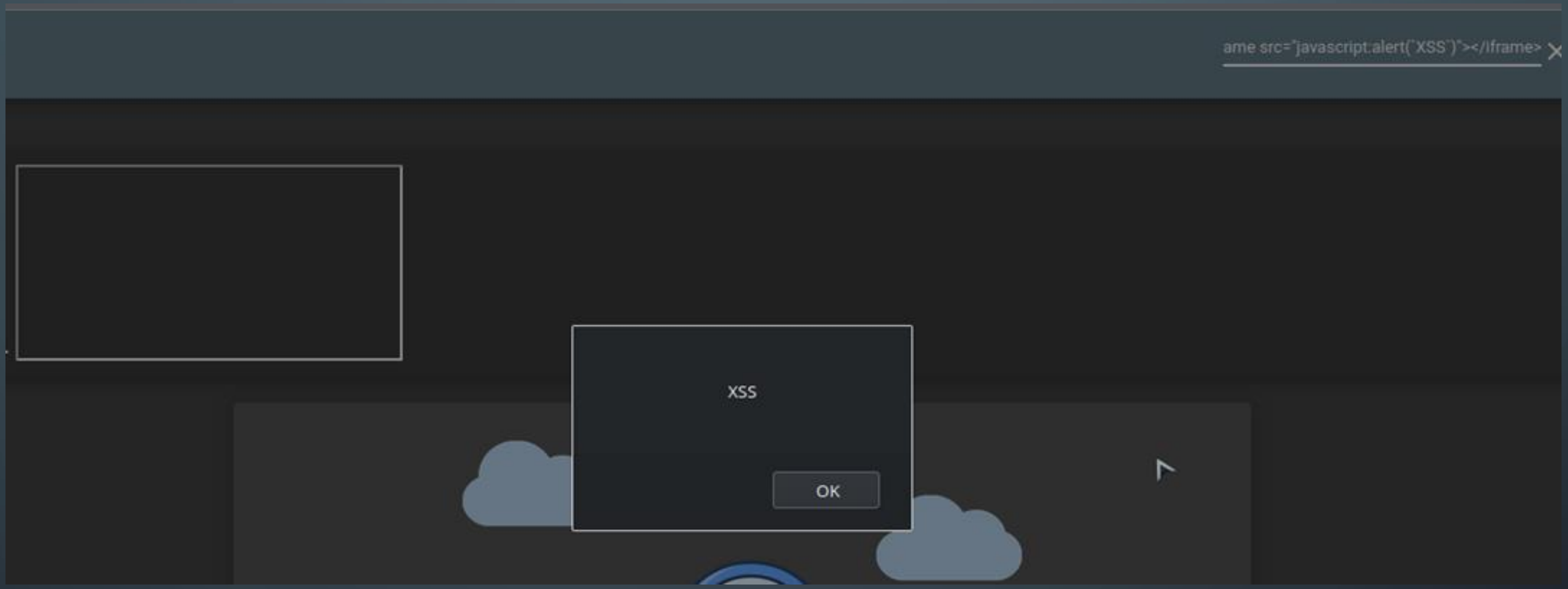
- Injecting script tags into places they shouldn't be
  - IRL, associated with a payload
  - Changing form submission functions, modifying page, etc
  - Here, just need an alert
- Don't think a simple filter will work!
  - Ways of bypassing
  - UTF-7, onError, URL hackery, iframes . . .

XSS



```
ame src="javascript:alert(`XSS`)"></iframe>
```

XSS



```
ame src="javascript:alert("XSS")"></iframe>
```

```
XSS
```

XSS

- Why is this a problem?
  - JS payloads can mess with your page's DOM
  - Change submit functions to send usernames and passwords into the aether
  - Submit forms you would never want to submit
  - Can also store XSS payloads via some user submission
    - Comments
    - Usernames

XSS

- How to defend against it?
  - Don’t allow JS to run from user input fields
  - Don’t put untrusted data except in allowed locations
  - Escape characters if possible
  - One of the trickier ones to defend against

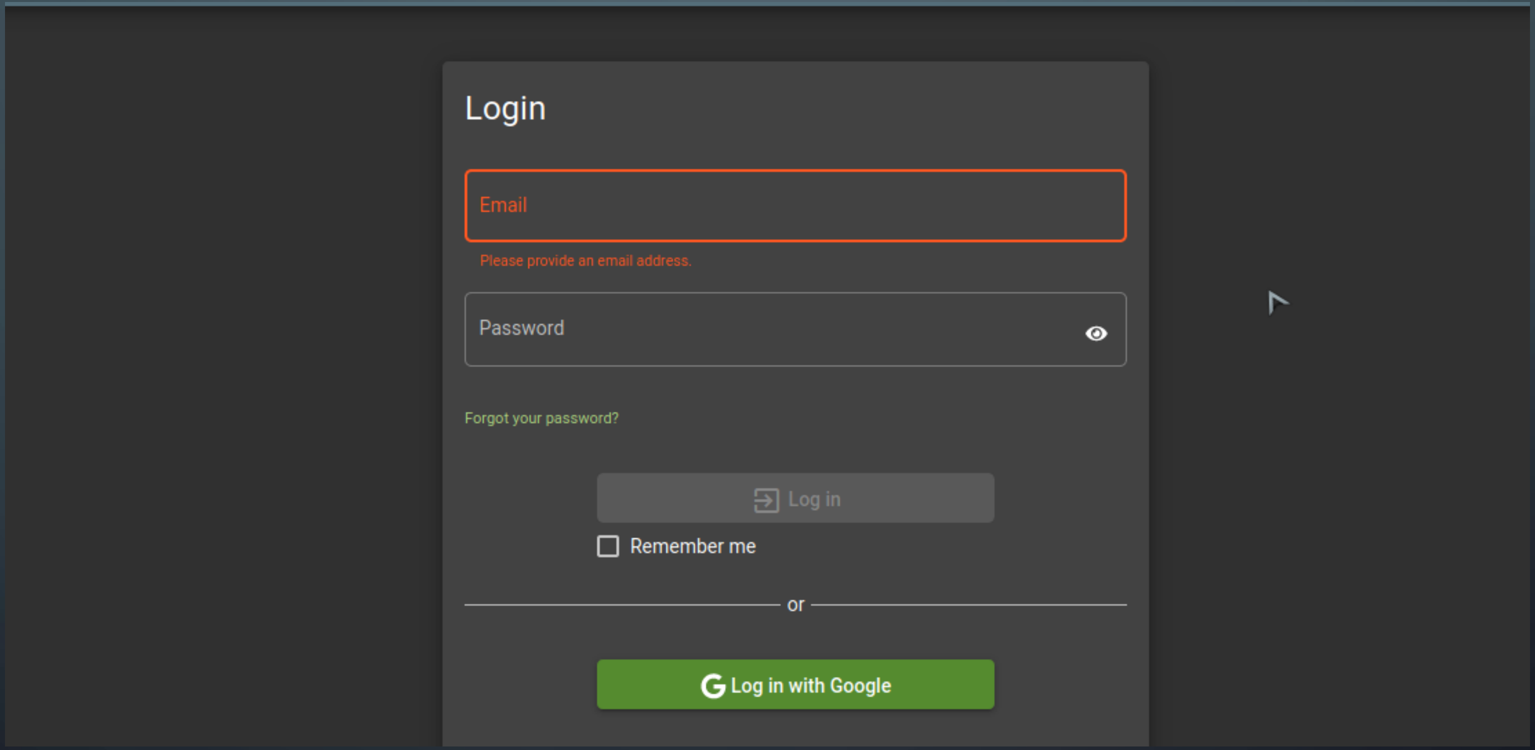
THIRD VULNERABILITY: SQL INJECTION

- Injecting SQL code to get some . . . Interesting results
- Can be used for a number of things
  - Passwordless logins
  - Data extraction
  - Deleting an entire table
- A normal SQL Query runs something like SELECT * FROM USERS WHERE Username=“<and then input goes here>”

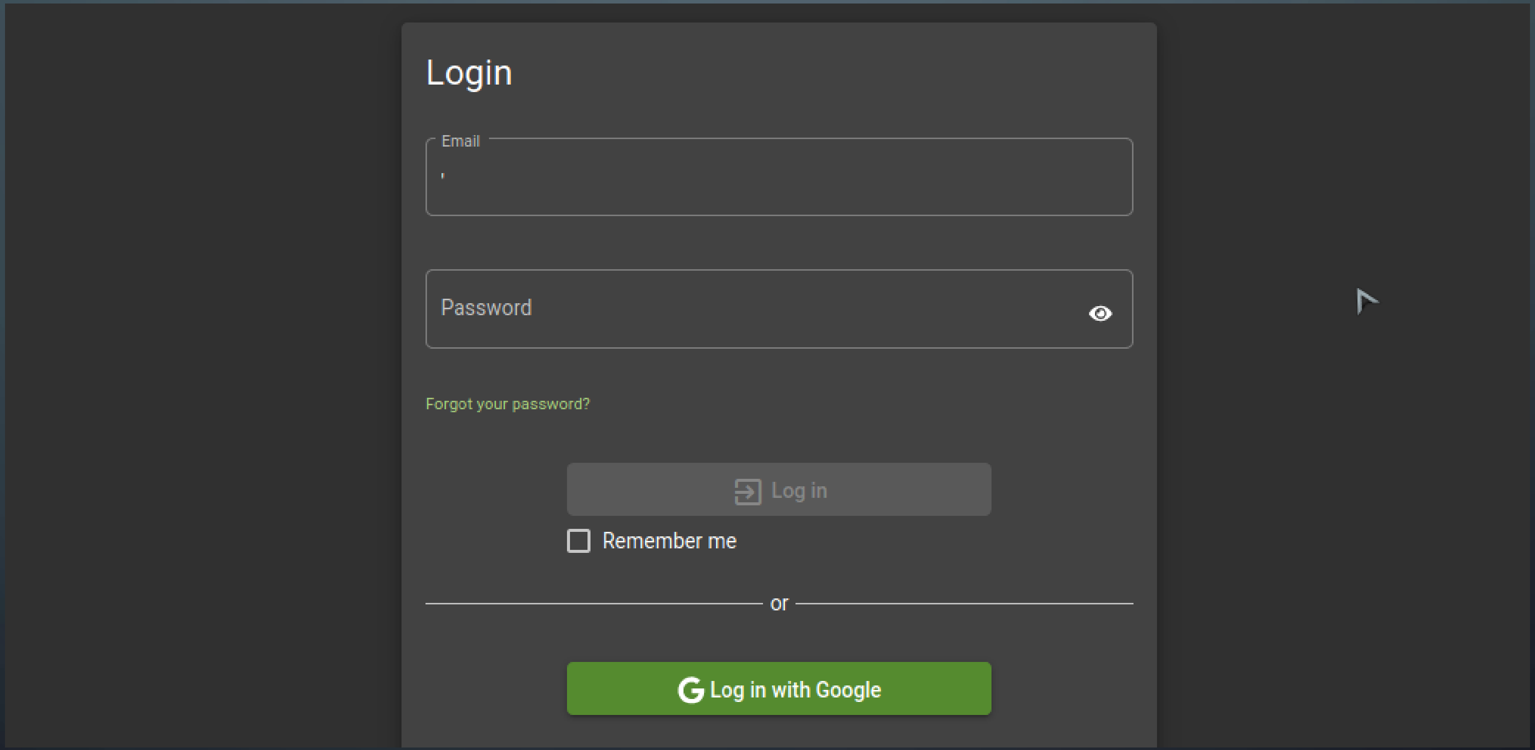
THIRD VULNERABILITY: SQL INJECTION

- Where's the problem with that query I just showed?
- The username data field is going directly into the query
  - NO sanitization
  - Extremely open to attacks
- Let's take a look

SQL INJECTION



SQL INJECTION


Login

Email

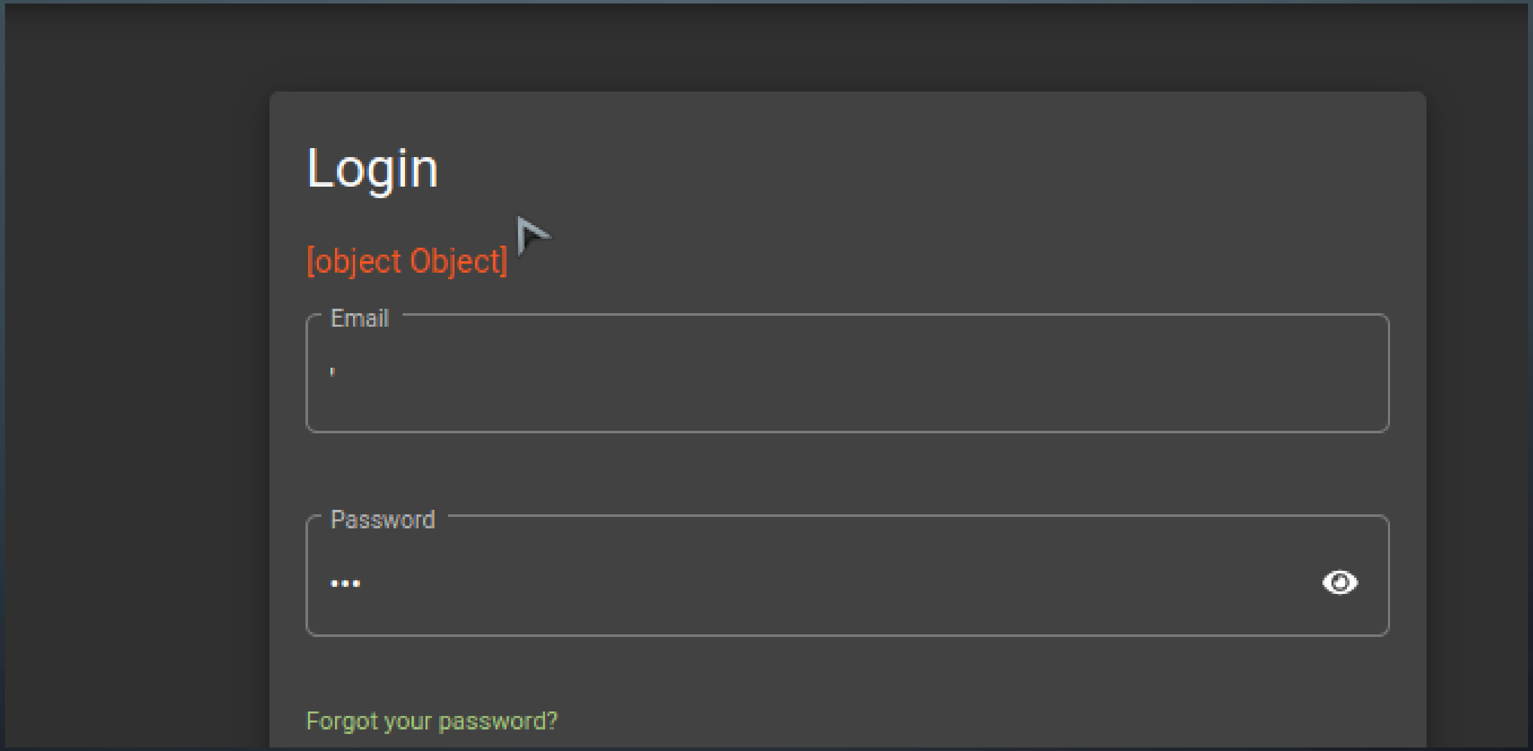
'

Password

Forgot your password?

SQL INJECTION

Wait…


Login

[object Object]

Email

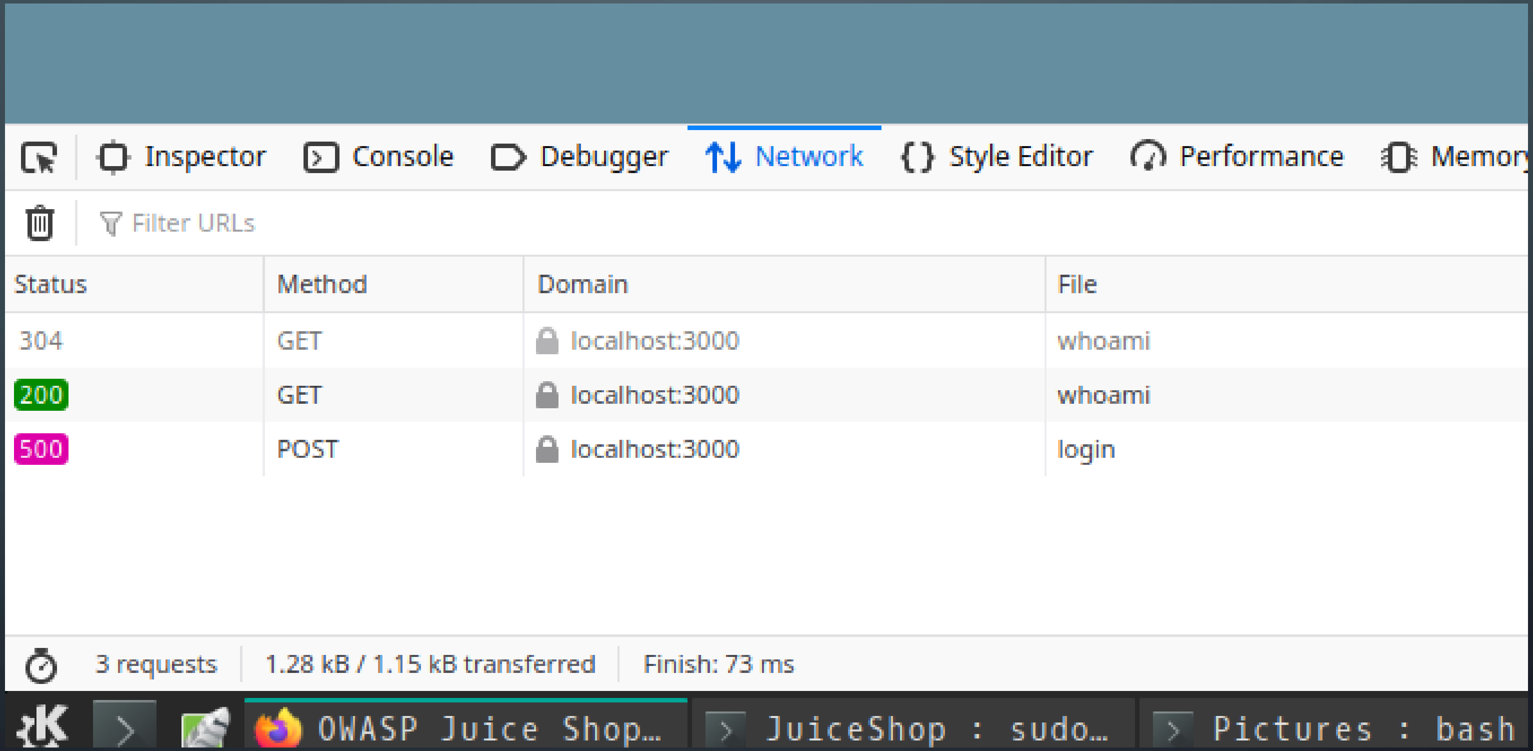
'

Password

•••

Forgot your password?

SQL INJECTION

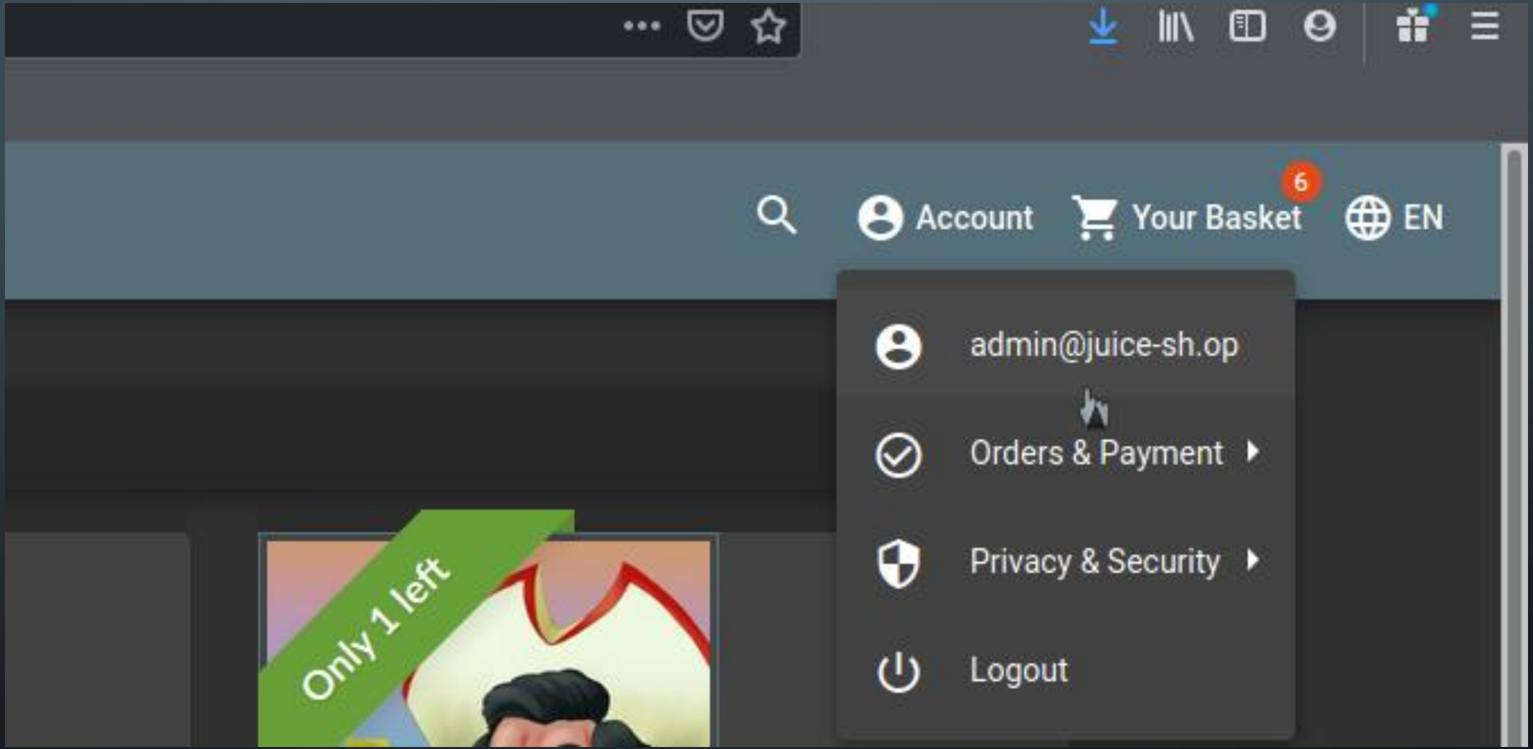


Status	Method	Domain	File
304	GET	localhost:3000	whoami
200	GET	localhost:3000	whoami
500	POST	localhost:3000	login

3 requests | 1.28 kB / 1.15 kB transferred | Finish: 73 ms

SQL INJECTION

Using the username ‘ OR 1=1;-- and literally any password, we get:



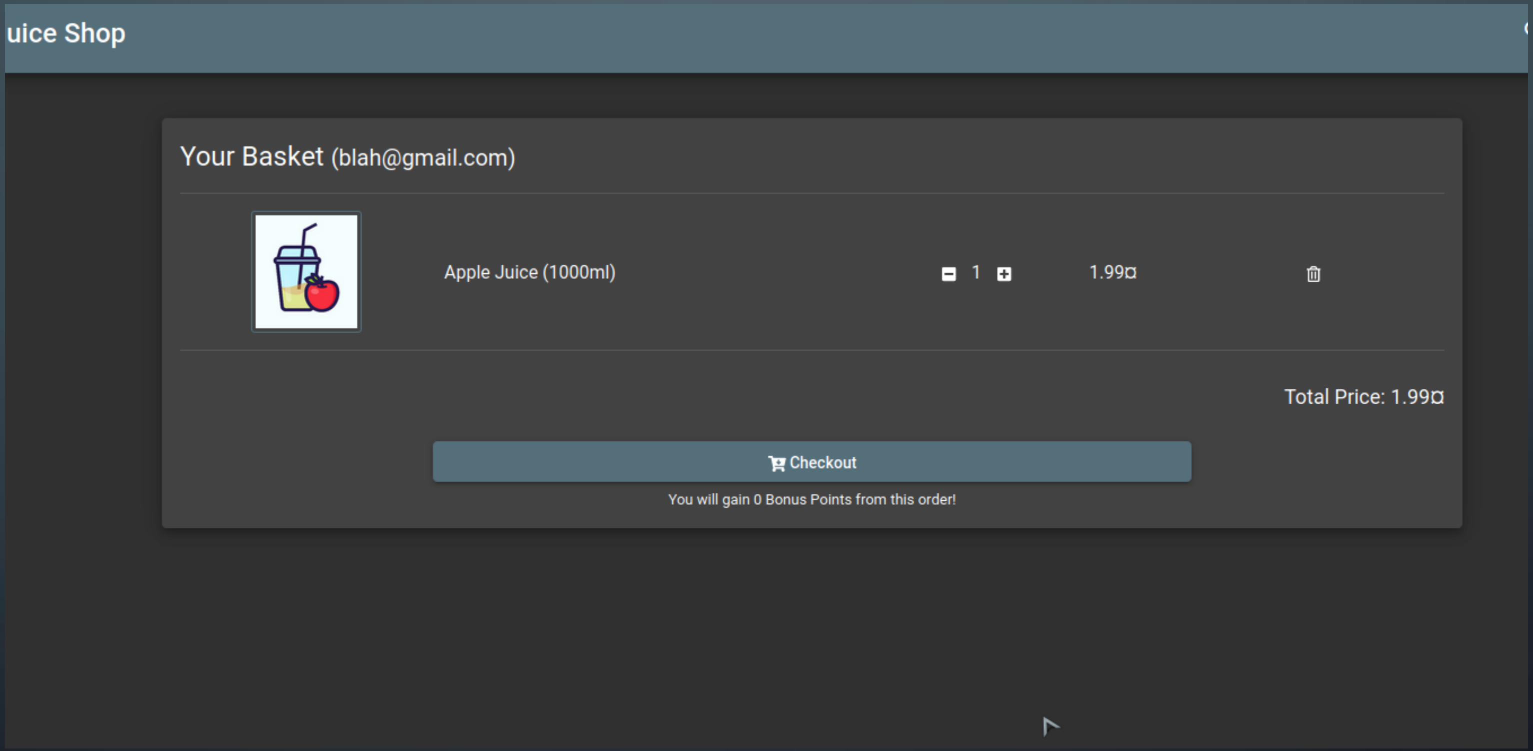
SQL INJECTION

- How to defend against it?
  - Sanitize your inputs!
  - Precompile SQL statements
  - Use stored procedures

FINAL VULNERABILITY: REQUEST MODIFICATION

- Modifying requests is actually surprisingly easy to do
- Browser dev tools give you all the resources necessary
  - Check requests/responses
  - Modify request payloads

REQUEST MODIFICATION



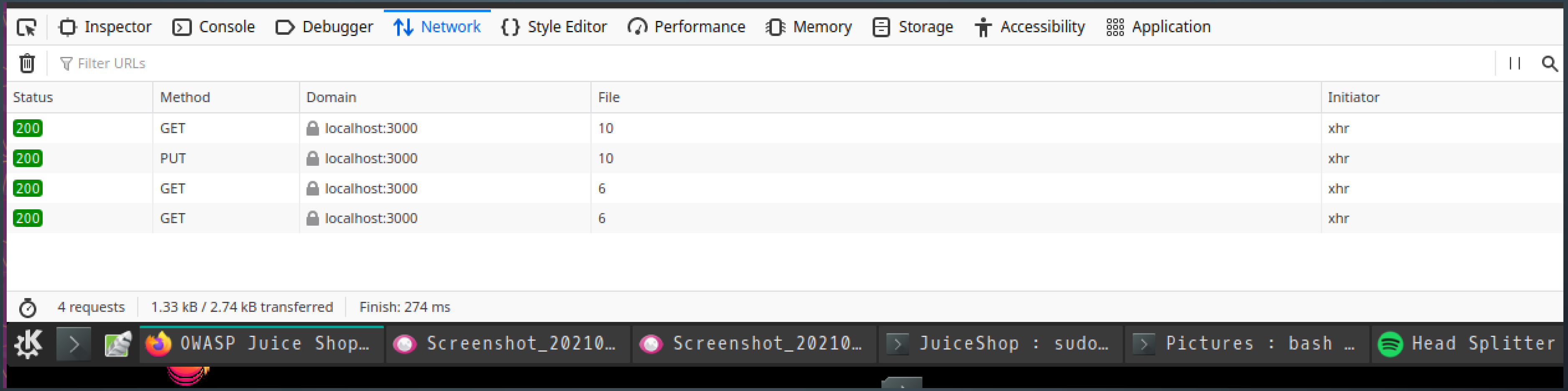
Your Basket (blah@gmail.com)

	Apple Juice (1000ml)	1	1.99¤	

Total Price: 1.99¤

You will gain 0 Bonus Points from this order!

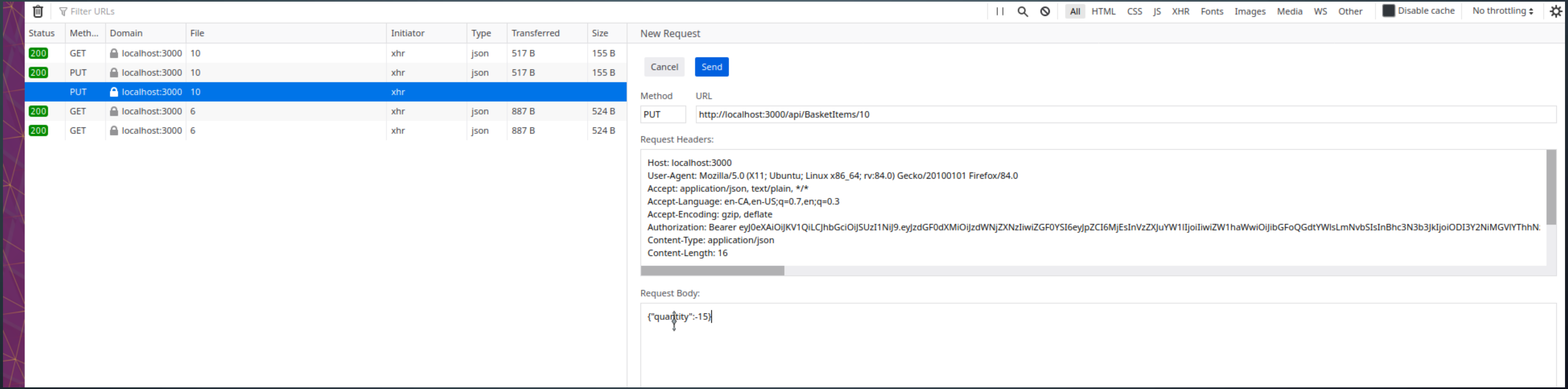
REQUEST MODIFICATION



Status	Method	Domain	File	Initiator
200	GET	localhost:3000	10	xhr
200	PUT	localhost:3000	10	xhr
200	GET	localhost:3000	6	xhr
200	GET	localhost:3000	6	xhr

4 requests | 1.33 kB / 2.74 kB transferred | Finish: 274 ms

REQUEST MODIFICATION



Status	Meth...	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost:3000	10	xhr	json	517 B	155 B
200	PUT	localhost:3000	10	xhr	json	517 B	155 B
	PUT	localhost:3000	10	xhr			
200	GET	localhost:3000	6	xhr	json	887 B	524 B
200	GET	localhost:3000	6	xhr	json	887 B	524 B

New Request

Cancel Send

Method: PUT

URL: http://localhost:3000/api/BasketItems/10

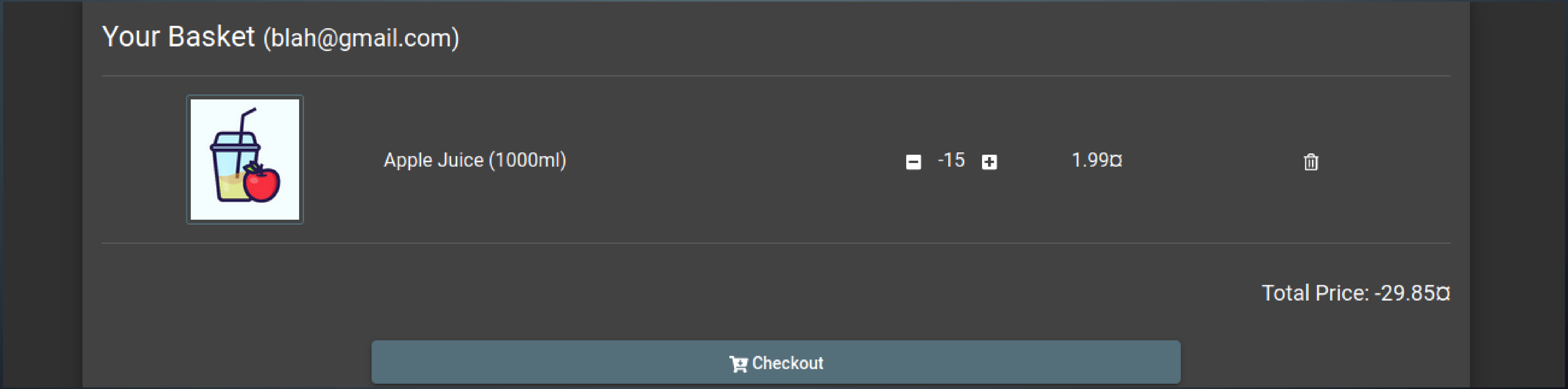
Request Headers:

```
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:84.0) Gecko/20100101 Firefox/84.0
Accept: application/json, text/plain, */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MjEsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJibGFoQGdtYWlsLmNvbSIsInBhc3N3b3JkIjoiODI3Y2NiMGVlYThhN
Content-Type: application/json
Content-Length: 16
```

Request Body:

```
{"quantity":-15}
```

REQUEST MODIFICATION



Your Basket (blah@gmail.com)

	Apple Juice (1000ml)	-15	1.99¤	

Total Price: -29.85¤

REQUEST MODIFICATION

- How to defend against it?
    - Validate client requests serverside
    - Always make sure values are within expected ranges
        - If not, reset to some default
    - Make sure authentication lines up with access

THIS IS JUST THE BEGINNING

- Other attacks:
  - CSRF
  - Iterable user IDs
  - Cookie modification
  - Deprecated interfaces
  - So much more

THIS IS JUST THE BEGINNING

- These are only a small sample of the possible vulnerabilities your site could have
- Don’t just build web apps, build secure web apps!
- How do you test?
  - Bug bounties
  - Pentests
  - Ethical hacking

THANKS FOR COMING!

QUESTIONS?