SCENARIO

The application contains a DOM based client-side vulnerability which web messaging and parses the message as JSON. We’ll try to exploit this vulnerability by forcing the user to inject a cookie that will cause XSS on the page and call the print() function.

**PROCEDURE**

1. Access the application to inspect its source code.
2. Upon inspecting we see that there is an event listener active expects a string that is parsed using JSON.parse(). In the JavaScript, we can see that the event listener expects a type property and that the load-channel case of the switch statement changes the iframe src attribute:
3. window.addEventListener('message', function(e) {

var iframe = document.createElement('iframe'), ACMEplayer = {element: iframe}, d;

document.body.appendChild(iframe);

try {

d = JSON.parse(e.data);

} catch(e) {

return;

}

switch(d.type) {

case "page-load":

ACMEplayer.element.scrollIntoView();

break;

case "load-channel":

ACMEplayer.element.src = d.url;

break;

case "player-height-changed":

ACMEplayer.element.style.width = d.width + "px";

ACMEplayer.element.style.height = d.height + "px";

break;

}

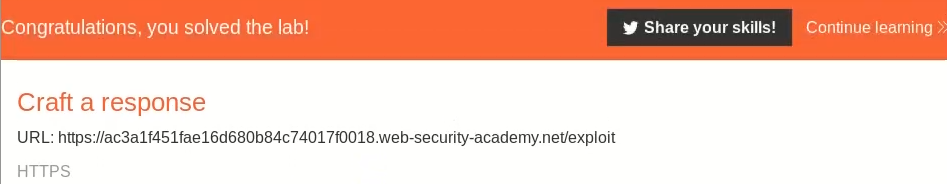
}, false);

1. When the iframe we constructed loads, the postMessage() method sends a web message to the home page with the type load-channel. The event listener receives the message and parses it using JSON.parse() before sending it to the switch. The switch triggers the load-channel case, which assigns the url property of the message to the src attribute of the ACMEplayer.element iframe. However, in this case, the url property of the message actually contains our JavaScript payload. As the second argument specifies that any targetOrigin is allowed for the web message, and the event handler does not contain any form of origin check, the payload is set as the src of the ACMEplayer.element iframe. The print() function is called when the victim loads the page in their browser..
2. So, according to that we will create a payload and inject it into the application using our exploit server to deliver it to the victim.

**PAYLOAD**

<iframe src=https://YOUR-LAB-ID.web-security-academy.net/ onload='this.contentWindow.postMessage("{\"type\":\"load-channel\",\"url\":\"javascript:print()\"}","\*")'>

**PROOF OF CONCEPT**

****

**REMEDIATION**

1. **Verify Message Origin:** Always validate the origin of a message received through the postMessage() method. An origin check ensures that messages are only accepted from trusted domains. In the event handler, you can check e.origin against a whitelist of trusted domains.
2. **Strictly Parse JSON and Validate Data:** While the application is parsing the JSON correctly (i.e., it's catching errors from maliciously formatted JSON), the actual content of the parsed JSON needs to be validated. Especially for values that are going to be used in sensitive operations like changing an iframe's src, ensure that the content is a trusted and safe URL.
3. **Avoid Direct Assignments:** Avoid directly assigning untrusted values from web messages to DOM elements' attributes or properties. Always sanitize and validate data before making such assignments.
4. **Use Allowlists:** Define strict rules and allowlists for what type of content can be processed. For instance, in the load-channel case, only allow specific URLs to be loaded into the iframe.
5. **Sandbox Iframes:** If you're using iframes to load external content, consider using the sandbox attribute on the iframe to restrict what the iframe can do. This can prevent scripts from running inside the iframe, among other restrictions.