Portswigger DOM-based Vulnerabilities Lab Notes

1. DOM XSS using web messages

This lab demonstrates a simple web message vulnerability. To solve this lab, use the exploit server to post a message to the target site that causes the print() function to be called.

 Solution

1. Notice that the home page contains an addEventListener() call that listens for a web message.
2. Go to the exploit server and add the following iframe to the body. Remember to add your own lab ID:

***<iframe src="https://YOUR-LAB-ID.web-security-academy.net/" onload="this.contentWindow.postMessage('<img src=1 onerror=print()>','\*')">***

1. Store the exploit and deliver it to the victim.

* When the iframe loads, the postMessage() method sends a web message to the home page.
* The event listener, which is intended to serve ads, takes the content of the web message and inserts it into the div with the ID ads.
* However, in this case it inserts our img tag, which contains an invalid src attribute.
* This throws an error, which causes the onerror event handler to execute our payload.

1. DOM XSS using web messages and a JavaScript URL

This lab demonstrates a DOM-based redirection vulnerability that is triggered by web messaging. To solve this lab, construct an HTML page on the exploit server that exploits this vulnerability and calls the print() function.

 Solution

1. Notice that the home page contains an addEventListener() call that listens for a web message. The JavaScript contains a flawed indexOf() check that looks for the strings "http:" or "https:" anywhere within the web message. It also contains the sink location.href.
2. Go to the exploit server and add the following iframe to the body, remembering to replace YOUR-LAB-ID with your lab ID:

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

🛠 How it works

iframe loads the vulnerable page (src="...lab-id...").

Once loaded, the iframe’s onload fires → it calls postMessage(...).

The payload:

Starts with javascript:, so when assigned to location.href, it executes JavaScript.

Ends with //http: so that the site’s weak filter sees "http:" and allows it.

The page’s addEventListener("message", ...) receives this string, passes the check, and assigns it to location.href.

Result: browser runs print(), solving the lab.

1. Store the exploit and deliver it to the victim.

* This script sends a web message containing an arbitrary JavaScript payload, along with the string "http:".
* The second argument specifies that any targetOrigin is allowed for the web message.
* When the iframe loads, the postMessage() method sends the JavaScript payload to the main page.
* The event listener spots the "http:" string and proceeds to send the payload to the location.href sink, where the print() function is called.

1. DOM XSS using web messages and JSON.parse

This lab uses web messaging and parses the message as JSON. To solve the lab, construct an HTML page on the exploit server that exploits this vulnerability and calls the print() function.

 Solution

1. Notice that the home page contains an event listener that listens for a web message. This event listener 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.

**What the vulnerable site does**

* The page has a message event listener (addEventListener("message", ...)).
* It expects incoming messages in JSON format, e.g.:
* {"type": "load-channel", "url": "SOME\_URL"}
* If the message has type: "load-channel", the script will **change an iframe’s src attribute** to whatever is inside "url".

Why this is dangerous

* Because we control the url value, we can point it not to a real URL but to a JavaScript URL like javascript:print().  
  When the iframe src becomes javascript:print(), the browser executes the code.

1. Go to the exploit server and add the following iframe to the body, remembering to replace YOUR-LAB-ID with your lab ID:

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

 The iframe loads the vulnerable page.

 Once loaded, it sends a crafted message with postMessage().

 The message says:

* type: "load-channel" → triggers the iframe update.
* url: "javascript:print()" → becomes the iframe’s src.

 Browser executes print() → proving XSS.

1. Store the exploit and deliver it to the victim.

* 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.

1. DOM-based open redirection

This lab contains a DOM-based open-redirection vulnerability. To solve this lab, exploit this vulnerability and redirect the victim to the exploit server.

 Solution

The blog post page contains the following link, which returns to the home page of the blog:

***<a href='#' onclick='returnURL = /url=https?:\/\/.+/.exec(location);***

***if(returnUrl) location.href = returnUrl[1];***

***else location.href = "/"'>Back to Blog</a>***

What happens here?

1. href='#'
   * Means the link itself doesn’t actually go anywhere.
   * Instead, the onclick JavaScript decides the destination.
2. /url=https?:\/\/.+/.exec(location)
   * This is a regular expression (regex).
   * It looks at the current page URL (location).
   * It tries to find something like:

url=http://SOMETHING

or

url=https://SOMETHING

* + If found, it saves the match in returnURL.

👉 Example:  
If the page URL is

https://lab.web-security-academy.net/post?postId=4&url=https://evil.com

Then returnURL[1] will be https://evil.com.

1. if(returnUrl) location.href = returnUrl[1]; else location.href = "/"
   * If the regex found a url=..., then redirect the user’s browser (location.href) to that URL.
   * If not, just go back to / (the homepage).

Why is this vulnerable?

* The script blindly trusts whatever is after url=.
* So if we add &url=https://evil.com, clicking “Back to Blog” will take the victim to our malicious site instead of the real blog.

The url parameter contains an open redirection vulnerability that allows you to change where the "Back to Blog" link takes the user. To solve the lab, construct and visit the following URL, remembering to change the URL to contain your lab ID and your exploit server ID:

***https://YOUR-LAB-ID.web-security-academy.net/post?postId=4&url=https://YOUR-EXPLOIT-SERVER-ID.exploit-server.net/***

1. DOM-based cookie manipulation

This lab demonstrates DOM-based client-side cookie manipulation. To solve this lab, inject a cookie that will cause XSS on a different page and call the print() function. You will need to use the exploit server to direct the victim to the correct pages.

 Solution

1. Notice that the home page uses a client-side cookie called lastViewedProduct, whose value is the URL of the last product page that the user visited.
2. Go to the exploit server and add the following iframe to the body, remembering to replace YOUR-LAB-ID with your lab ID:

***<iframe src="https://YOUR-LAB-ID.web-security-academy.net/product?productId=1&'><script>print()</script>"***

***onload="if(!window.x)this.src='https://YOUR-LAB-ID.web-security-academy.net';window.x=1;">***

***</iframe>***

 **The injection point**

* Because the cookie value is inserted into HTML without proper escaping, anything we put in the cookie can change the HTML structure.
* That’s where our payload goes.

 **Why the payload looks like this**

'><script>print()</script>

* The leading **'>** is crucial:
  + ’ closes any open string (like inside an href attribute).
  + > closes the current tag.
  + This “breaks out” of the HTML safely so that what follows is interpreted as new HTML, not just part of an attribute.
* Then comes <script>print()</script>, which is valid JavaScript that the browser will execute.

 **How the exploit server sets it up**

* You host an iframe pointing to a product page but with the malicious payload appended:

1. Store the exploit and deliver it to the victim.

* The original source of the iframe matches the URL of one of the product pages, except there is a JavaScript payload added to the end.
* When the iframe loads for the first time, the browser temporarily opens the malicious URL, which is then saved as the value of the lastViewedProduct cookie.
* The onload event handler ensures that the victim is then immediately redirected to the home page, unaware that this manipulation ever took place.
* While the victim's browser has the poisoned cookie saved, loading the home page will cause the payload to execute.