Intigriti 0126 CTF Challenge: Exploiting insecure postMessage handlers

BY AYOUB · JANUARY 28, 2026

At Intigriti, we host monthly web-based Capture The Flag (CTF) challenges as a way to engage with the security researcher community. January's challenge presented participants with CRYPTIGRITI, a cryptocurrency trading platform where users could buy and trade Bitcoin (BTC), Monero (XMR), and a custom digital currency, 1337COIN.

This article provides a step-by-step walkthrough for solving [January's CTF challenge] while demonstrating techniques for exploiting insecure postMessage implementations in real-world web applications.

Let's dive in!

Challenge overview

CRYPTIGRITI presents itself as a modern DeFi (decentralized finance) platform with a sleek interface similar to almost all other popular cryptocurrency exchanges. Looking at the challenge rules, we can read the following:

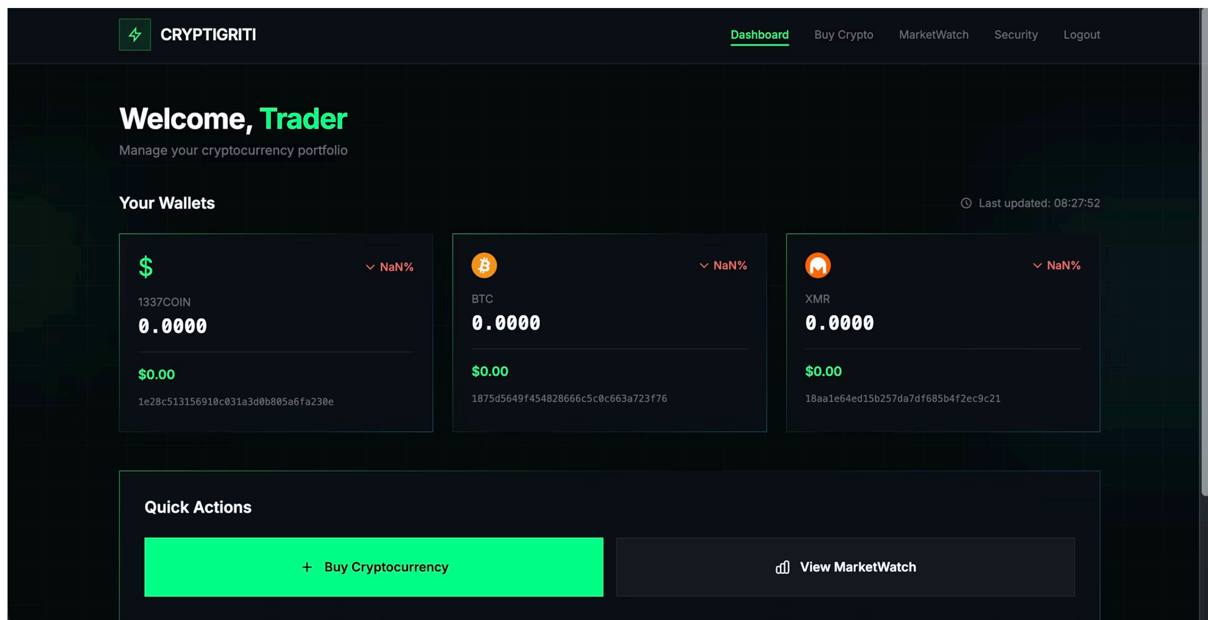
- We must find a flag in the following format: `INTIGRITI{.*}`
- The correct solution should leverage a client-side vulnerability
- Self-XSS or MiTM attacks are not allowed
- The attack should not require more than a single click (submitting a URL)

The challenge description hinted that we'd need to transfer a 1337COIN from an admin account to capture the flag. Unlike previous challenges where we had direct access to source code, this time we'd be working mostly with code that's available to us on the client-side, relying on reconnaissance and our understanding of common web vulnerabilities.

Initial reconnaissance

As usual, we'll need to map out any possible information that's related to our target application. With tools like Wappalyzer and BuiltWith, we can easily determine that the backend is in Node.js. The response headers also reveal that the application is running on ExpressJS, a minimal Node.js web app framework.

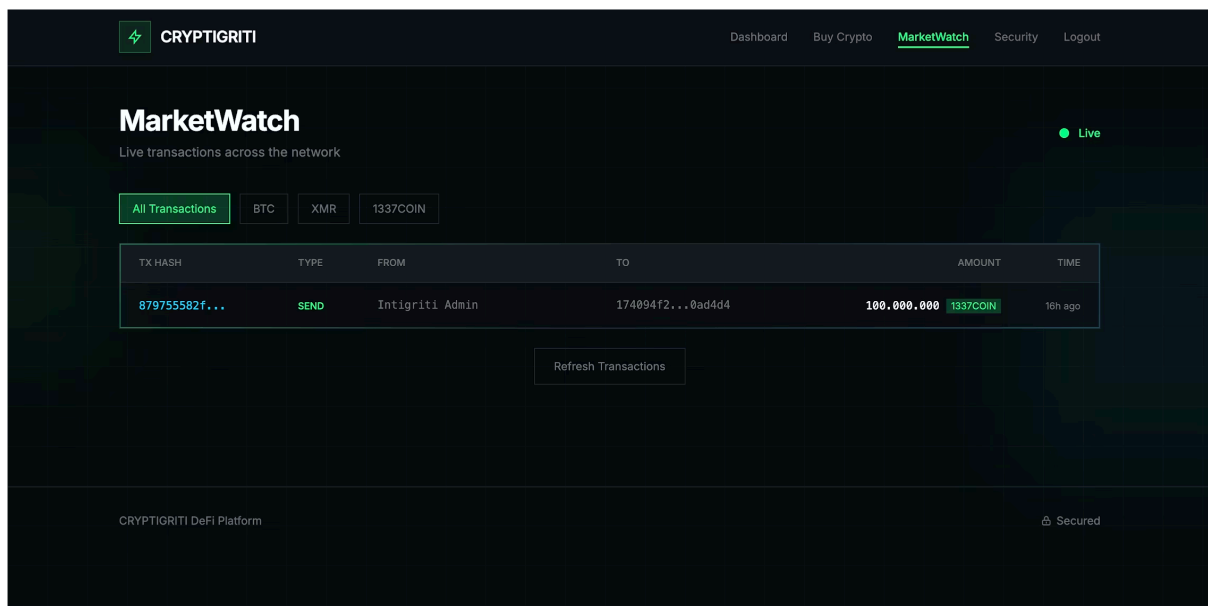
After creating a new account, we're greeted with a modern dashboard showing three cryptocurrency wallets: Bitcoin (BTC), Monero (XMR), and 1337COIN. Each wallet had a unique address generated for our account, but all balances stood at $0.00 USD.



CRYPTIGRITI dashboard

The platform provided several features that caught our attention. The most notable features include purchasing functionality to buy BTC or XMR, a public MarketWatch feed displaying all recent transactions, and a security reporting page where security researchers can submit proof of concept URLs for the security team to review.

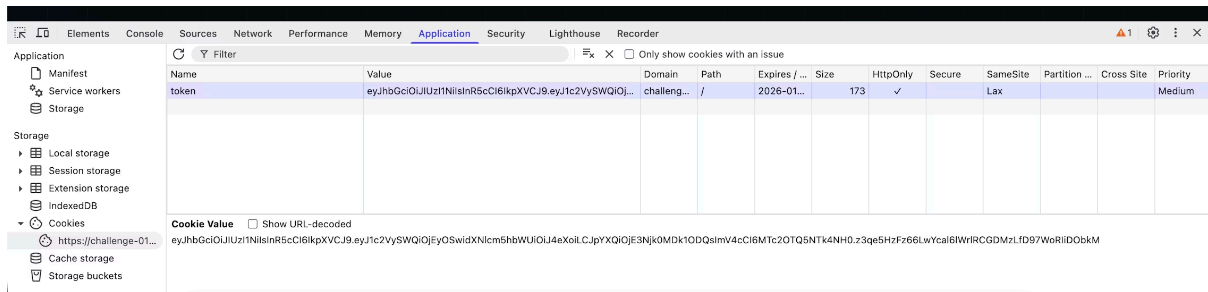
The MarketWatch feature immediately stood out. Scrolling through the feed, we noticed an initial transaction from "Intigriti Admin" sending 100,000,000 1337COIN to their own address. This confirmed that a privileged admin account existed and held the valuable currency we needed to capture our flag.



CRYPTIGRITI MarketWatch

Understanding authentication

Looking at our browser's developer tools, we could see that authentication was handled through JWT tokens stored in cookies. Examining the session cookie revealed some interesting properties:



CRYPTIGRITI cookie policy

The **HttpOnly: true** setting meant JavaScript couldn't access the cookie directly, which is a common security practice to limit client-side attacks, such as XSS. The **SameSite: Lax** policy would prevent the cookie from being sent in most cross-site requests, though it would still be sent for top-level navigations. We took note of these properties, as they could potentially become more important later on as we're tasked with capturing the flag using a client-side vulnerability.

From experience, we also know that some JWT implementations are flawed and can allow for [authentication bypasses](), injection attacks, and more. For this reason, we also quickly tested for common [JWT vulnerabilities](), including algorithm confusion, signature bypass, and weak secrets, but the authentication implementation seemed solid enough. We'd need to find another attack vector.

Understanding the transaction flow

Next up was targeting the purchase functionality. It worked as expected, selecting a cryptocurrency and entering an amount took us through a standard checkout flow. The page displayed a transaction summary and allowed users to confirm their purchase. While exploring this functionality, we noticed something interesting when examining the page source in the browser's console.

Discovering the postMessage vulnerability

Digging through the checkout page's JavaScript, we found a postMessage event listener:

```
window.addEventListener('message', async function(event) {
    if (event.data.type === 'submitTransaction') {
        const transactionData = event.data.transaction;

        const response = await fetch('/api/transaction', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({
                toAddress: transactionData.toAddress,
                currency: transactionData.currency,
                amount: transactionData.amount
            })
        });

        if (response.ok) {
            showSuccess('Transaction completed!');
        }
    }
});
```

Web messages are widely used within web applications to transfer data between multiple windows, without dealing with restrictions like CORS or Same-Origin Policy (SOP).

In this instance, the parent document had a web message listener that accepted messages from any origin without validating the `event.origin`, and it directly processed transaction data without any additional checks before passing it to the API. This meant that if we could trick any victim into visiting our specially crafted proof of concept page, we would essentially be able to send a malicious postMessage to execute an unauthorized transaction on behalf of the victim.

As referenced earlier, our primary goal is to transfer 1337COIN to our wallet. But since we do not have any funds in our account, we cannot purchase or trade any. This means that we'll have to trick the admin into sending us the funds to our account, as that's the only known wallet on the network that seems to hold 1337COIN.

Crafting the proof of concept

Our attack path was straightforward. We'd create a malicious proof of concept that opens the checkout page and sends a crafted web message with malicious transaction data. The checkout page would listen for incoming web messages and process the transaction by making the API call on behalf of the admin as long as we're on the same host. Ultimately, leading to the transfer of 1337COIN to our wallet.

Dealing with SameSite

Initially, we attempted to open the checkout page on `https://challenge-0126.intigriti.io/checkout.html?currency=BTC&price=1`, however, we later confirmed that this was not possible since the bot visiting our proof of concept URL wasn't authenticated on that same host. This meant that the cookies would've never been forwarded under the current configuration. To resolve this issue, we had to adjust our proof of concept to use the same host as the bot.

Thanks to our recon work from before, we understood that ExpressJS was used. This setup often involves deploying a reverse proxy (such as Nginx) in front of the application. This also means that the same

application is accessible internally via the loopback host (localhost), commonly via port `80`, `3000`, or `8080`. After a quick guess, we found that the same application was indeed running locally on `localhost:8080`.

Difference between a forward proxy vs reverse proxy server

Another issue we encountered during the testing was that the web message was sent too early. Even before the checkout page had time to initialize. To account for this, we added a small time delay to allow the full page to load first, including any background requests. Afterward, we sent our web message to transfer the funds.

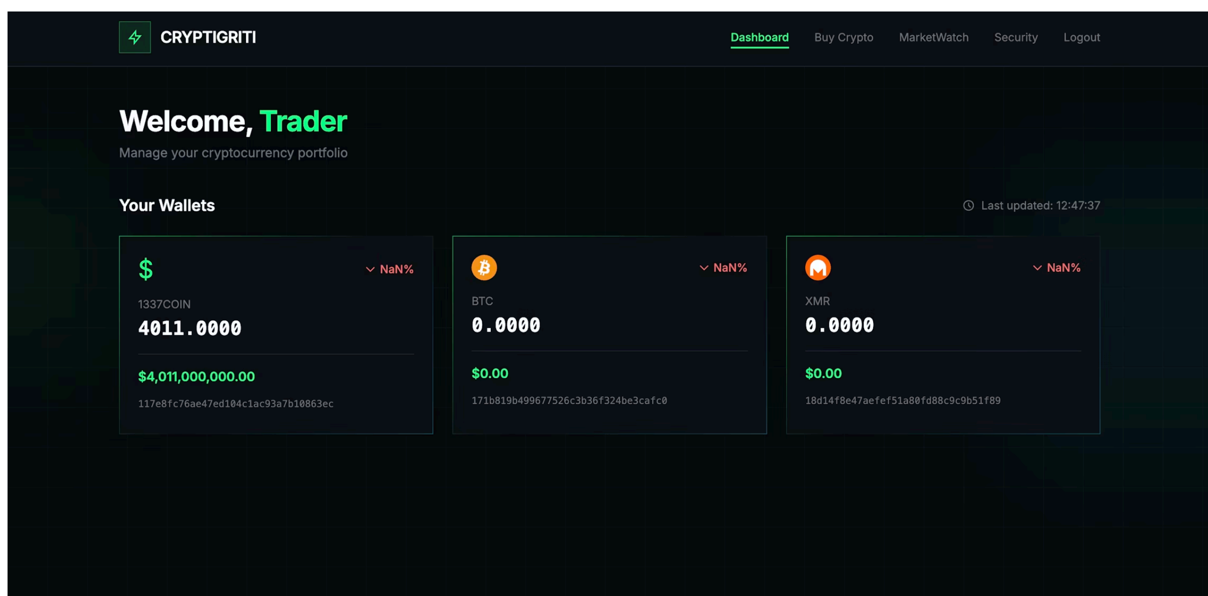
Here's the full proof of concept that would have drained the administrator's wallet:

```
<script>
  const ATTACKER_WALLET = '1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6p7q';
  const AMOUNT = 1337;

  // Open checkout page
  const target = window.open(
   'http://localhost:8080/checkout.html?currency=BTC&price=45000',
   '_blank'
  );

  // Leave enough time for background requests to finish first, then exploit
  setTimeout(() => {
   target.postMessage({
    type: 'submitTransaction',
    transaction: {
     toAddress: ATTACKER_WALLET,
     currency: '1337COIN',
     amount: AMOUNT
    }
   }, '*');
  }, 10000);
</script>
```

Our final step was to host this payload in a location accessible to the host. Fortunately, there are numerous online services that allow us to do exactly this.

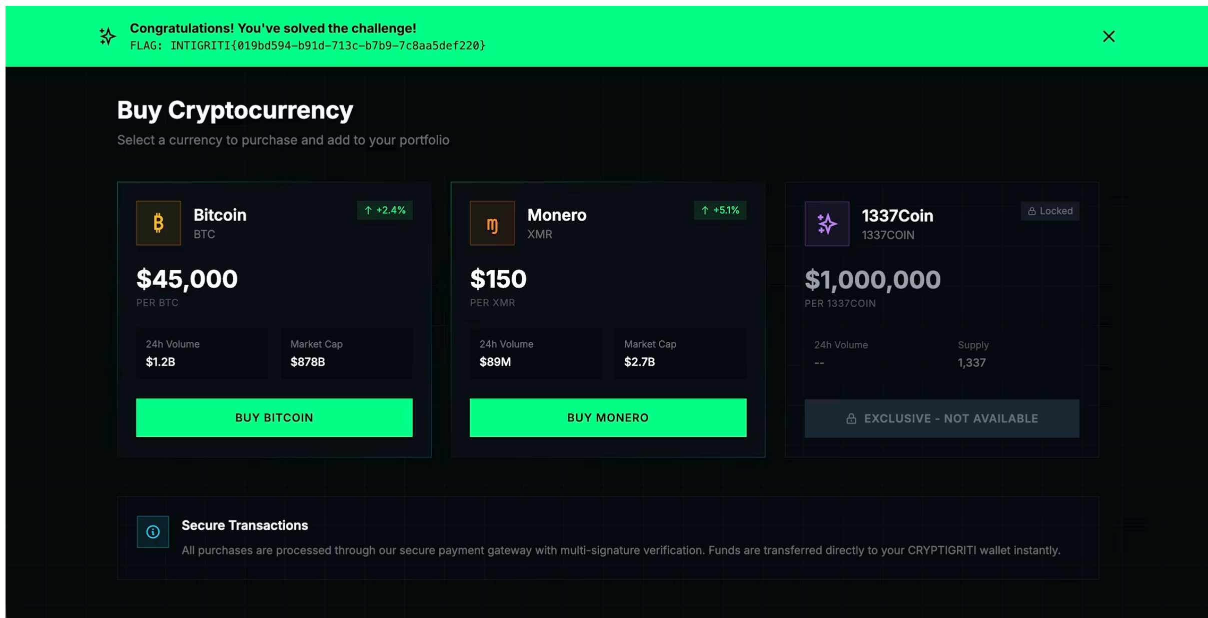
Capturing the flag

All we had to do now to capture our flag was to instruct the security bot to visit our malicious page. 30 seconds after submission, we refreshed our dashboard. Our 1337COIN balance had changed from $0.00 to $4,011,000,000.00 USD (4,011 1337COIN), confirming that our exploit had worked:



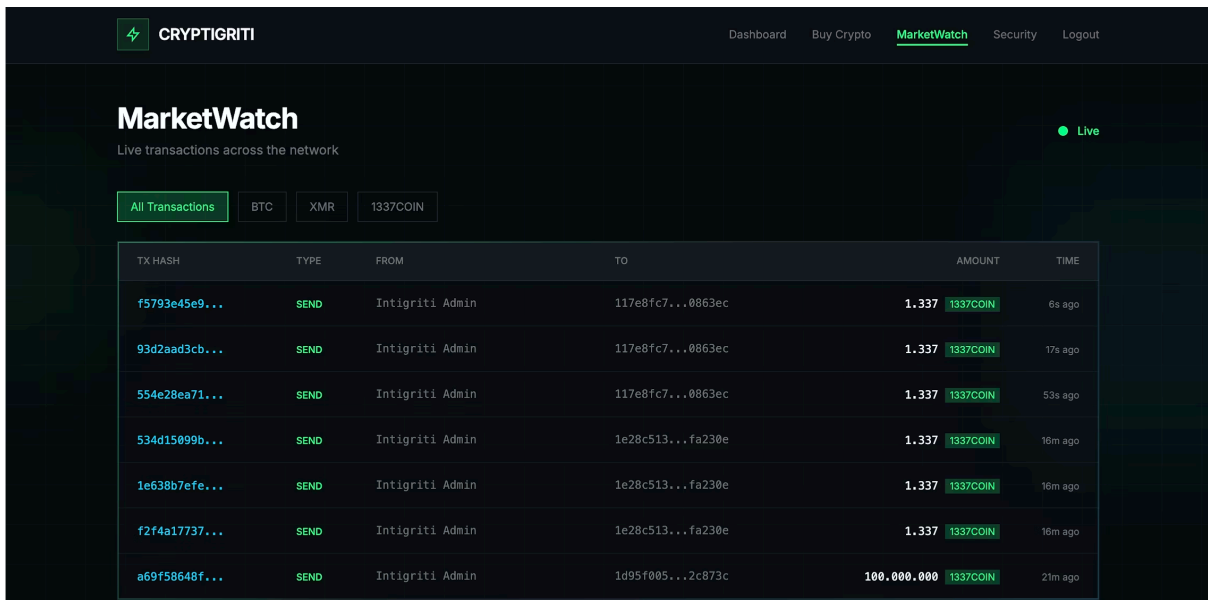
Updated balance confirms the postMessage exploitation

More importantly, a new banner appeared at the top of the page, returning the flag:



Capturing the flag for Intigriti 0126 CTF

Checking MarketWatch confirmed that our transaction had gone through, with "Intigriti Admin" as the sender and our wallet address as the recipient!



Draining victim's wallet via postMessage exploitation

Conclusion

Intigriti 0126 CRYPTIGRITI challenge demonstrated a vulnerability that remains surprisingly common in modern web applications: insecure implementation of postMessage handlers. The lack of origin validation on the checkout page's message listener allowed us to forge transactions on behalf of any authenticated user. Which, in real-world scenarios, would've introduced devastating financial impact. It also highlights the importance of testing for client-side web vulnerabilities and how weaponizing them correctly can lead to a critical impact.

We hope you enjoyed this month's challenge! Make sure to follow our official Twitter/X account to stay on top when the next challenge releases. If you solved it using a different approach, we'd love to hear about it in our Discord community.

AUTHOR

Ayoub

Senior security content developer

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com