November CTF Challenge: Exploiting JWT vulnerabilities to achieve RCE

BY AYOUB · NOVEMBER 26, 2025

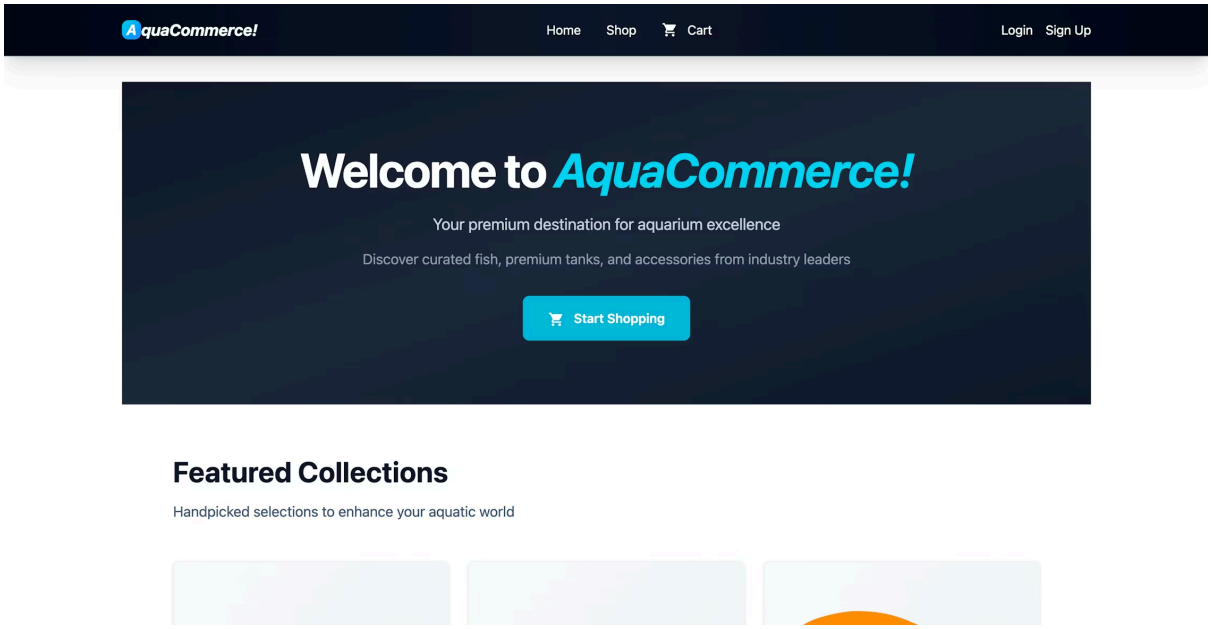
At Intigriti, we host monthly web-based Capture The Flag (CTF) challenges as a way to engage with the security research community. This month, we've decided to take on a challenge ourselves as a way to give back to the community. In response to one of our recent articles, we decided to focus on JSON Web Token (JWT) vulnerabilities.

This article provides a step-by-step walkthrough for solving November's CTF challenge while demonstrating techniques for exploiting JWT vulnerabilities in flawed authentication implementations.

Let's dive in!

Challenge overview

AquaCommerce! presents itself as a modern e-commerce website where users can purchase all their fishing and aquarium accessories. A quick look at the top navigation bar reveals that the challenge provides access to an account, likely to help customers manage their orders and make cart sessions persistent.



INTIGRITI CTF 1125: AquaCommerce!

Going back to the challenge rules, we can notice the following:

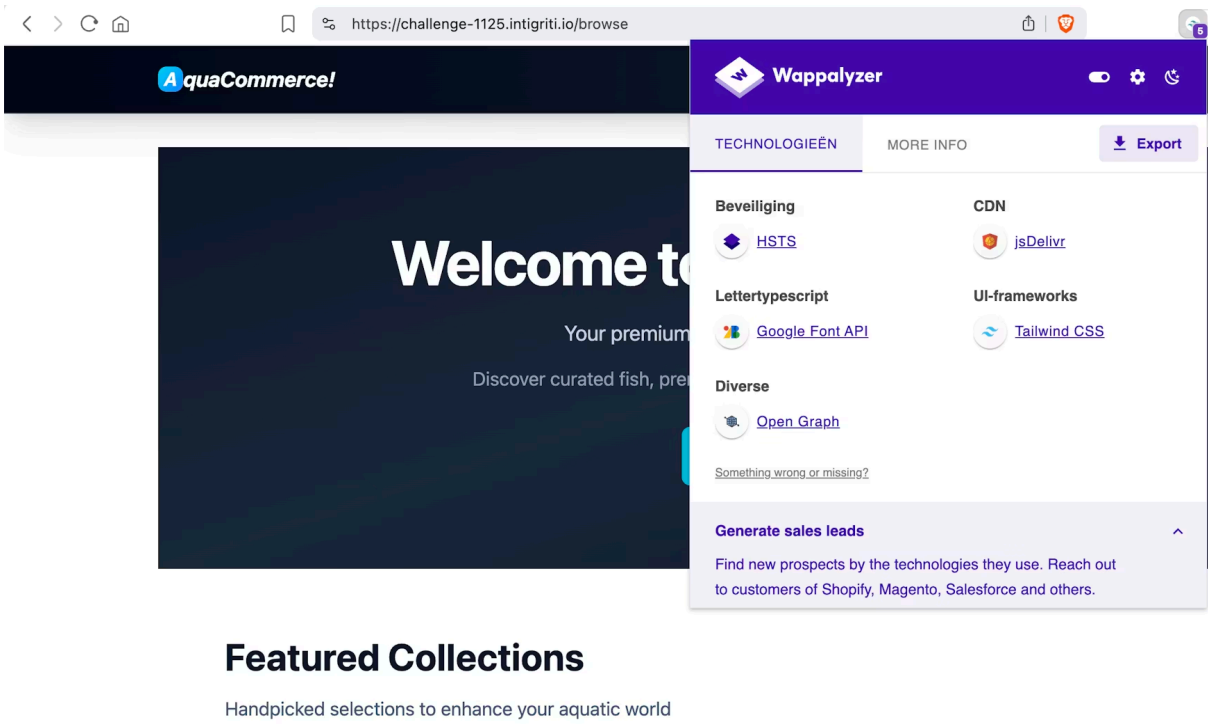
- We must find a flag in the following format: `INTIGRITI{.*}`
- The correct solution should leverage remote code execution

- Self-XSS or MITM attacks are not allowed
- And any attack that requires user interaction is out of scope

Unlike previous challenges, we don't have access to the source code. This means that we'll need to gather some more information about how the target application is built to better understand our pathway to achieving remote code execution.

Initial reconnaissance

As usual, we'll need to map out any possible information that's related to our target and that could be of use later on. Using tools like Wappalyzer and BuiltWith, we can retrieve the following:

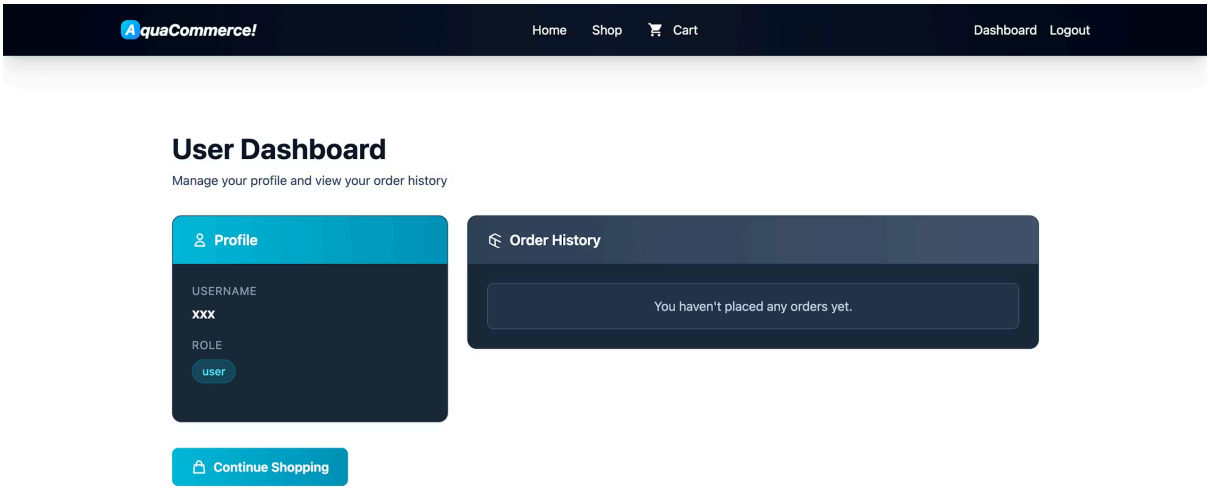


Wappalyzer result

Sometimes, applications are hosted in front of reverse proxies, like in this instance. This allows developers to practically decide what to forward from the client to the server. This challenge appeared to have been deployed behind Nginx, a popular reverse proxy service, as confirmed by the 404 page.

Difference between a forward proxy vs reverse proxy server

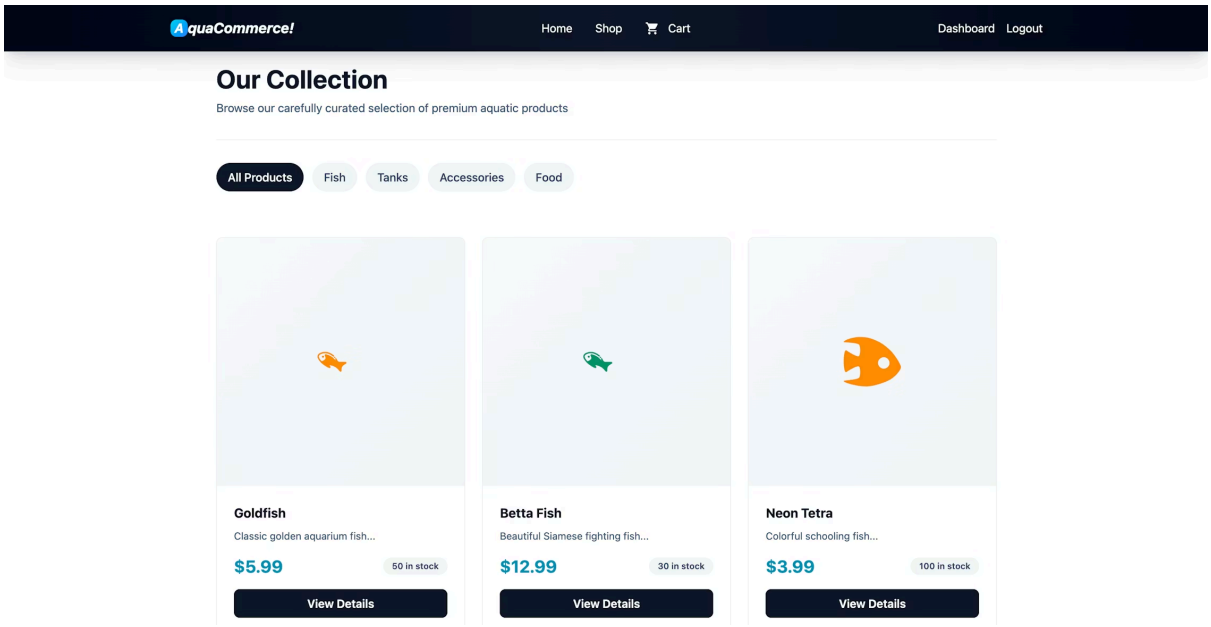
In cases like these, we'll need to move on and see if we can find relevant information about the target elsewhere. Let's take a more detailed look at the application and start with registering for a new account. After signing up, we're greeted with a clean, modern dashboard. The first thing that caught our attention was the navigation bar. There are several links present, and it's the "Dashboard" link that caught our attention.



AquaCommerce! user dashboard

The dashboard itself was fairly minimal. It displayed our username, role (which showed as "user"), and an empty order history. Nothing immediately stood out as vulnerable here, but the role badge was interesting, as it suggested there might be other roles in the system, as in-app roles are commonly implemented to allow for role-based access controls.

Before we make any assumptions, we must examine all the remaining functionalities that the target has to offer. One of them was order processing, a typical e-commerce function that allows users to add items to their cart and go through the checkout process. Although the ordering process went smoothly, we noticed that the confirmation page explicitly stated, "This is a demonstration. Your order has not actually been placed in the system."



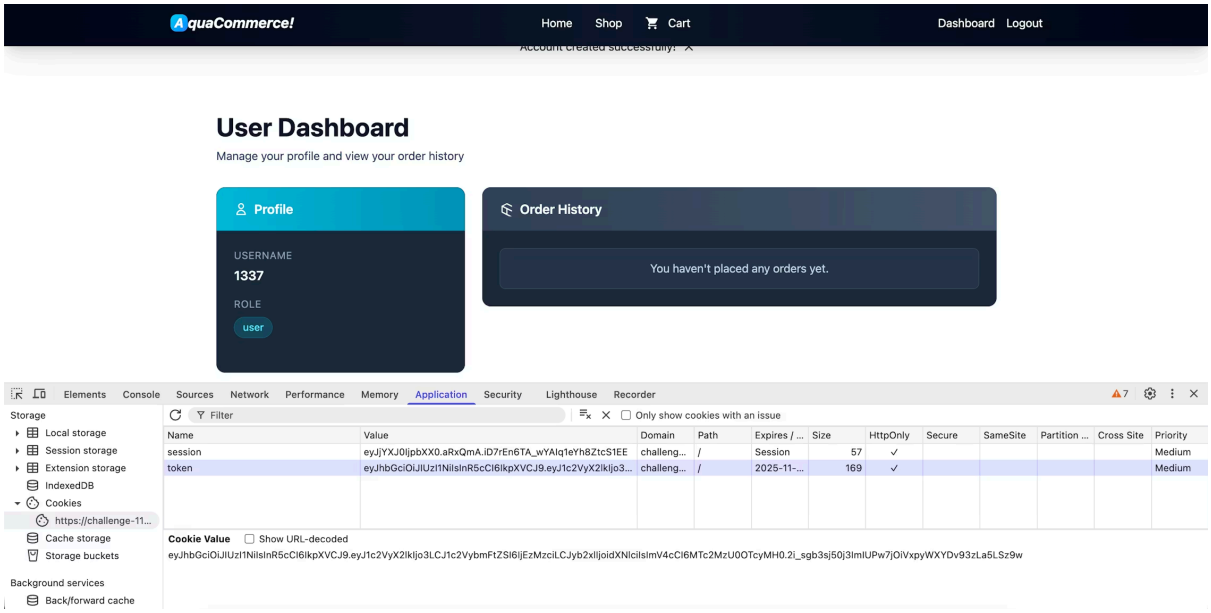
AquaCommerce! Shop Catalogue

This likely indicates that no actual processing is being done on the backend, ruling out possible SQL and NoSQL injection attacks that can, in severe cases, also be leveraged to execute code on the vulnerable server.

Let's take a closer look at the authentication process. The HTTP response from earlier seemed to indicate that JWTs are used to keep track of individual user sessions.

Authentication via JWTs

Signing into our newly created account, we can notice a new cookie has been assigned to our session. From its structure, we can easily tell that this is a JSON Web Token:



Session handling within AquaCommerce!

Decoding our JWT payload reveals the following claims:

```
{
  "user_id": 7,
  "username": "1337",
  "role": "user",
  "exp": 1763549720
}
```

The presence of a `role` field immediately sparked our curiosity. If regular users have `"role": "user"`, there's possibly an admin role supported. From the JWT exploitation article, we understand that the incorrect implementation of JSON web tokens can lead to all sorts of vulnerabilities. In this instance, we'd want to forge our own tokens and see if the application supports any other roles.

Exploiting JSON Web Token vulnerabilities

Learn more about exploiting JWT vulnerabilities in our comprehensive article:

https://www.intigriti.com/researchers/blog/hacking-tools/exploiting-jwt-vulnerabilities

Attempting the JWT 'none' algorithm attack

JWT tokens can be vulnerable to the `none` algorithm attack, a classic vulnerability that stems from improper validation of the `alg` JWT header property. It allows any attacker to forge tokens by removing the signature entirely and setting the algorithm to `none`.

In practice, this would involve us:

1. Decoding the header part
2. Setting the header property `alg` to `none`
3. `Tampering with the claims`

4. And lastly, removing the signature altogether

If the backend does not throw an exception and accepts the JWT with no signature, we would be able to practically forge tokens ourselves. Let's try it.

First, we'll need to decode the header property and set the **alg** header property to **none**:

```
{
  "typ": "JWT",
  "alg": "none"
}
```

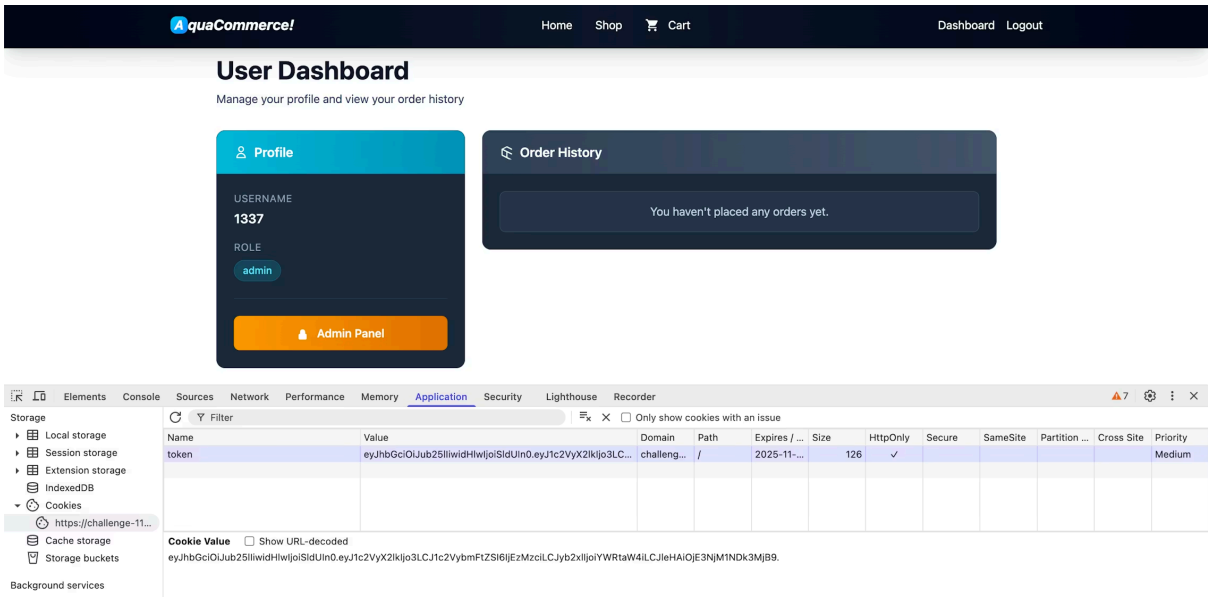
Next, we need to tamper with the payload to give ourselves admin privileges:

```
{
  "user_id": 7,
  "username": "1337",
  "role": "admin",
  "exp": 1763549720
}
```

Finally, we'll need to base64url-encode both the header and payload, then join them with a period, followed by an empty signature (another period with nothing after it). Your final JWT token will look like the following:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyX2lkIjo3LCJ1c2VybmFtZSI6IjEzMzciLCJyb2xlIjoiYWRtaW4iLCJleHAiOjE3NjM1NDk3MjB9.
```

Let's try it in our request and see if the application accepts it:

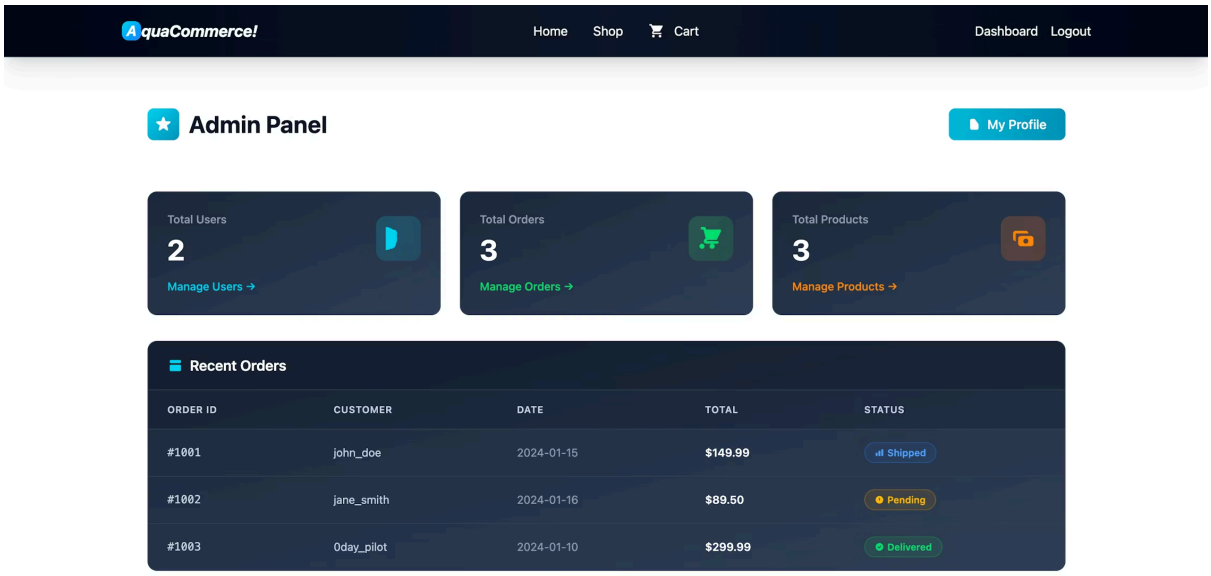


Exploiting JWT none algorithm attack in AquaCommerce!

Now that we've figured out a way to elevate our in-app privileges, we must look into the remote code execution vulnerability, as that'll be required to locate the flag on the server's file system.

Exploring the admin panel

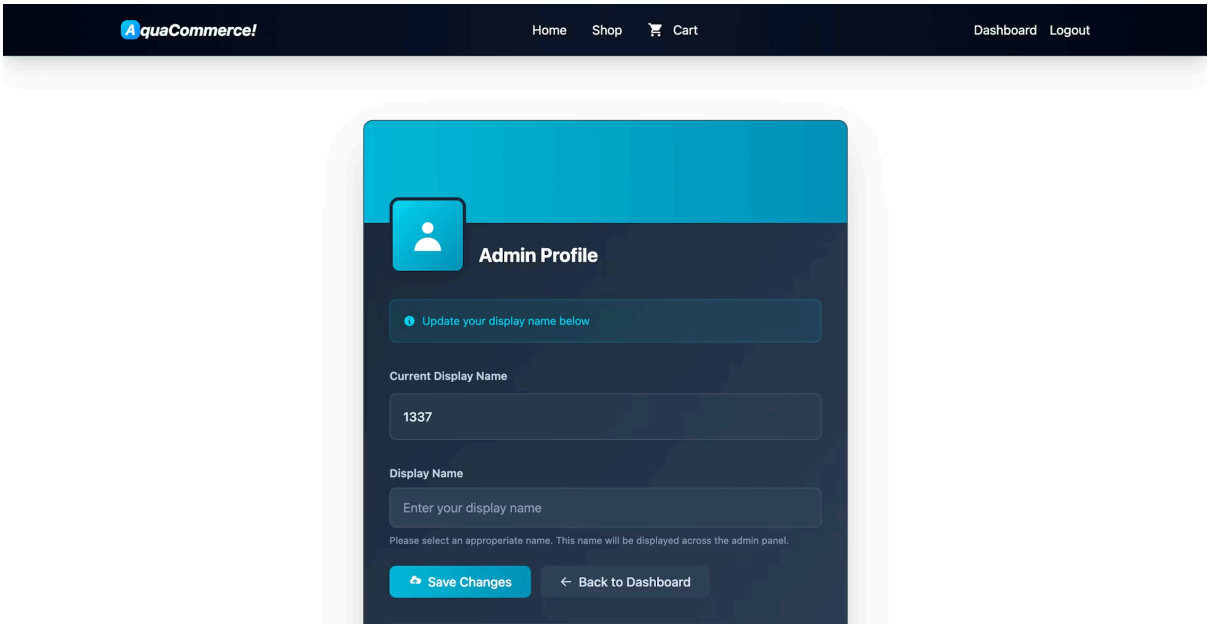
Our last step revealed a new admin dashboard that we now have access to. Clicking through to this panel shows a management & statistics view. It also appears to include a table with recent order data, which may indicate that the checkout process does work and possibly susceptible to SQL injection attacks.



AquaCommerce! admin panel

As expected from a CTF challenge, we'll need to click through all the links and look for interesting areas. In this instance, all the actions were disabled, and the data seems to be dummy data, as our newly created account did not appear in the user management list.

But there was one more button we hadn't clicked yet: "My Profile" in the top right corner of the admin panel. As we know, input fields could lead to all sorts of injection vulnerabilities, including SQLis, XSS, SSRFs and even SSTIs.



AquaCommerce! admin profile

The display name field was editable, and the backend seems to have handled our request as the page was rendering our display name back to us. In this case, we noticed that the input field was indeed vulnerable to server-side template injection.

We won't be able to go over the entire input testing process, but if you wish to learn more about the testing process for injection vulnerabilities, we recommend you have a look at our web hacking blog.

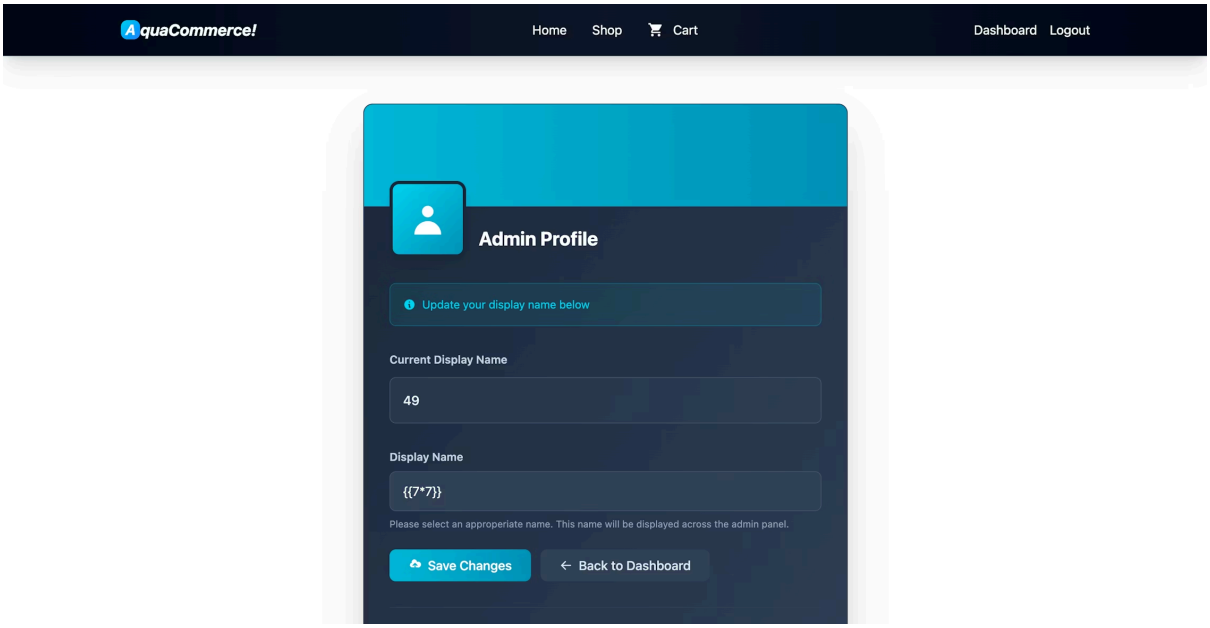
Testing for SSTI

Server-side template injection vulnerabilities occur when user input is embedded into a template engine without proper sanitization. From before, we identified that a Python framework is used for the web app. Django and Flask both support Jinja2 as a template engine, which has a specific syntax: `{{ expression }}`.

To confirm the presence of this SSTI, we'll need to send a basic payload like the following:

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

Jinja2 will evaluate the contents, in this case a simple mathematical equation, and render the output in the HTTP response:



Server-side template injection (SSTI) in AquaCommerce!

As you can see, our profile display name field returns 49 instead of our payload, confirming the presence of a server-side template injection. But conducting simple mathematical equations wasn't our goal. We needed to achieve arbitrary code execution to read the flag file, which is likely located on the server side.

From SSTI to RCE

Jinja2 SSTI exploitation typically follows a pattern of traversing Python's object hierarchy to access dangerous functions. There are many ways to achieve RCE in Jinja2, but we needed to find one that worked in this specific environment.

When all dangerous functions are disabled, we'll need to be on the lookout for gadgets that we can take advantage of. If you wish to dive deeper into exploiting SSTI vulnerabilities, we recommend you give our comprehensive article a read.

While looking for possible payloads online, we came across one that actively used the **config** object, which is often available in Flask templates:

```
{{ config.__class__.__init__.__globals__['os'].popen('whoami').read() }}
```

However, using this payload seemed to make the target application render our payload as a literal string instead of the evaluated output. This indicated that our input somehow got auto-escaped. In instances like these, we'd need to experiment and look for other possible techniques. For example, instead of relying on Flask's **config** object, we could traverse Python's class hierarchy directly.

A quick search online brings us to the following payload that actively leverages Python's **__subclasses__()** method to find classes with dangerous capabilities:

```
{% for x in ().__class__.__base__.__subclasses__() %}
  {% if "warning" in x.__name__ %}
    {{x()._module.__builtins__["__import__"]("os").popen('whoami').read()}}
  {% endif %}
{% endfor %}
```

Let's break this payload down:

1. First, it gets all subclasses of object (the base class of everything in Python) and loops through them.
2. Next, it filters for classes with "warning" in their name (like warnings.WarningMessage), and most dangerous functions will be equipped with this property.
3. Afterward, we try to use the dangerous class to access `__builtins__` to ultimately import the os module.
4. Finally, we use that imported package to execute arbitrary commands via the os.popen() method.

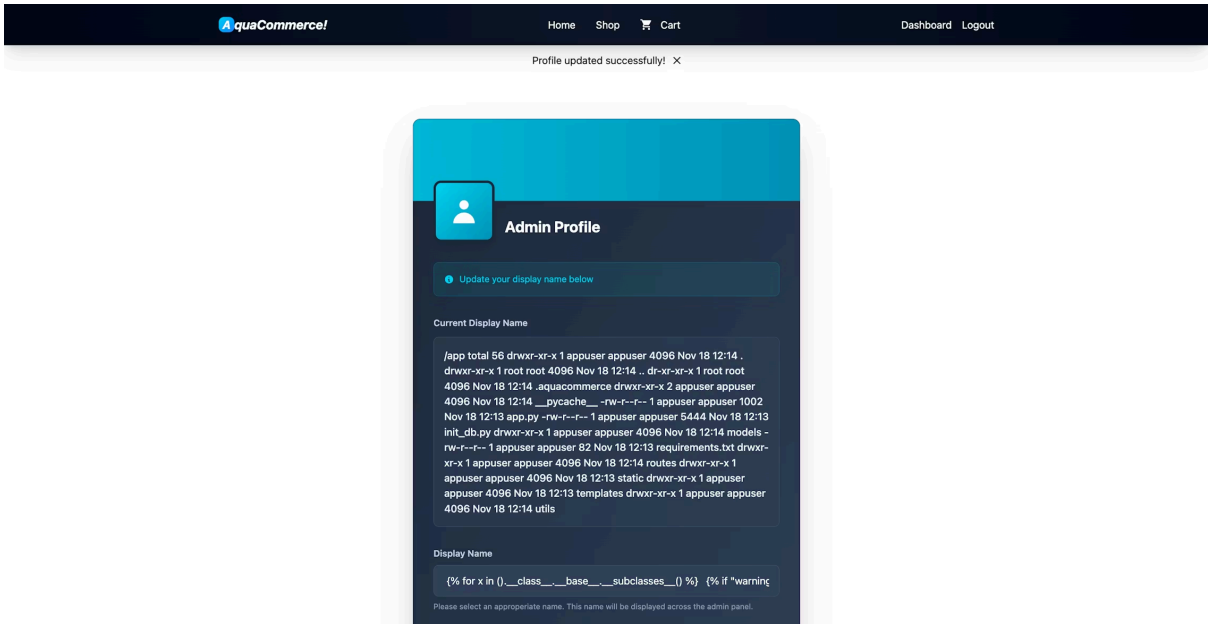
The `.read()` method is a helpful function to help us display the output on the page. If this were a blind SSTI vulnerability, we'd have to rely on other approaches, such as exfiltrating data through an out-of-band payload.

Capturing the flag

We successfully demonstrated arbitrary code execution on the system. This challenge requires us to submit a flag in the following format: `INTIGRITI{.*}`.

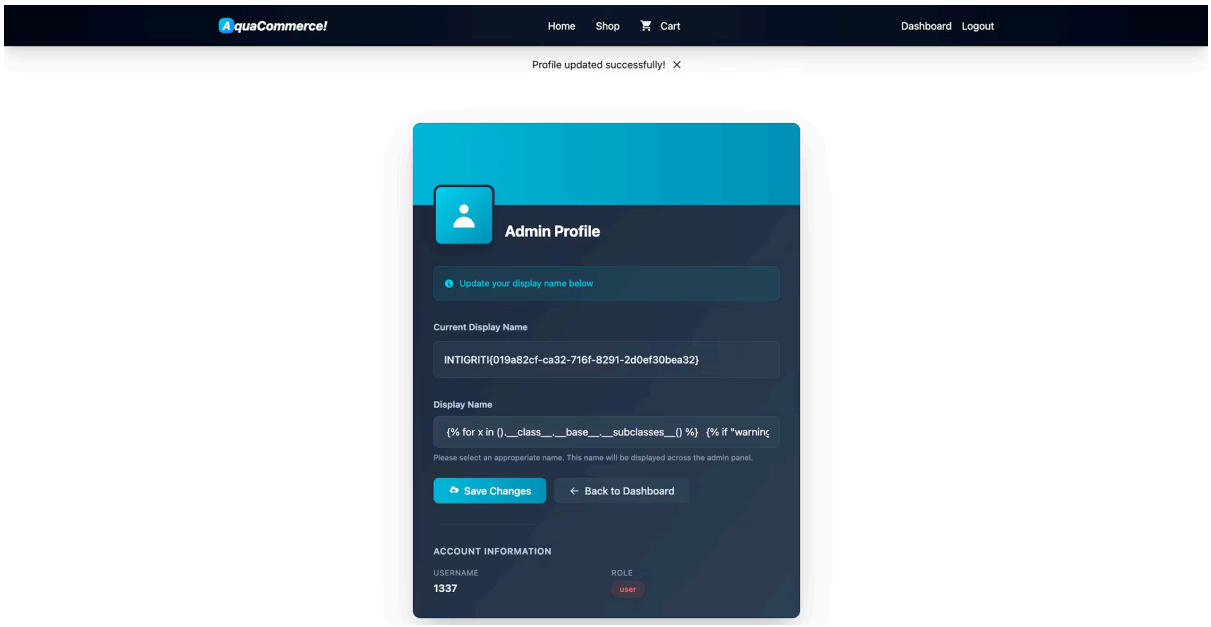
Similar to what we'd do in any other CTF, the first thing after achieving RCE was to understand our execution context. We knew a flag file existed somewhere on the filesystem, but had no specific path information.

Executing `whoami` revealed that we're running as a non-privileged (root) user, which is actually a best practice. Next up is to look at where we are and what's located in our current working directory. The following 2 commands will help us get that information:



AquaCommerce! Remote Code Execution (RCE)

In this instance, we didn't have to look far, as it seems that there's a hidden directory called `.aquacommerce` in the `/app` folder. Hidden directories are common for storing configuration files, secrets, and, in this case, the flag we need to solve this challenge.



Capturing the flag in AquaCommerce! via SSTI

The contents of the text file confirm that we've captured the flag and thereby solved November's Intigriti challenge!

Conclusion

AquaCommerce! demonstrated how seemingly isolated vulnerabilities can be chained together to achieve full system compromise. The JWT bypass gave us access to the admin panel, and the SSTI vulnerability gave us code execution.

We hope you enjoyed this challenge! Make sure to leave a follow on our official Twitter/X account to be notified when the next challenge drops. If you solved it using a different approach, we'd love to hear about it in our Discord community and our challenge post.

AUTHOR

Ayoub

Senior security content developer