Lab 3 -- Andrew Stevenson

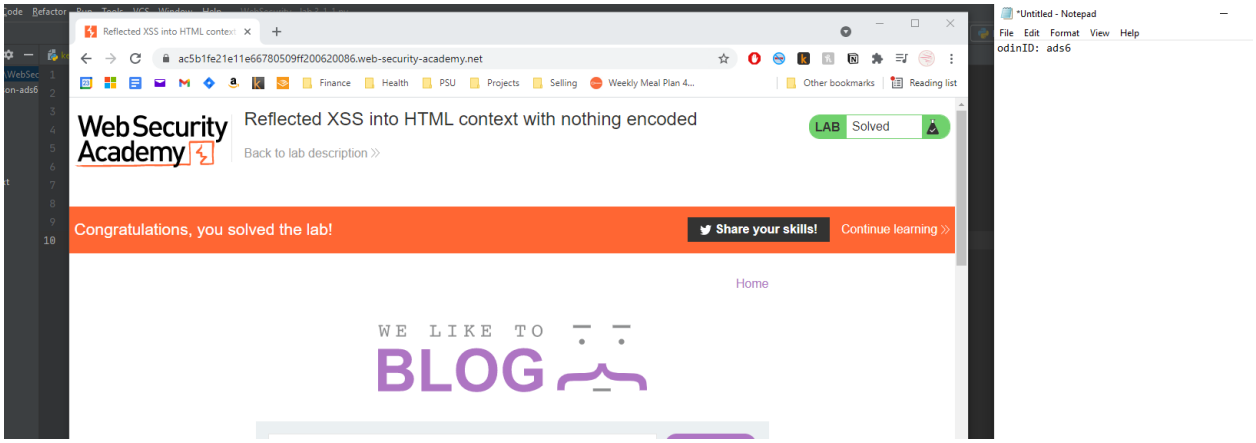
Lab 3.1

1. cross-site-scripting/reflected (1)

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: Unsanitized use of the query string within the HTML allows the adversary to inject arbitrary scripts into the page for the client to unwittingly execute.

Remediation:Sanitize the querystring by html encoding it before using it within the HTML.

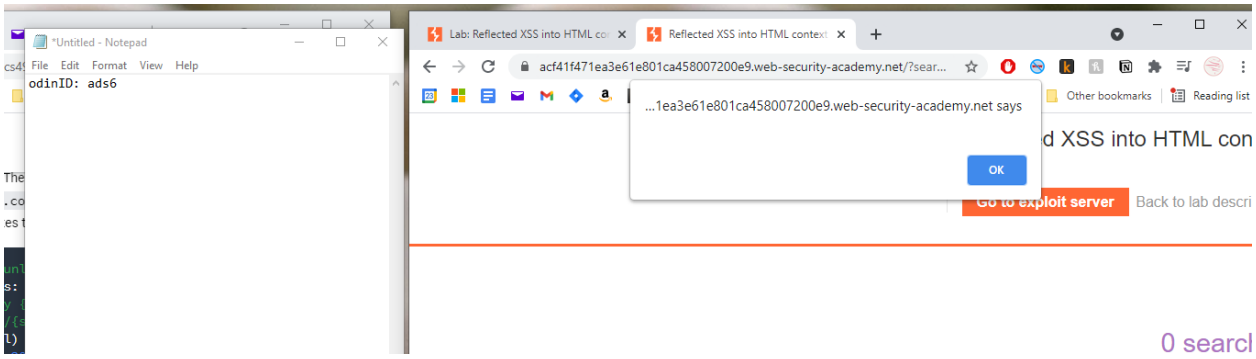


2. cross-site-scripting/reflected (2)

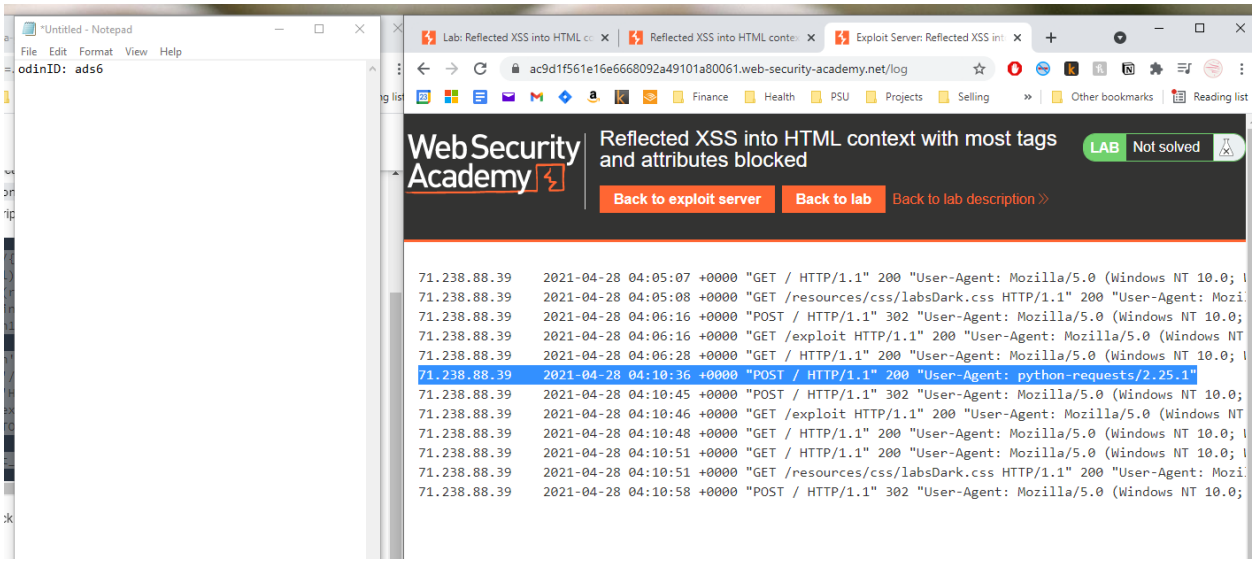
- List which window event attributes are allowed for your lab notebook

Onresize, onstorage

- Include a screenshot of the pop-up



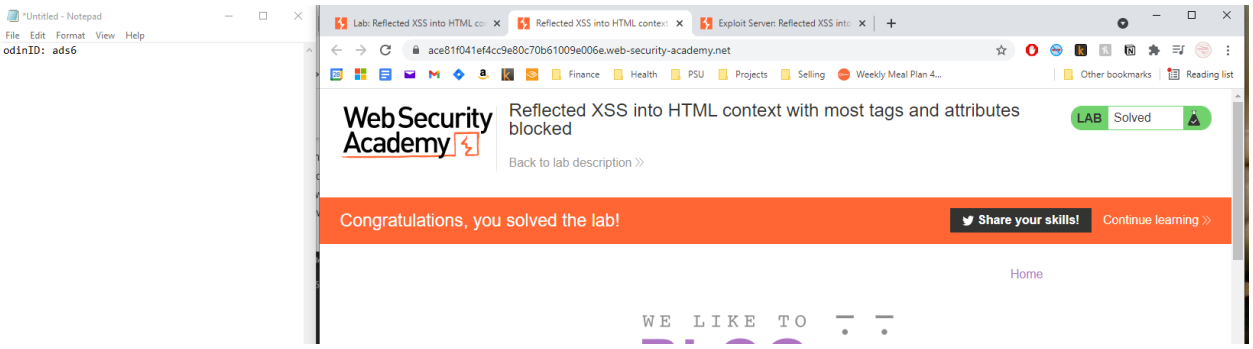
- Take a screenshot of the above log entry and include it in your lab notebook.



- Take a screenshot showing completion of the level that includes your OdinId

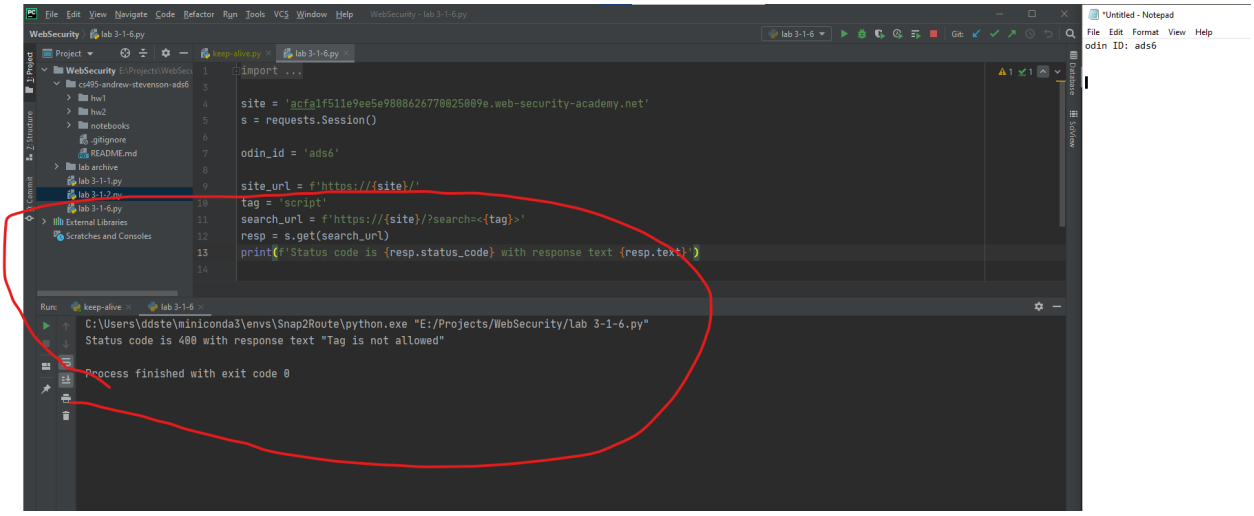
Vulnerability: Although the WAF blocks many scripting tags from being incorporated into the HTML, a few are allowed through, including the onresize= tag. Thus, the adversary may force execution of their attack page via an iframe inserted into the url that automatically resizes the client window, leading to execution of the onresize= payload.

Remediation: Properly html encode the query string before incorporating it into the HTML.

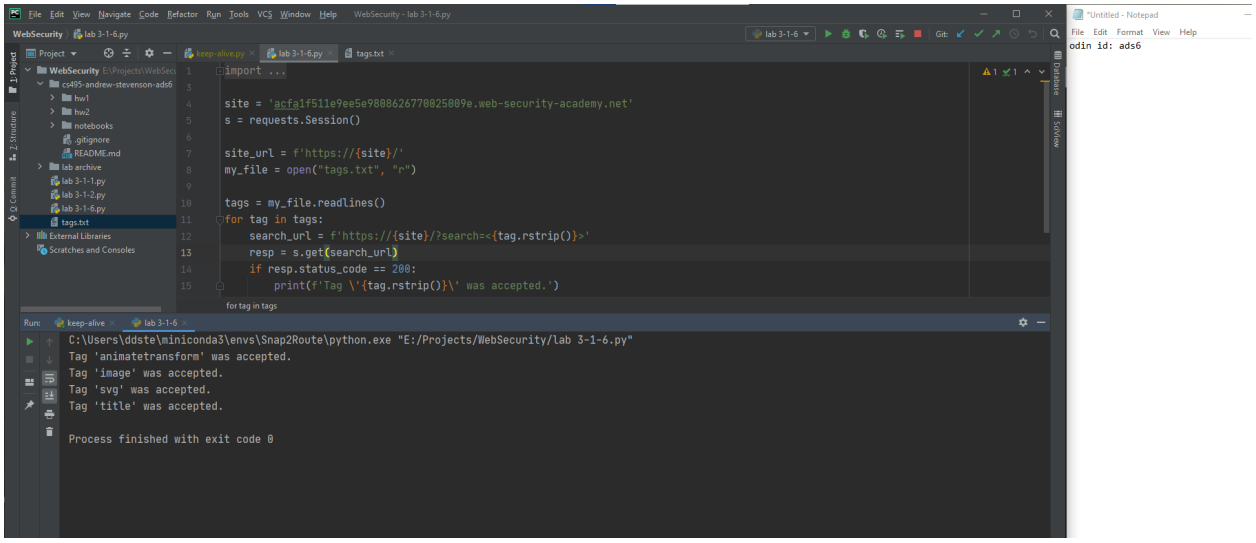


6. cross-site-scripting/reflected (3)

- Show the HTTP status code and response text obtained



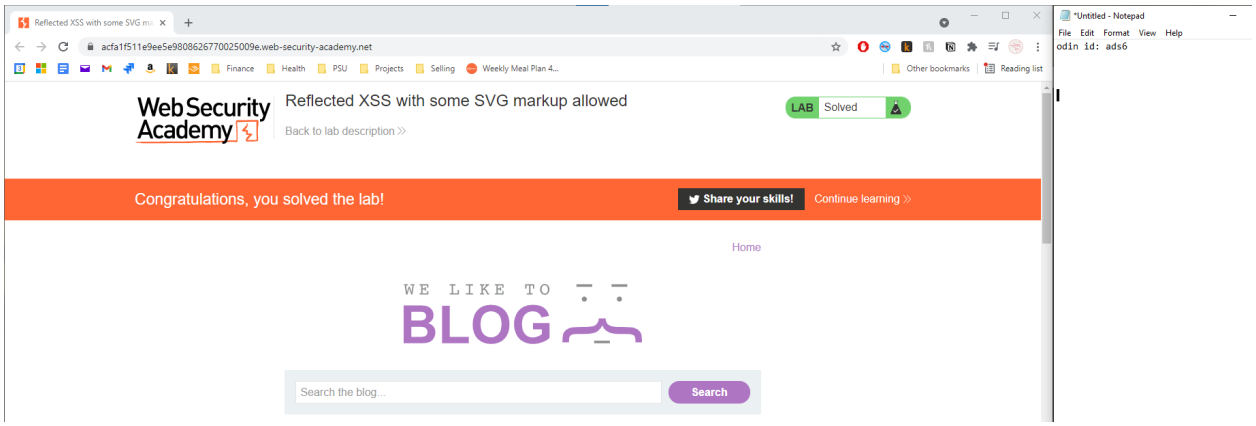
- Show the list of tags that are not filtered



- Take a screenshot showing completion of the level that includes your OdinId

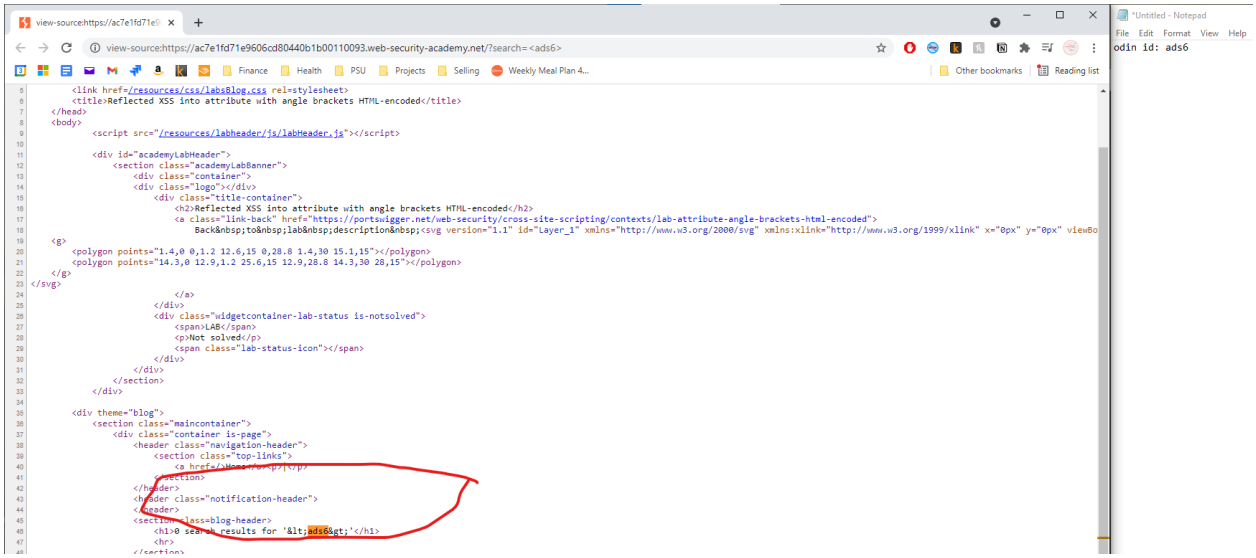
Vulnerability: Not all tags are being filtered out from the querystring that gets converted to HTML, allowing the adversary to run an XSS attack using the <svg> and <animatetransform> tags.

Remediation: Filter out all tags, or --better yet--html encode the querystring contents before converting them to html.

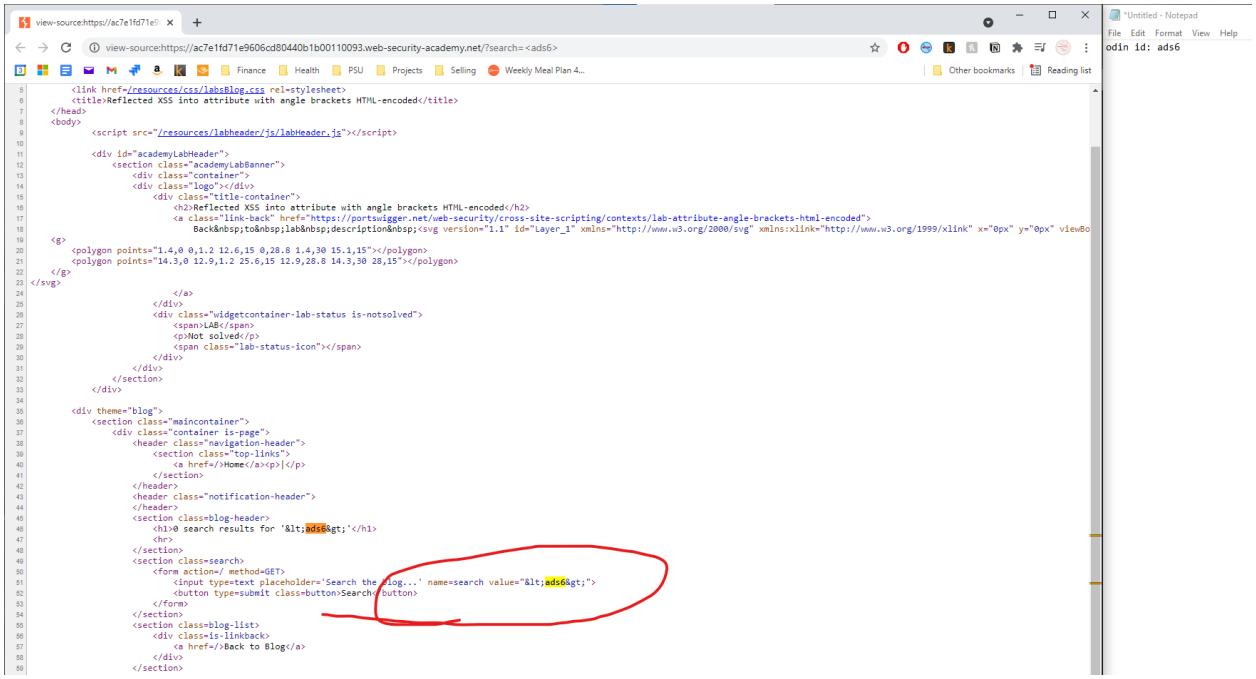


7. cross-site-scripting/reflected (4)

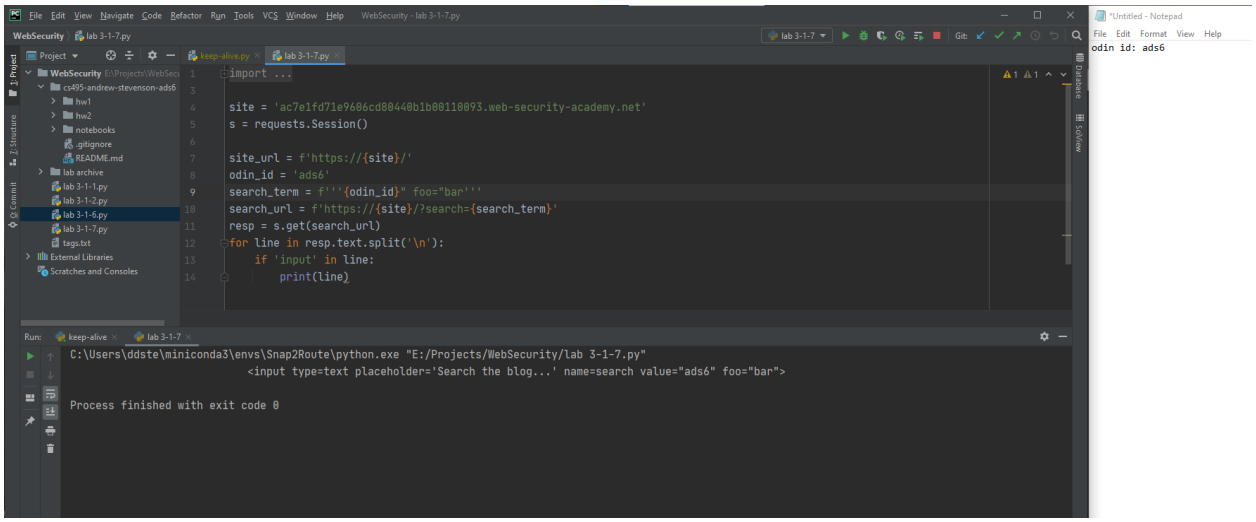
- Show the part of the source demonstrating that the content has been HTML-encoded in the HTML context



- Show the part of the source demonstrating that the reflected content also appears within an HTML tag's context



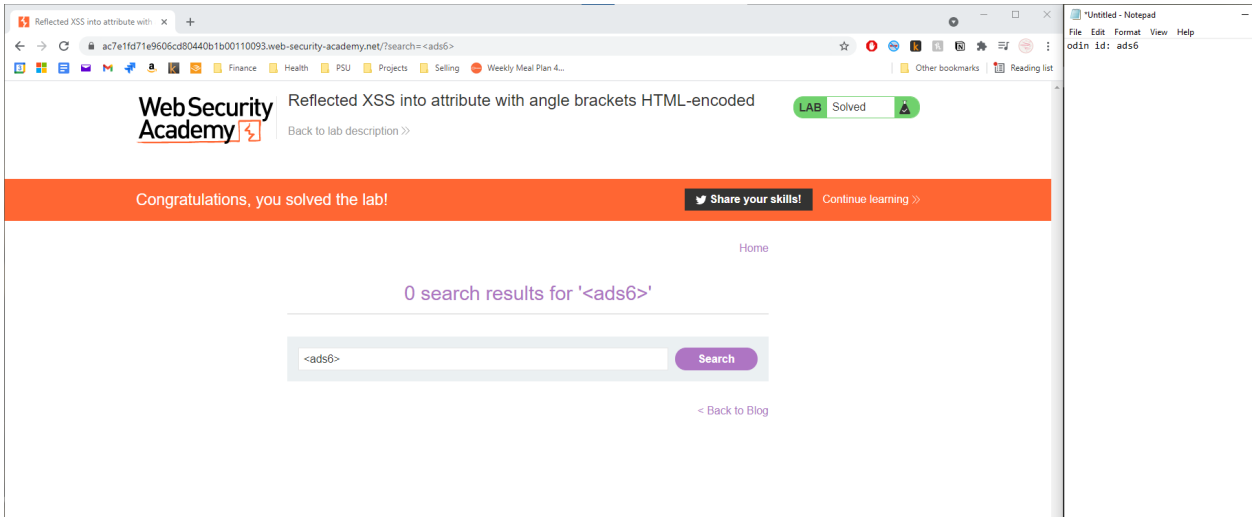
- Take a screenshot of the output for your lab notebook



- Take a screenshot showing completion of the level that includes your OdinId

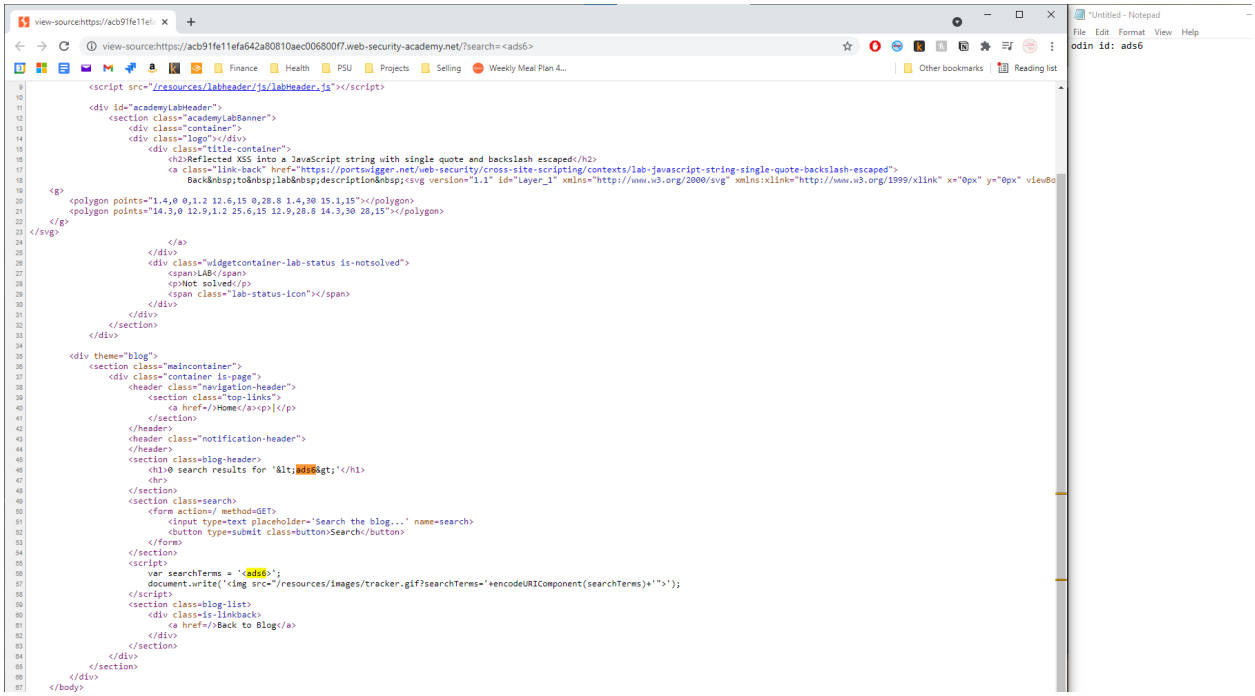
Vulnerability: The adversary may bypass the html tag encoding by including a close quote mark in the querystring, allowing them to add arbitrary attributes to the input tag including an onmouseover attribute that will execute a cross site attack.

Remediation: HTML encode single and double quotation marks too.

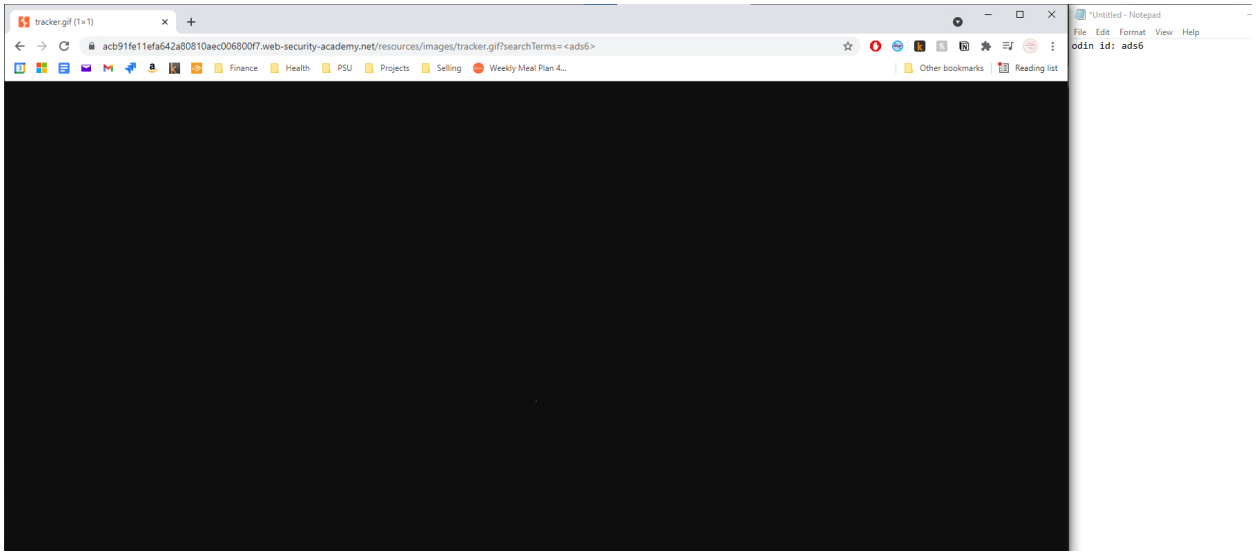


8. cross-site-scripting/reflected (5)

- Within the source, show the two contexts that the search term appears in.



- Open the tracker image in a new tab and show its URL that contains the search term

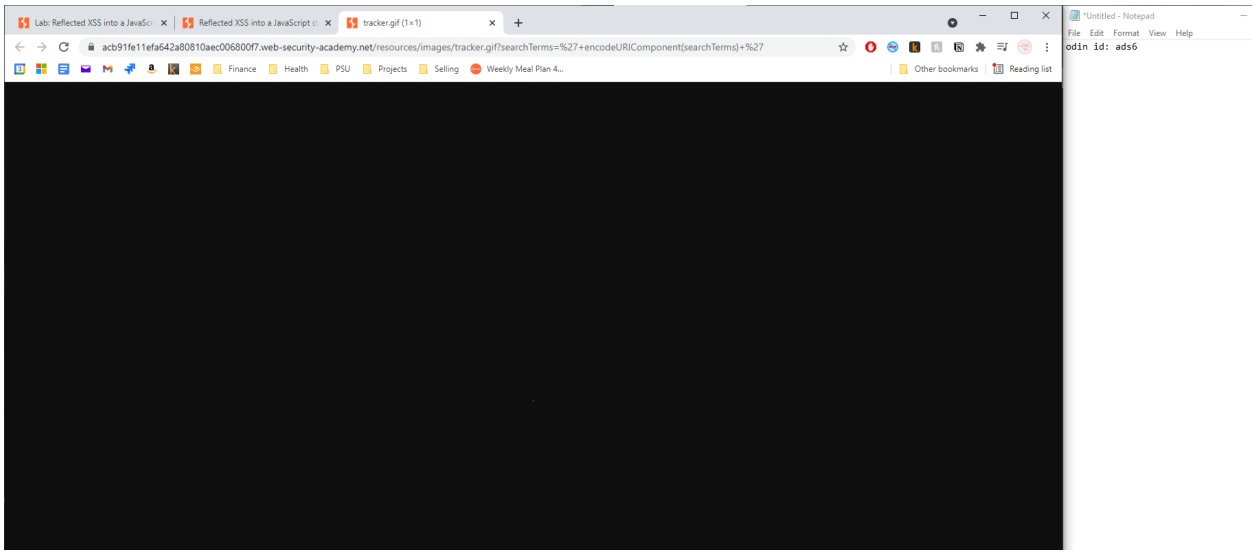


- Perform the search for `</script>` and explain why the string below the search form appears and where it came from

  The '; document.write(' '); below the search bar occurs because the inclusion of a </script> tag ends the javascript block and forces the remaining javascript to be included as content under the parent tag.

- Show its URL and explain why it no longer contains the search term

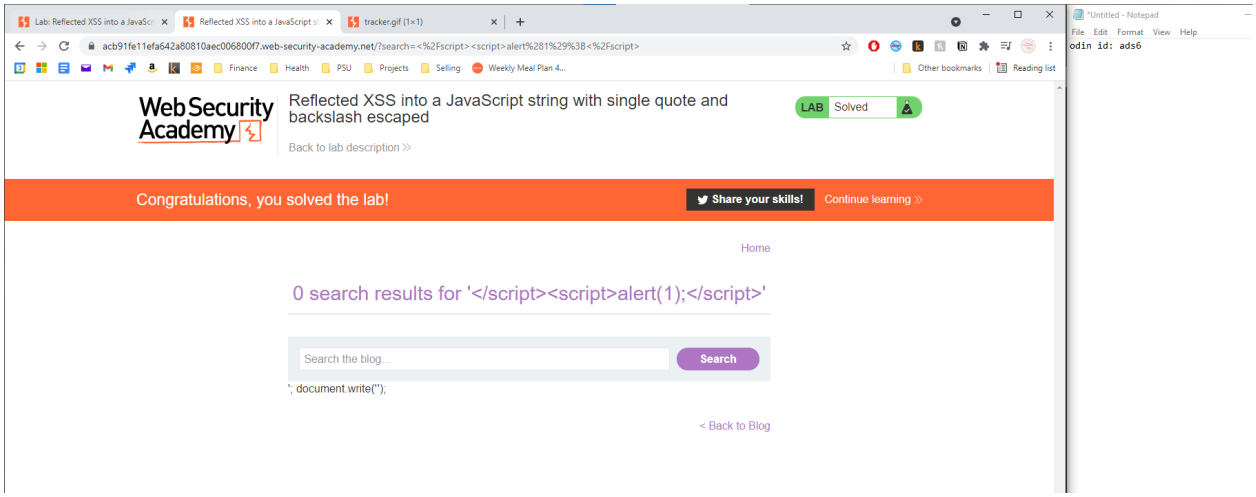
  The url no longer contains the search term because the provided term "</script>" was interpreted as the ending of the <script> block and thus the actual <img> tag, including the image link, is nothing more than the literal javascript being parsed as HTML.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The search string appears unencoded within the <script> tag, allowing the adversary to terminate the script block by including a </script> string within the search string, followed by any arbitrary javascript encased within a new <script> block.

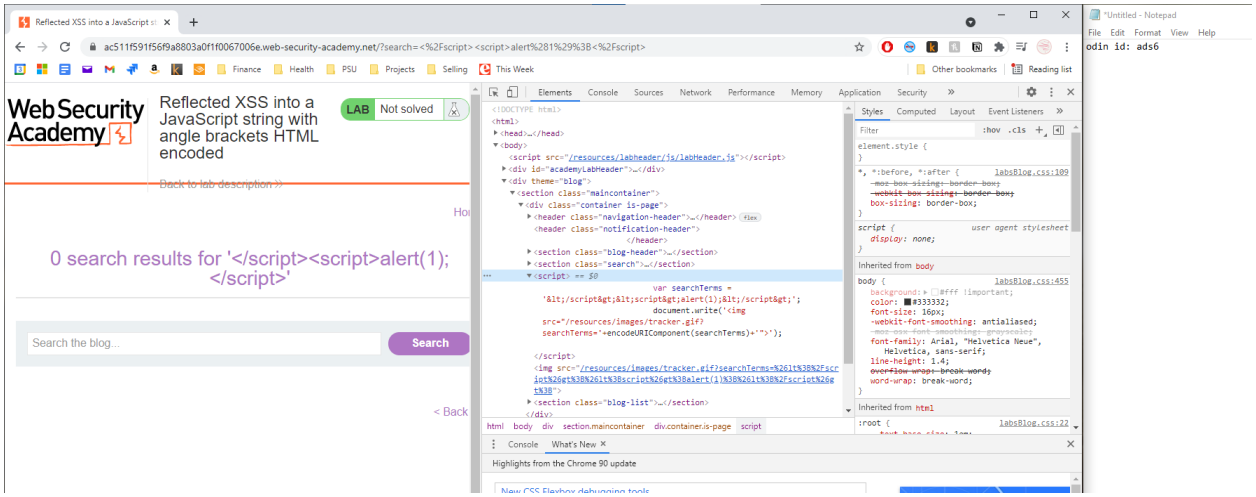
Remediation: HTML encode the search string before including it in the <script> block.



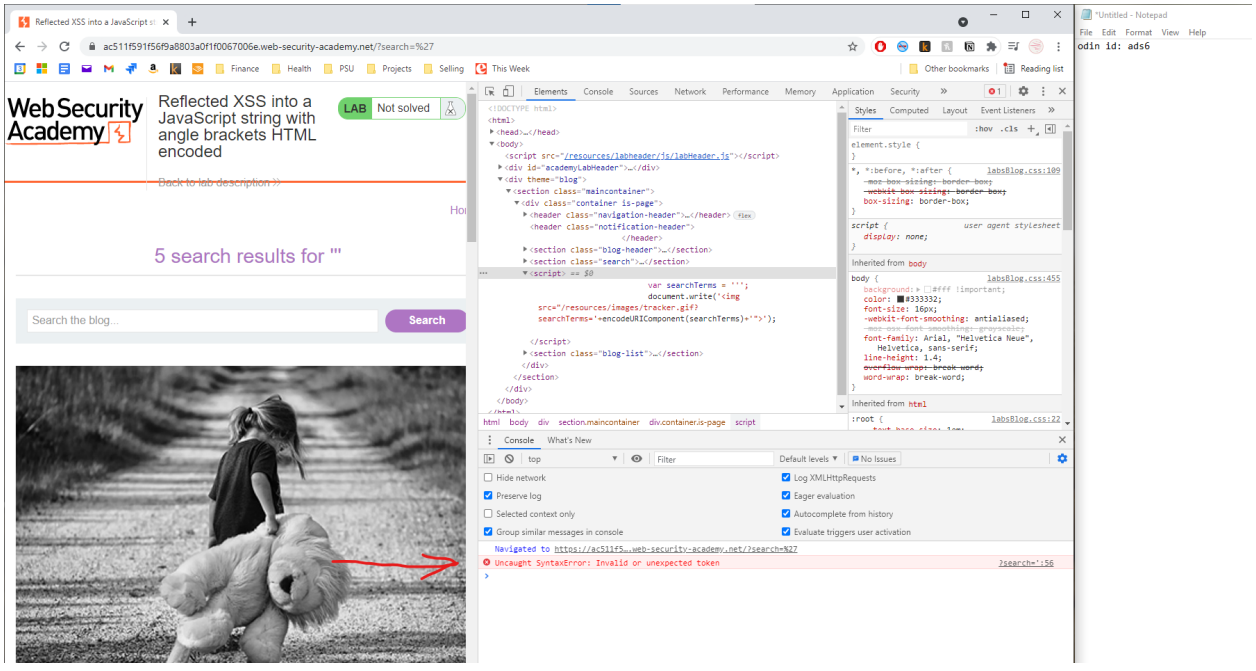
9. cross-site-scripting/reflected (6)

- Take a screenshot of the search term as it is reflected back in the Javascript code. What has been done to the search term as it appears in the Javascript code?

  It appears the search terms are being html encoded.



- Show the error that is returned and the line number it occurs on.



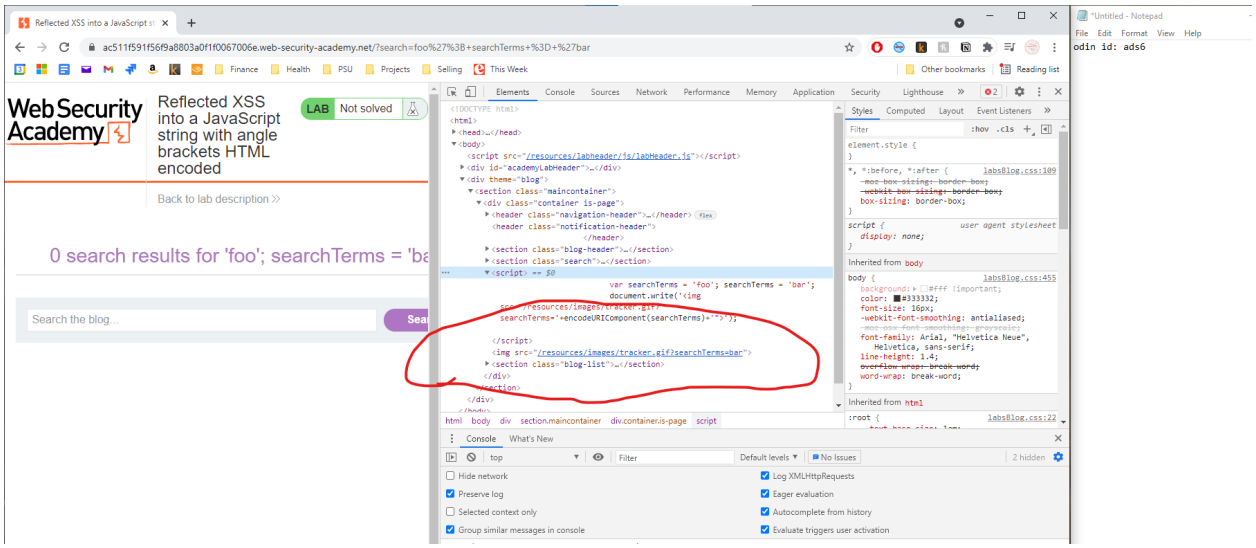
- Why does the search term not appear in the `<img>` tag?

Because the javascript engine inserting the tag believes there is only an empty string, followed by the terminal semicolon and then a comment.

- What does // do in the Javascript code?

  It is the comment marker.

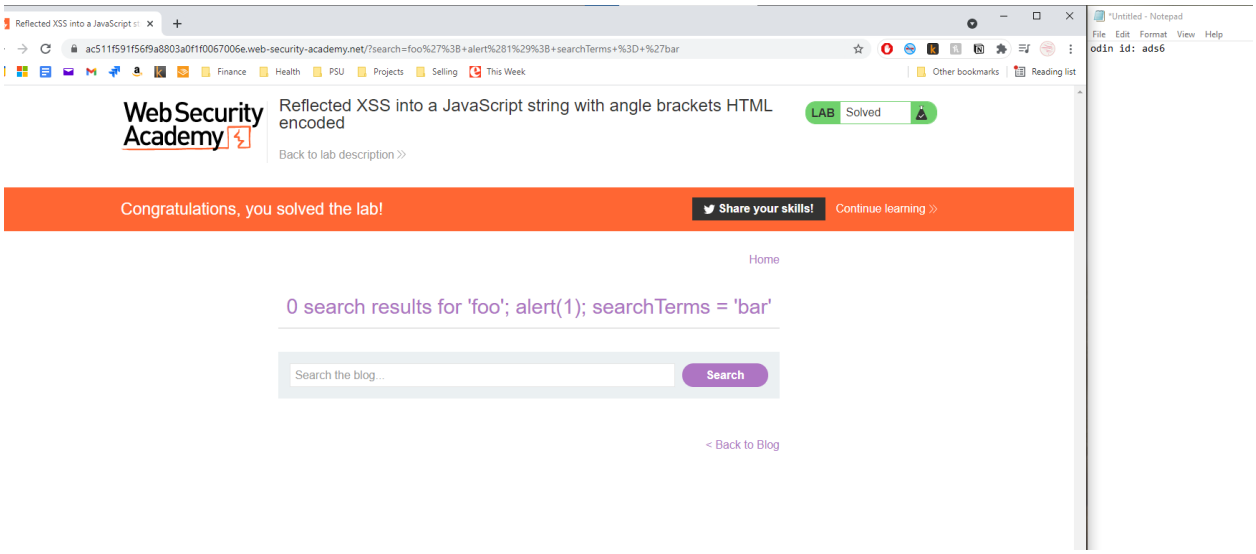
- Take a screenshot of its URL demonstrating successful injection of the second Javascript assignment.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The html encoding of the search terms that are subsequently included in the javascript do not properly escape the string terminating character (') allowing the adversary to exploit the single quote to escape out of the string and thereby insert arbitrary javascript into the client code.

Remediation: Escape the string terminating character before including it in the javascript.



10. cross-site-scripting/reflected (7)

- Explain why this error has happened and what needs to be done to fix it

  The encoding function blindly escaped the input with a single backslash, yielding the javascript code

  var searchTerms = '\\";

  Which is the interpreter sees as:

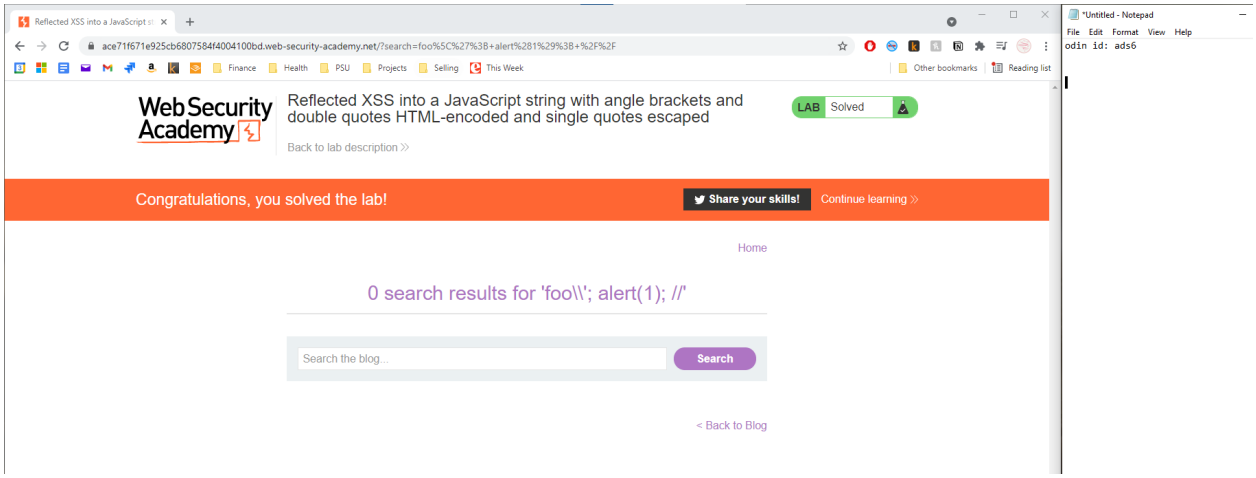
  var searchTerms = '\\' '; // The final single quote is extra

  Because the \\ is just the escaped code for \. To fix it, the escape char \ must also be escaped.

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The single-quote escape function does not account for the presence of the escape character, allowing the adversary to escape from the string using the \' combination similarly to before.

Remediation: Escape the backlash too.



11. cross-site-scripting/reflected (8)

- Show this line of Javascript

```
document.getElementById('searchMessage').innerText = message;
```

- Show the line that defines the template literal.

```
var message = `0 search results for 'ads6'`;
```

- Explain the results

  The line:

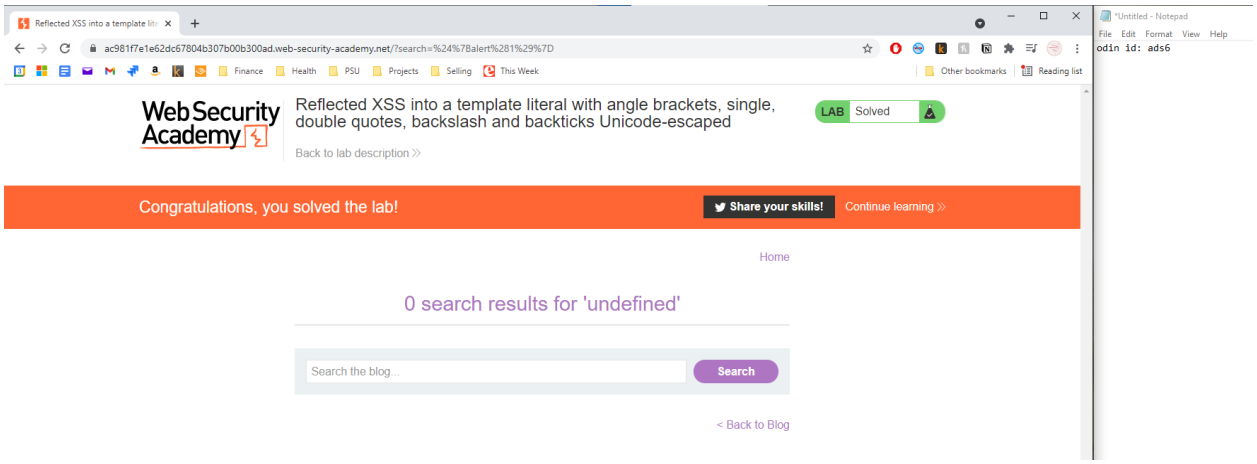
  var message = `0 search results for 'cs${490+5}'`;

  Tells the interpreter to resolve the expression 490 + 5 and put that value into the string before proceeding to set the text element's value.

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The unfortunate use of a template literal, combined with the failure to escape the characters $, {, and } permits the adversary to inject executable code within the template literal itself.

Remediation: Don't use a template literal in this case; and, more generally, escape the above-mentioned characters.



12. cross-site-scripting/dom-based (1)

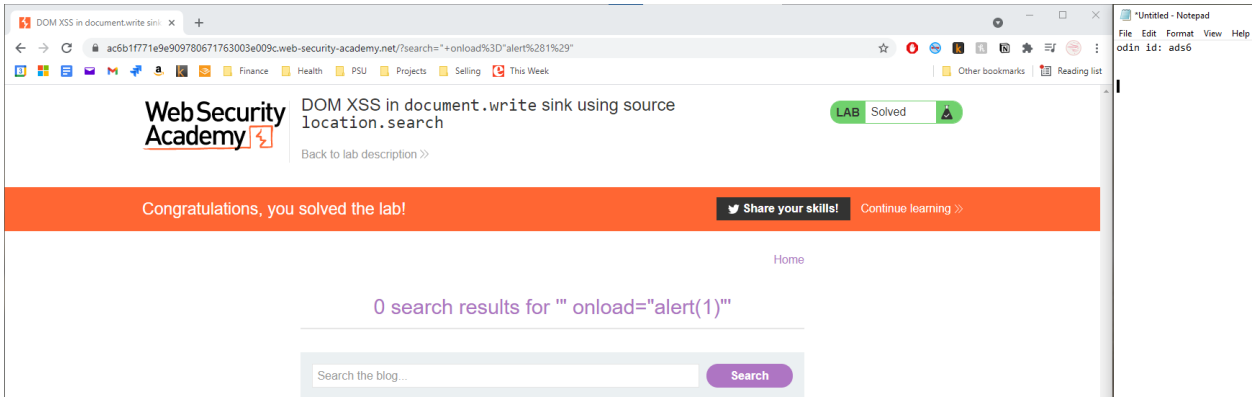
- Take a screenshot of the <img> tag in Developer Tools and show that its syntax has been broken.

```
▶ <script>…</script>
  <img src="/resources/images/tracker.gif?searchTerms=" ads6">
▶ <section class="blog-list">…</section>
```

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The failure to html encode the " character allows the adversary to escape the string within the DOM and insert arbitrary html code (including javascript.)

Remediation: HTML encode the " character.



13. cross-site-scripting/dom-based (2)

- How does the store variable get set? Can one always assume that it is one of the values in the stores list?

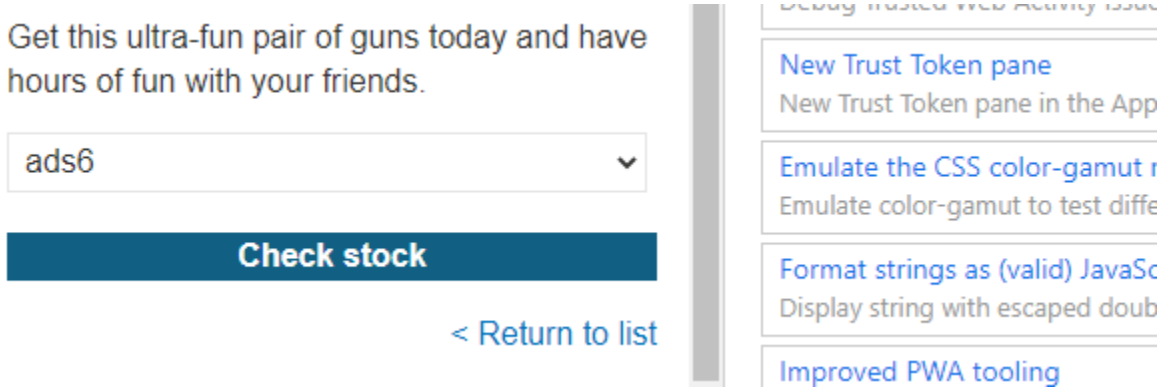
```
var store = (new URLSearchParams(window.location.search)).get('storeId');
```

  Since the store value is coming from the url search string, one cannot assume it will be one of the values in the stores list.

- What is the purpose of the first if statement? What is the purpose of the second if statement?

The purpose of the first if is to make sure the default option is the same as what was specified in the search string. The purpose of the second if statement is to make sure that default choice does not appear twice in the select menu.

- Take a screenshot showing that you have successfully injected a bogus store that is not one of the initial ones in the stores list.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The search string is not html encoded before inserting it into the <option> block, allowing the adversary to insert arbitrary html (and by extension, javascript) into the page.

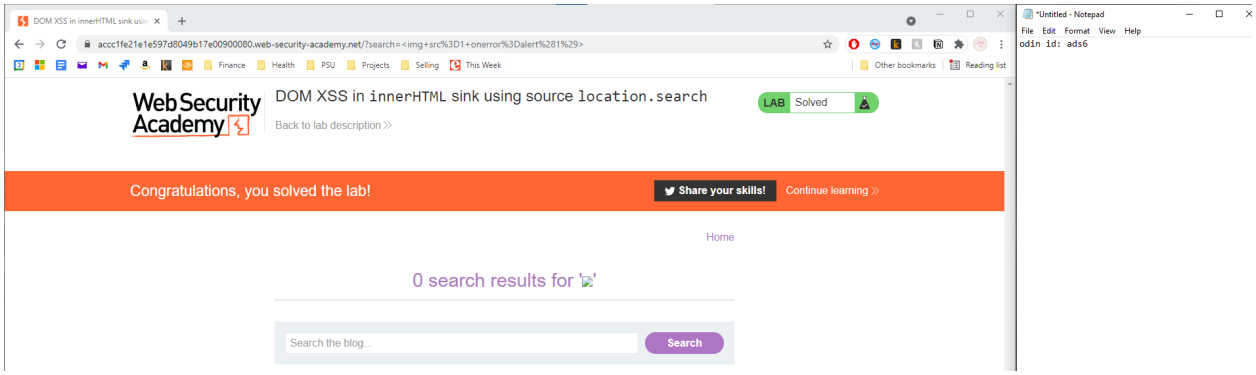
Remediation: HTML encode the search string before using it in the HTML.

14. cross-site-scripting/dom-based (3)

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The search string is not html encoded before inserting it via the innerHTML call, allowing the adversary to insert arbitrary html (and by extension, javascript) into the page.

Remediation: HTML encode the search string before using it in the HTML.



15. cross-site-scripting/dom-based (4)

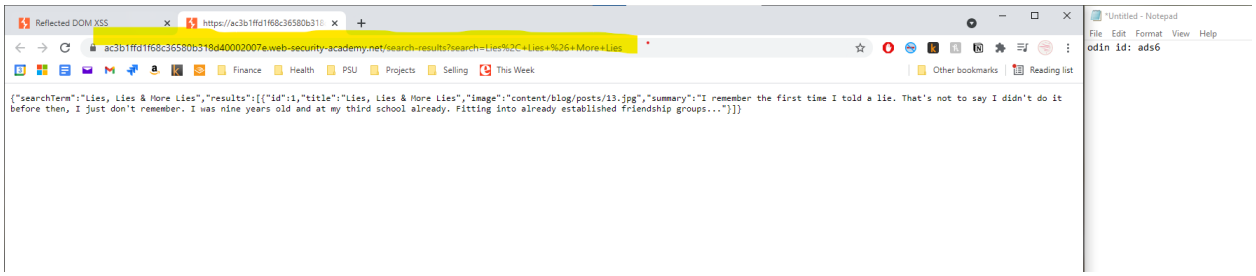
- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: Because the contents of the querystring are used unsanitized within the backlink, the adversary may inject arbitrary javascript into the link by changing the querystring.

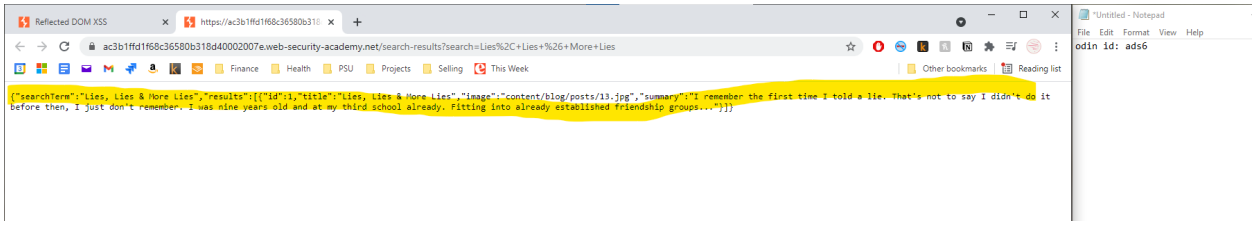
Remediation: HTML encode the search string before using it in the backlink.

16. cross-site-scripting/dom-based (5)

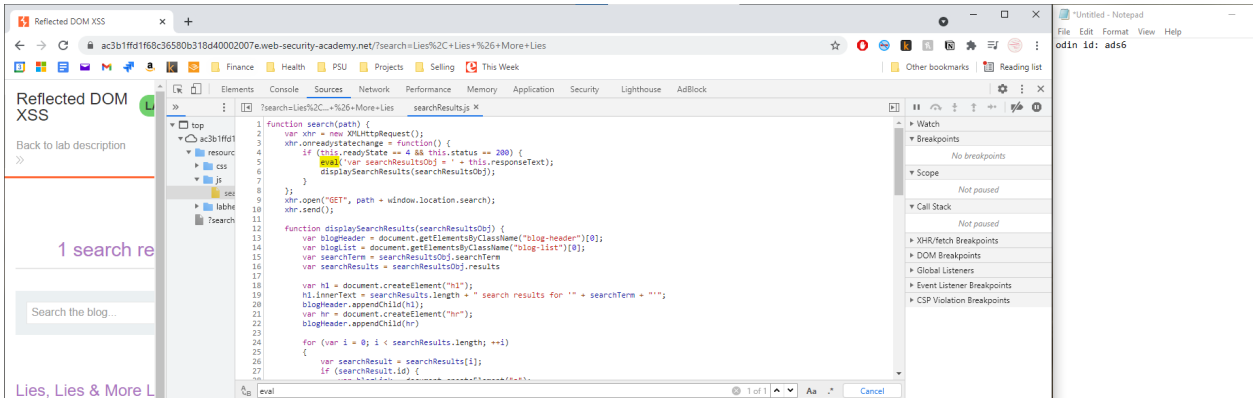
- Show a screenshot of the URI path of the XHR request



- Show a screenshot of the JSON response that echoes the search term



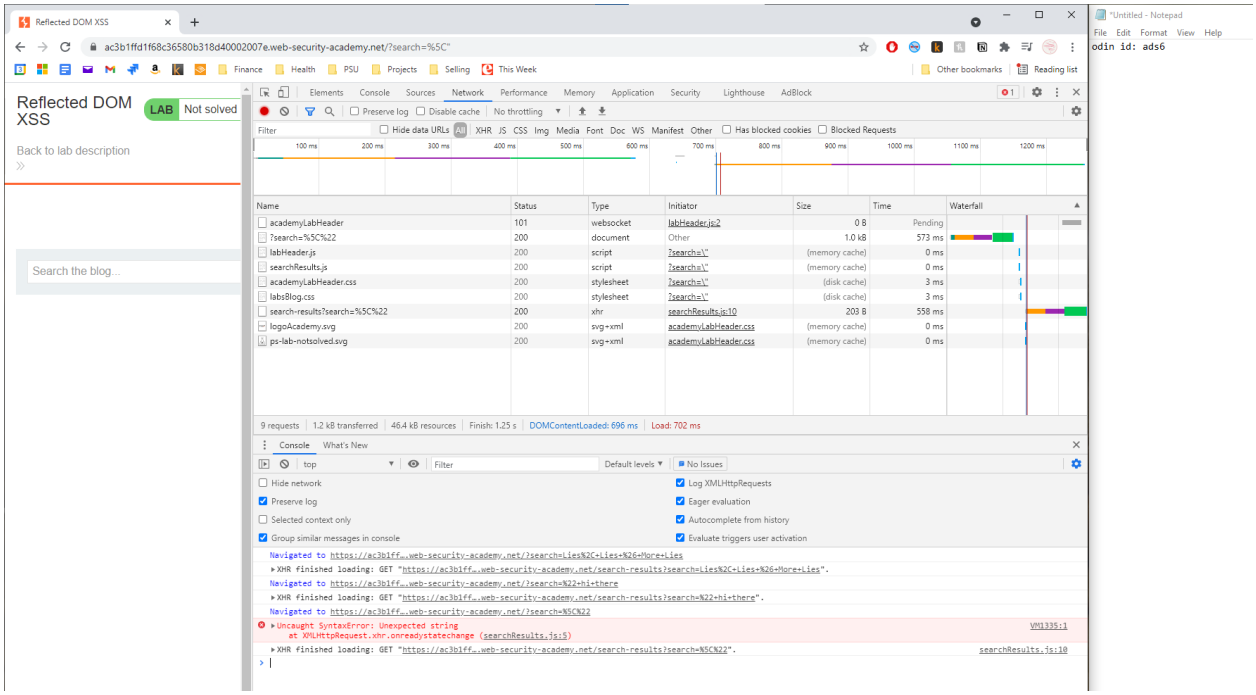
- Take a screenshot of the vulnerable line of code

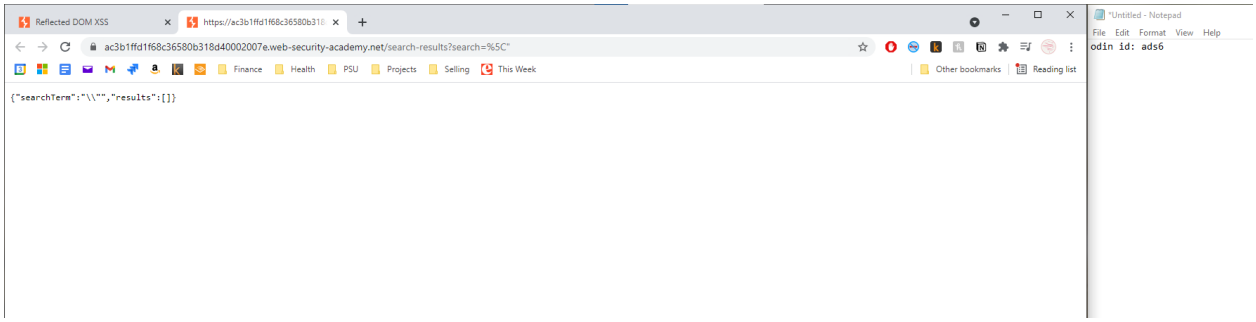


- What has been done to the delimiter to prevent syntax from being broken?

The double quote has been escaped with a backslash.

- Take a screenshot of the error that has been produced in the console. Examine the JSON response to verify that you have broken syntax.

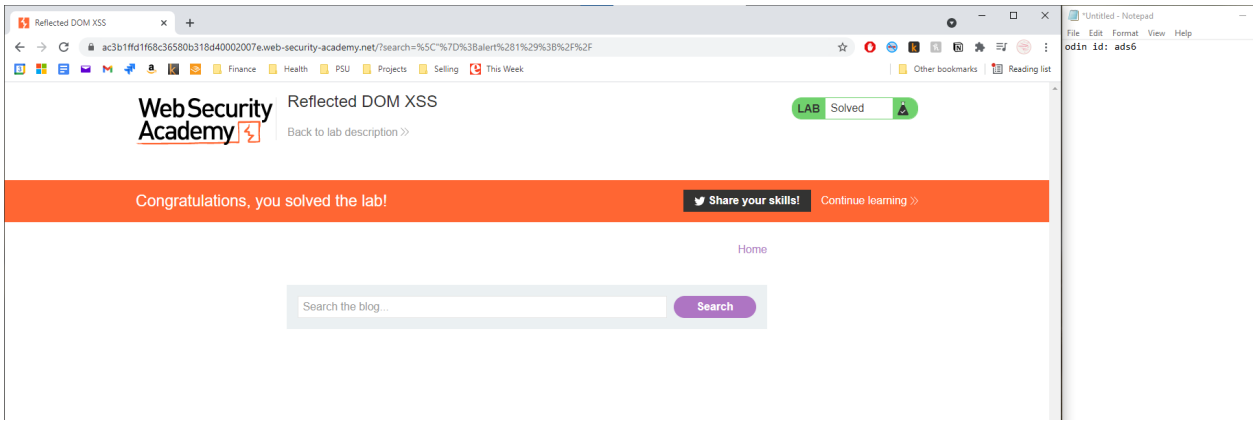




- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: Because the query string contents are not adequately sanitized before incorporating them into the json document, the adversary may force the client to execute arbitrary javascript by incorporating it into the search string.

Remediation: Properly escape backslashes as well as double quotes.

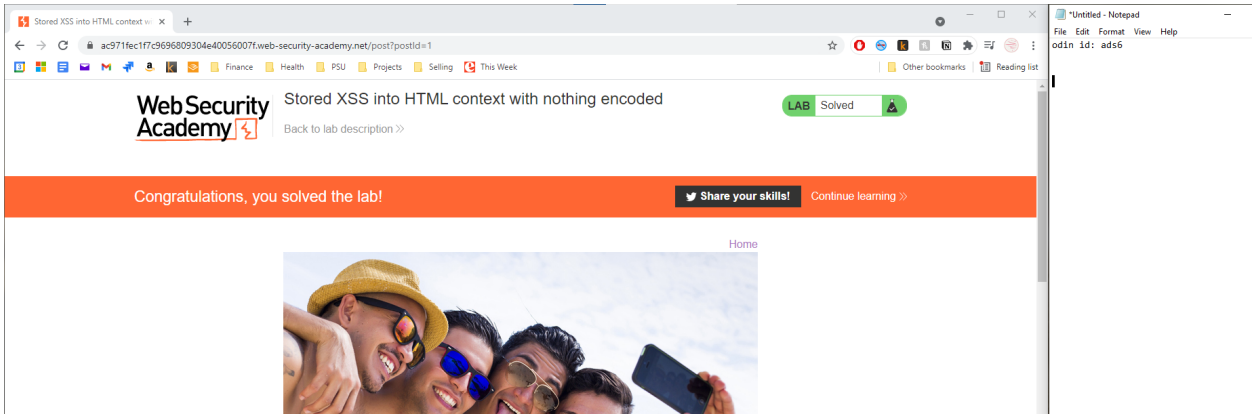


18. cross-site-scripting/stored (1)

- Take a screenshot showing completion of the level that includes your OdinId

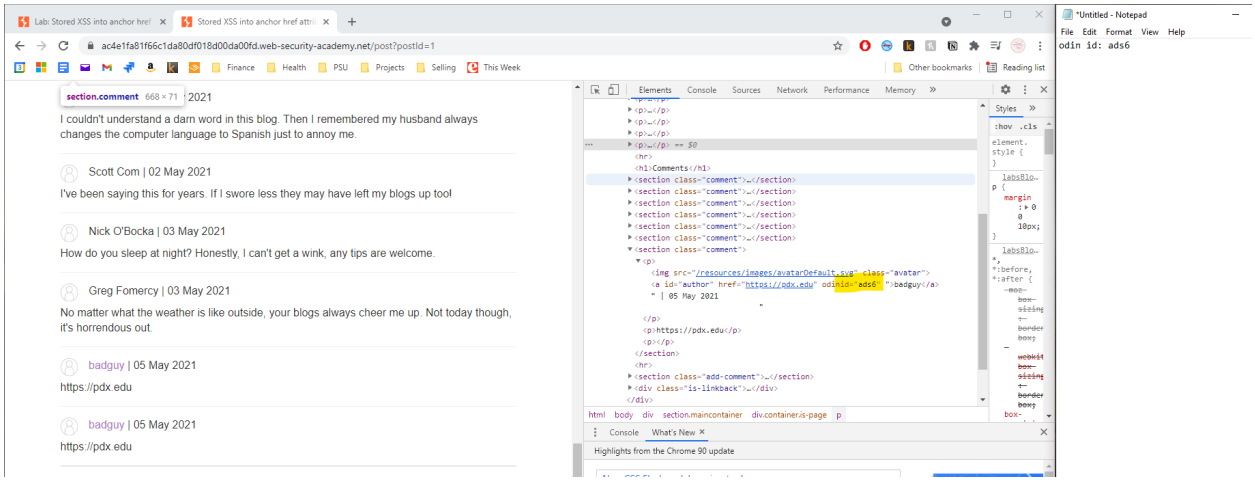
Vulnerability: Because the comment is not sanitized before incorporating into the page's html, the adversary may inject arbitrary javascript into the page via the comment function.

Remediation: HTML encode the comment before incorporating it into the page.

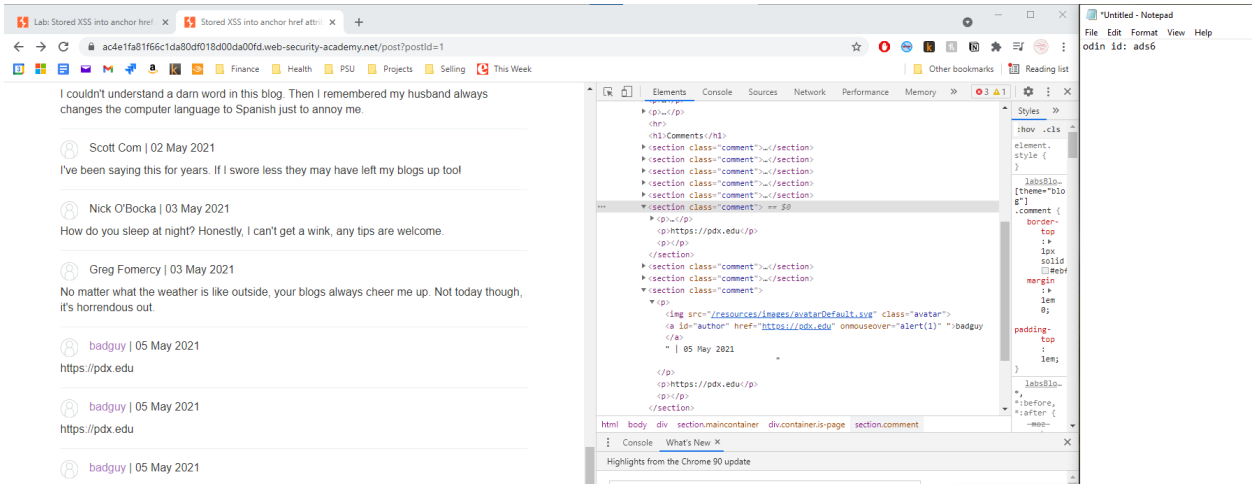


19. cross-site-scripting/stored (2)

- Take a screenshot showing that you have successfully added the OdinId attribute to the author's website link.



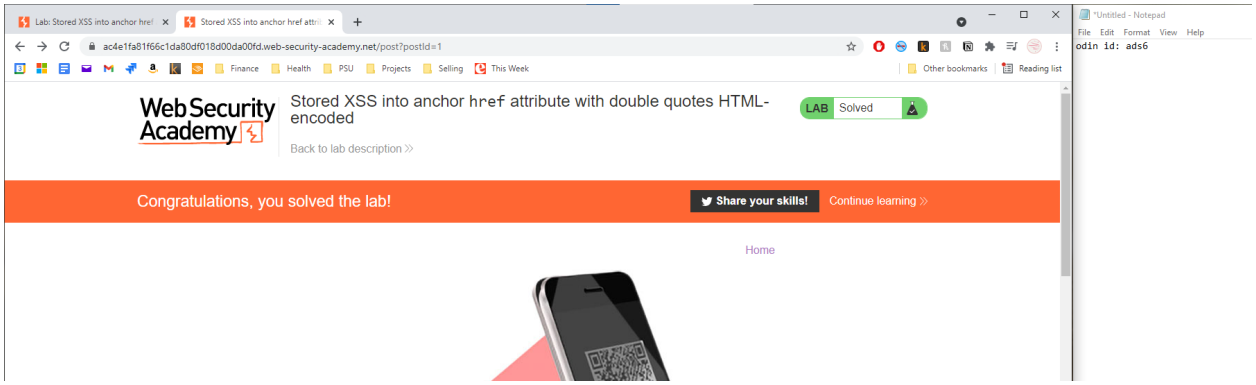
- Show the stored <a> tag you have used that pops up this alert()



- Take a screenshot showing completion of the level that includes your OdinId

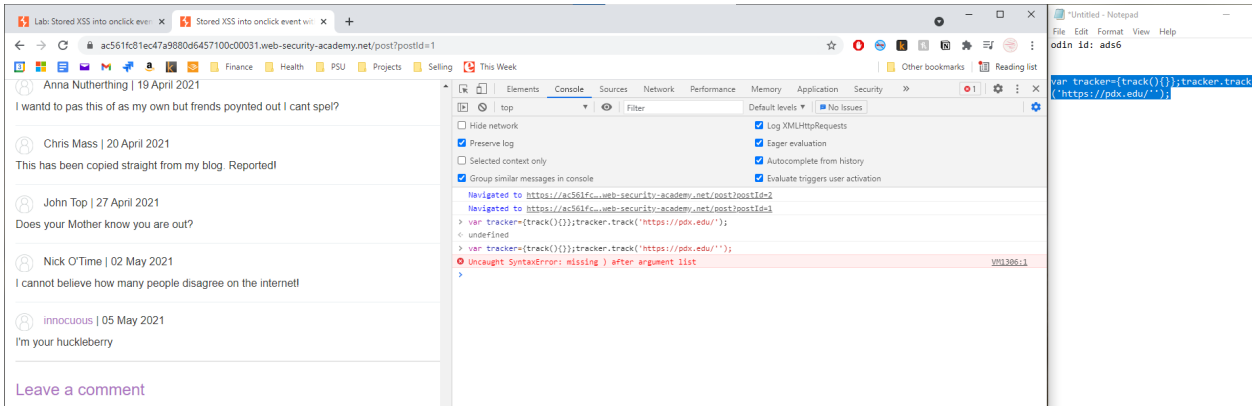
Vulnerability: Because the double quotes are not properly incorporated before incorporating the website field into the <a> tag in the HTML, the adversary may break out of the context of the a-tag and inject arbitrary javascript which any browser loading the page thereafter will execute.

Remediation: HTML encode the url before incorporating it into the page.



20. cross-site-scripting/stored (3)

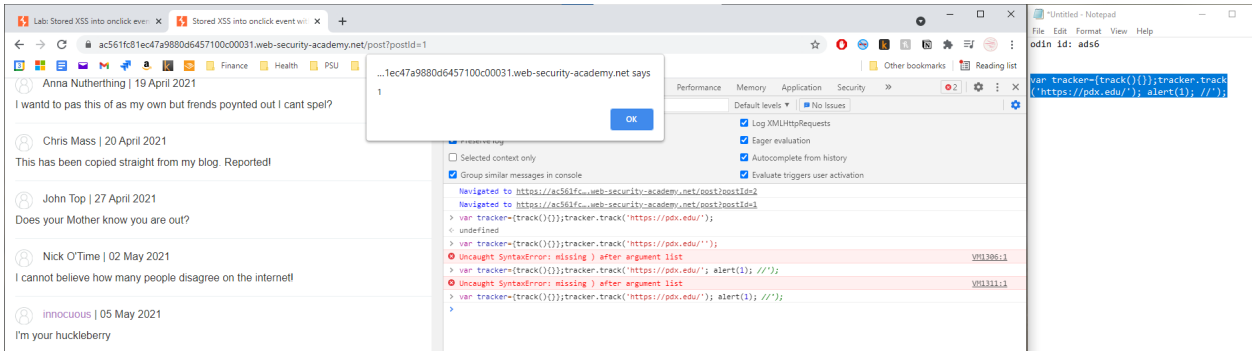
- Replace the URL with `https://pdx.edu'` (e.g. the original URL with a single-quote added to break Javascript syntax). Execute the code and show a screenshot of the error that is returned.



- Explain what happens when the URL is replaced with `https://pdx.edu');//`

  The subsequent javascript will be ignored, as it is all considered to be on a single line and is preceded by the comment marker.

- Finally, using this URL, insert an `alert(1);` into it and take a screenshot of the results in the console including the pop-up



- Explain why they differ

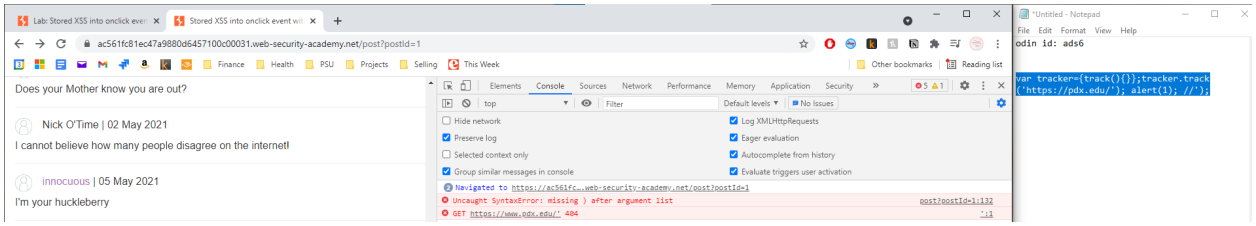
The double quote is not being escaped with a backslash.

- Are the results the same?

Yes.

- Take a screenshot of the error message in the console. Is it similar to one that you have seen when crafting an exploit?

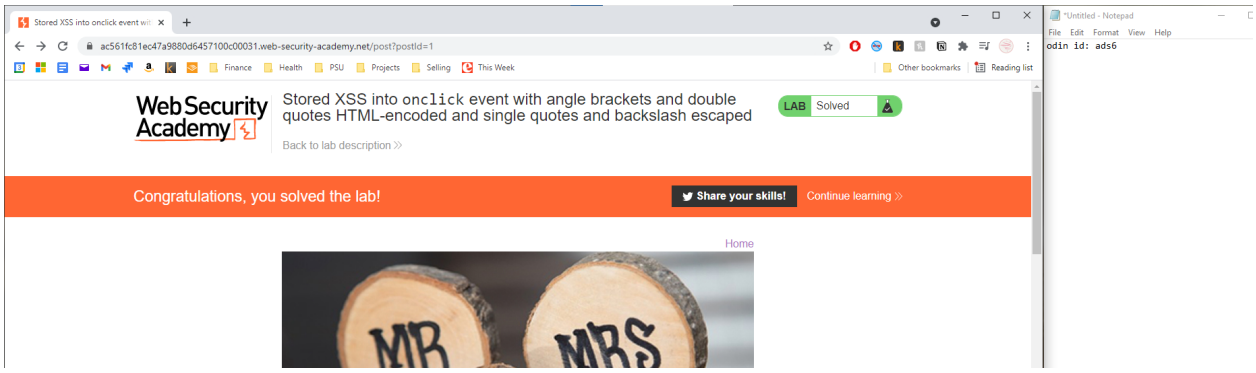
Yes, it appears to be identical.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The contents of the website field are HTML decoded after the javascript encoding, allowing the adversary to slip in an HTML encoded single quote that will get through the filter and permit them to escape from the string within the context of the javascript. Thus, the adversary may inject arbitrary code to run as part of the web page.

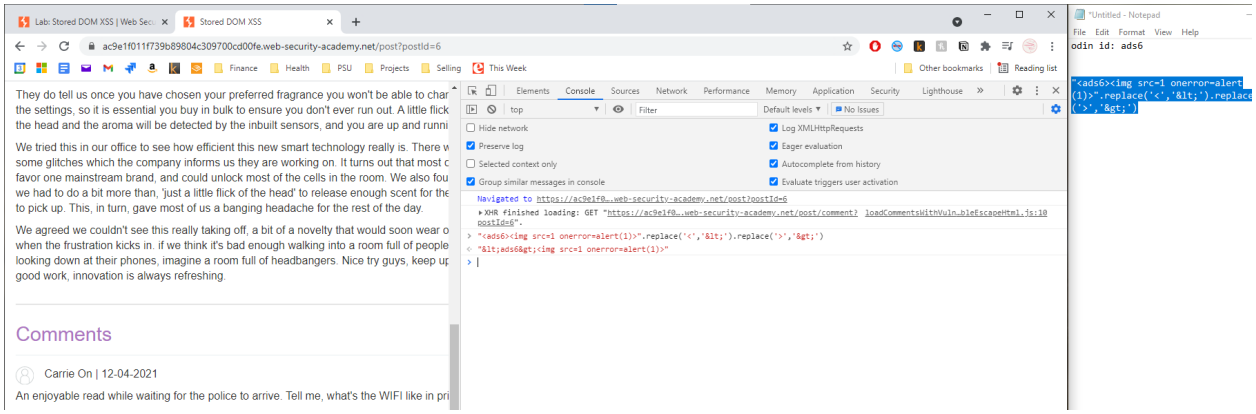
Remediation: HTML decode the field before escaping it for the javascript.



23. cross-site-scripting/dom-based (6)

- Take a screenshot of the result and explain the issue. What other built-in String method could be used instead to fix this particular issue?

The issue is that the replace() method only replaces the first instance of the value (unless the value is a regex.) The (more) correct method would have been replaceAll().



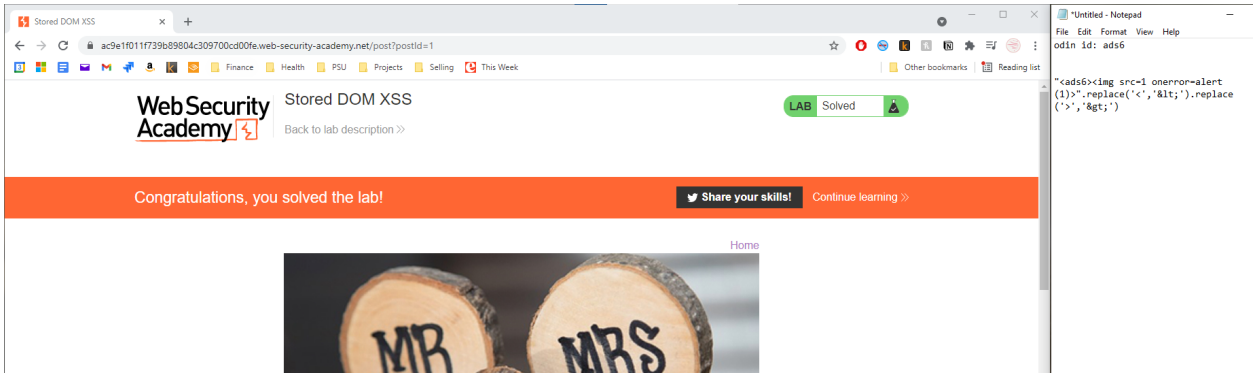
- Which three fields of JSON are vulnerable to a cross-site scripting attack?

Website, body, author

- Take a screenshot showing completion of the level that includes your OdinId

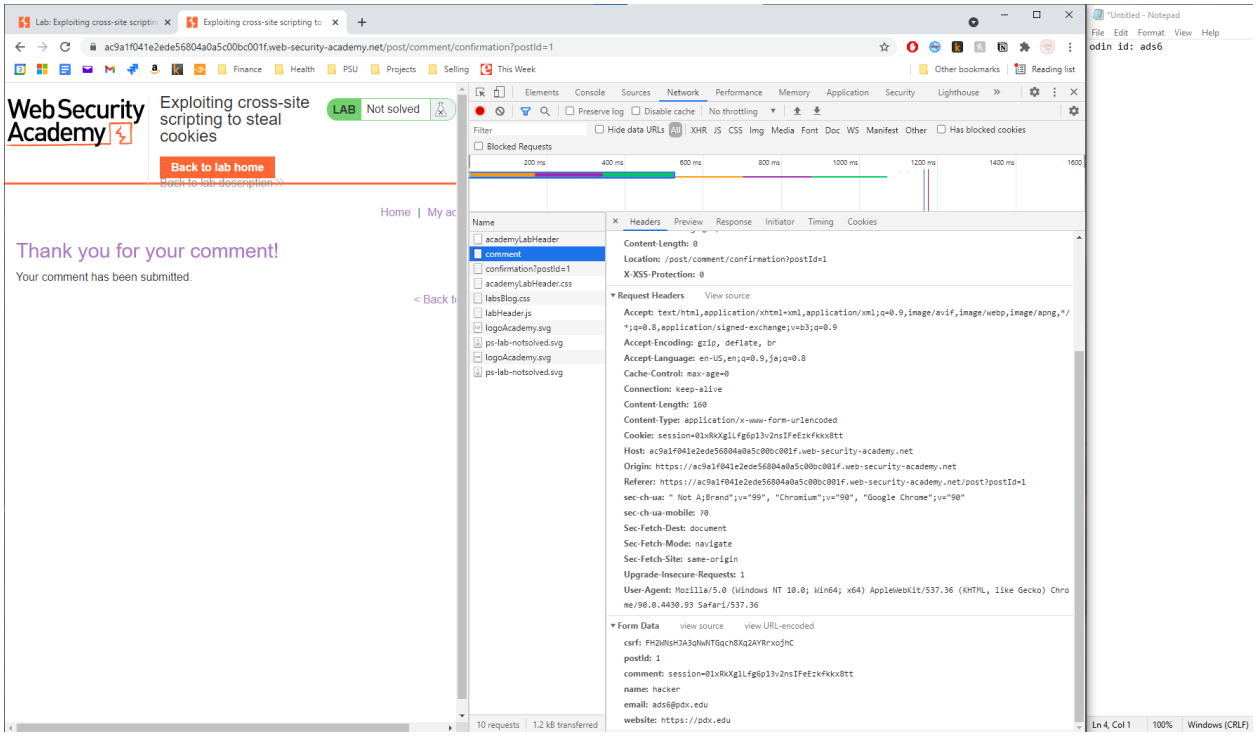
Vulnerability:The inappropriate use of the javascript replace() function, when replaceAll() would have been more correct, allows the adversary to bypass the escapeHTML function by prepending their attack string with a simple <>.

Remediation: Use replaceAll() instead of replace().



25. cross-site-scripting/exploiting (1)

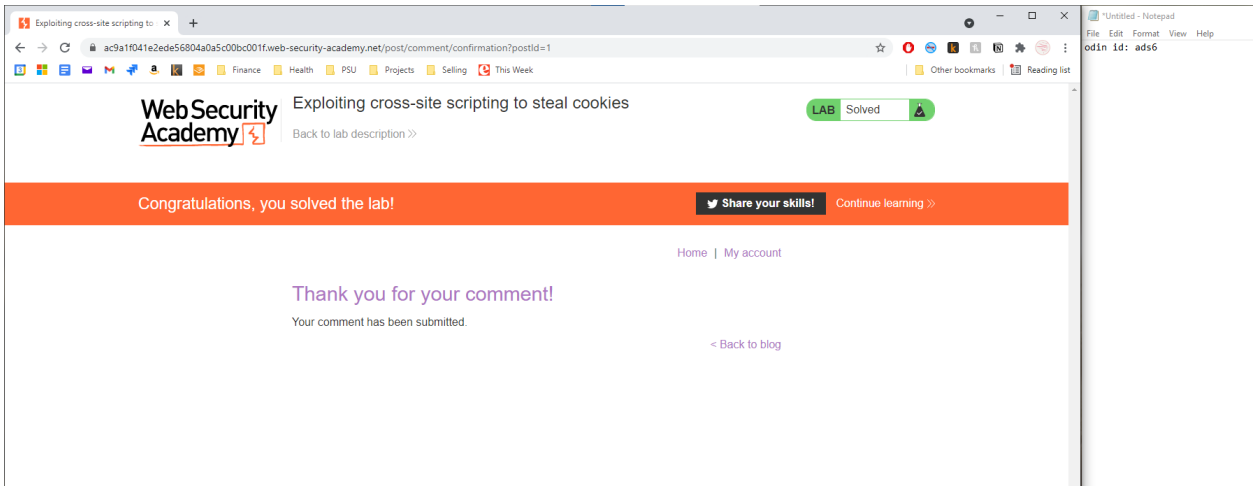
- Take a screenshot of the headers showing all of the form data for your POST including your own exfiltrated cookie sent as a comment.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The above vulnerability in comments submission feature allows the adversary to exfiltrate the secret cookie from other users, either by using another site to "silently" register the user's private data, or (less subtly) to automatically add other user's cookies to the HTML.

Remediation: HTML encode all the comment fields before incorporating them into the HTML.

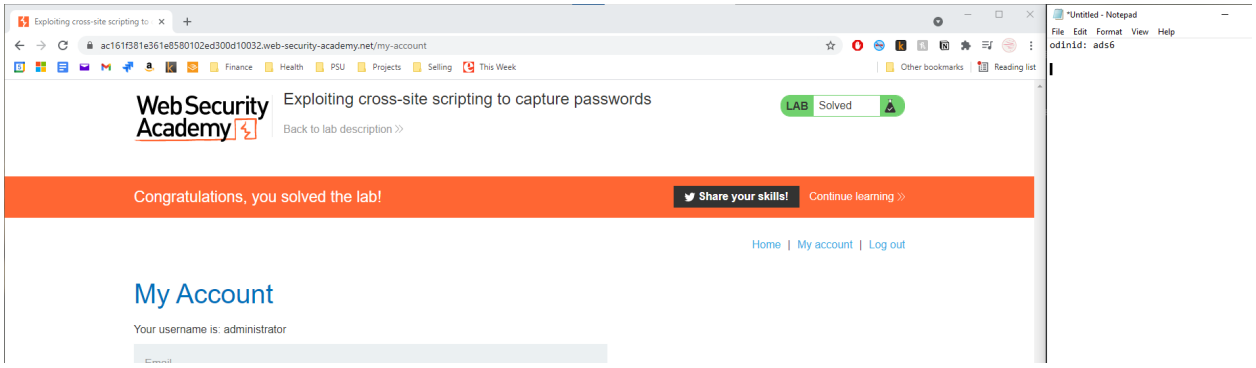


28. cross-site-scripting/exploiting (2)

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability:The failure to properly sanitize the comments before adding them to the HTML allows the adversary to inject a form into the site that all users will load; since the administrator automatically fills out this form, the adversary may also automate the posting of these credentials as a second comment via the onchange event handler.

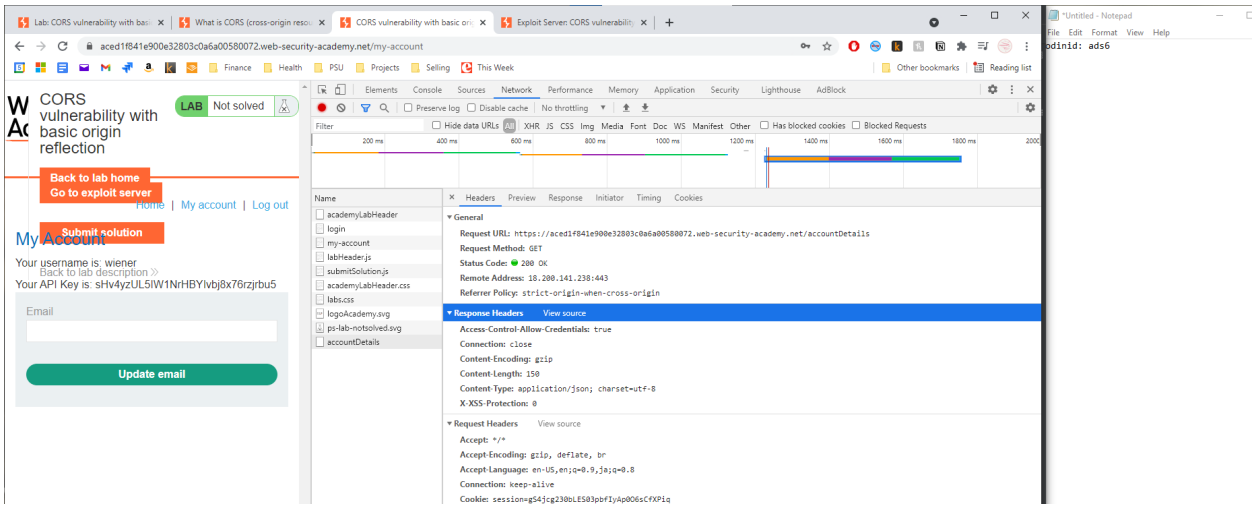
Remediation: HTML encode all the comment fields before incorporating them into the HTML.



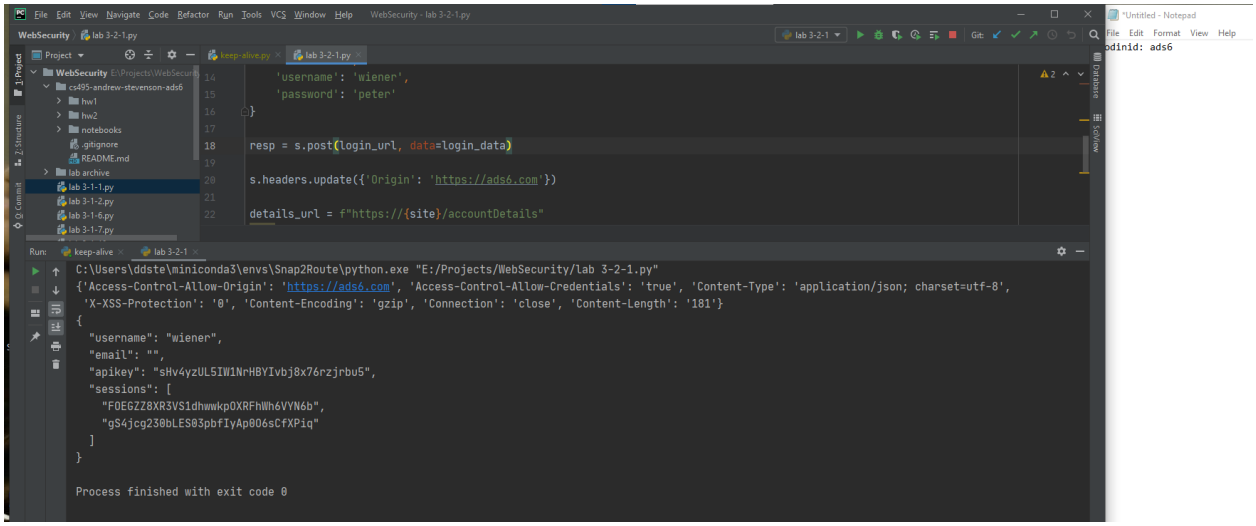
Lab 3.2

1. cors (1)

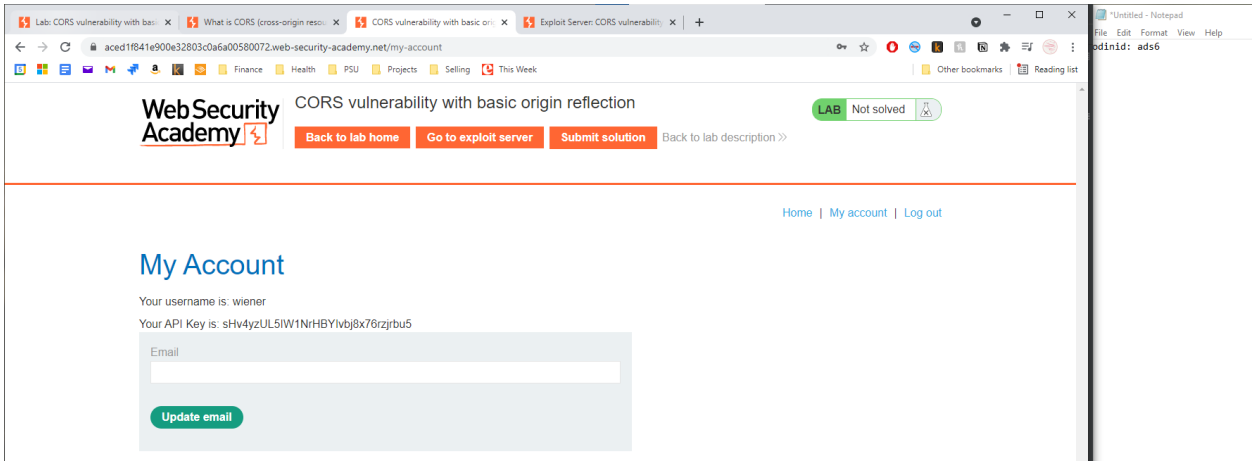
- Show a screenshot of the CORS header that enables credentials to be sent to get the key.



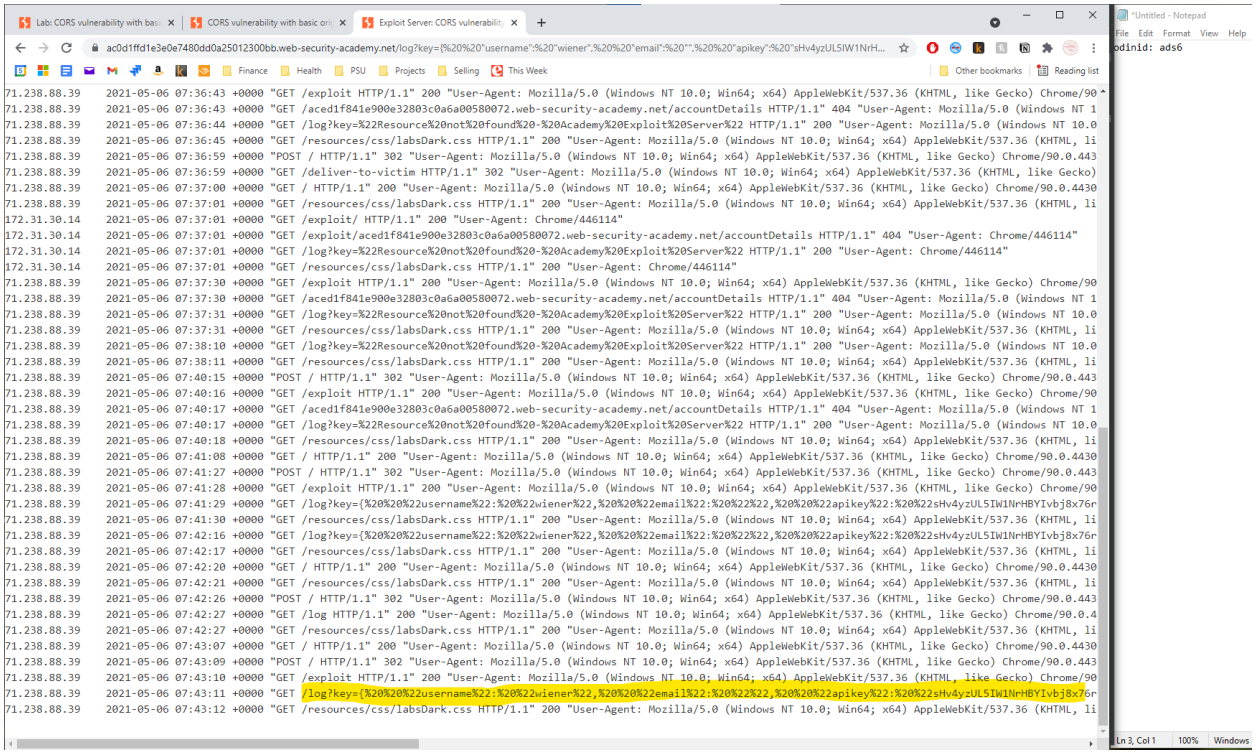
- Show a screenshot of the headers in the response to see that the site simply reflects the header in `Access-Control-Allow-Origin:`.



* Take a screenshot of the API key and include it in your lab notebook.



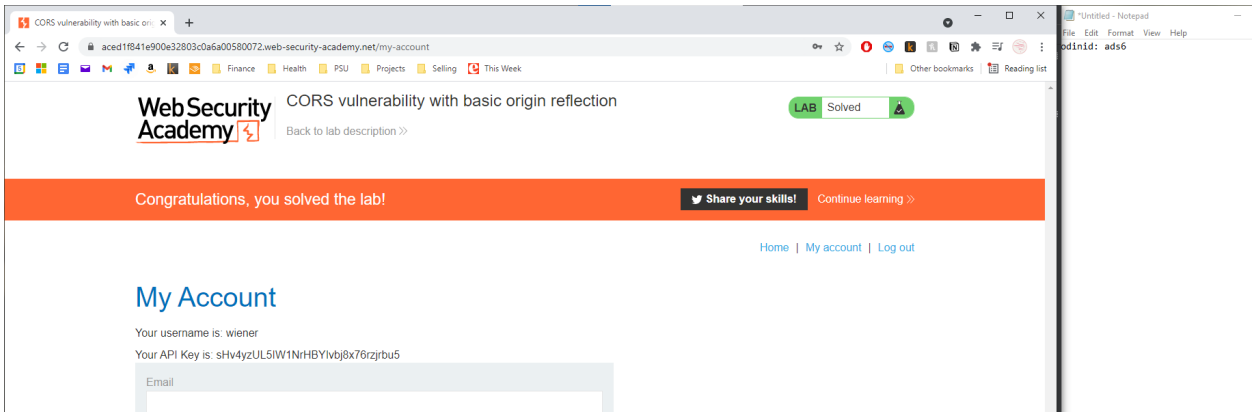
- Take a screenshot of the API key as it appears in the browser window.



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The server reflects back the request's Origin header in its Access-Control-Allow-Origin response header. Thus, the adversary may collect stored user information when the victim visits a site the adversary controls after logging in to the target site.

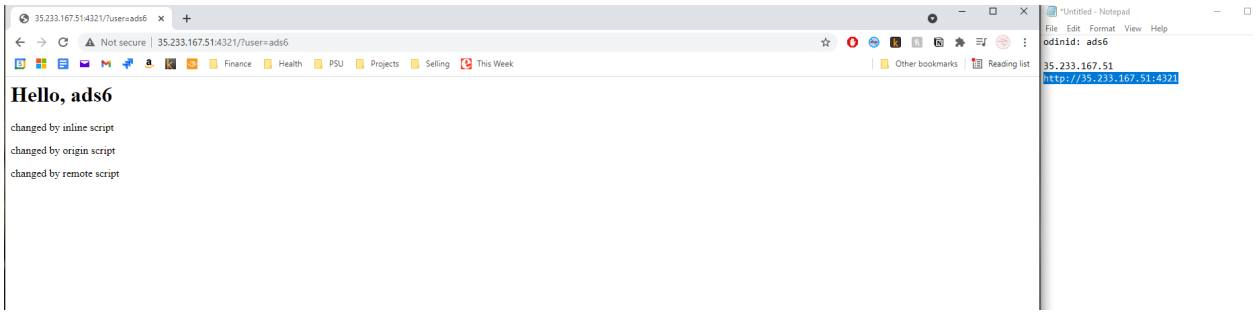
Remediation: Properly set the Access-Control-Allow-Origin header to restrict which sites may view the user information.



8. Content-Security-Policy examples

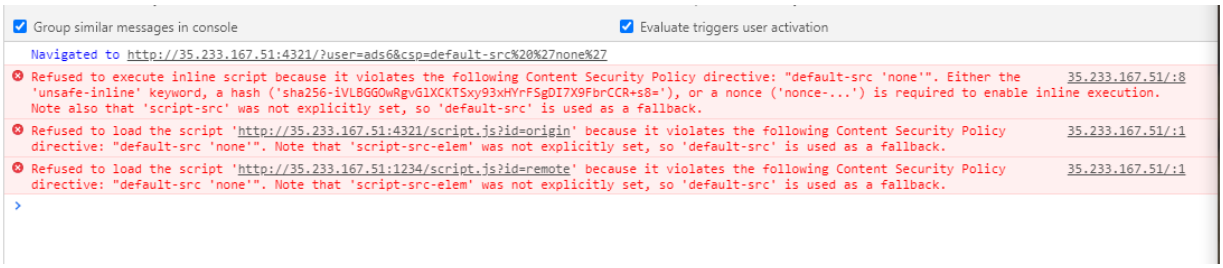
Example #1

- Take a screenshot of the page result that includes the URL in the browser

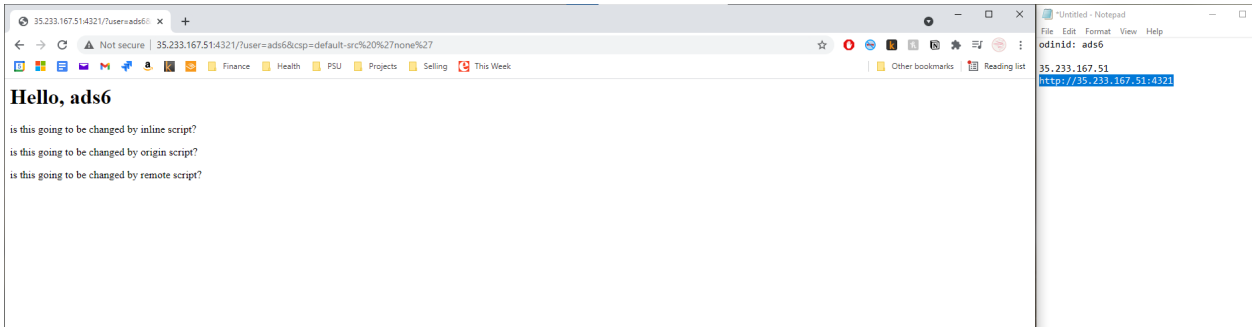


Example #2

- Take a screenshot of the console output showing all scripts blocked



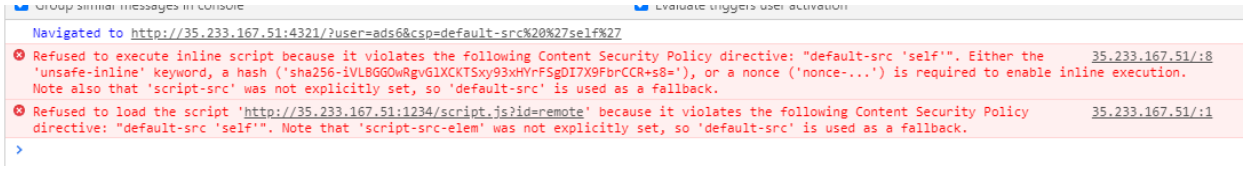
- Take a screenshot of the page result that includes the URL in the browser



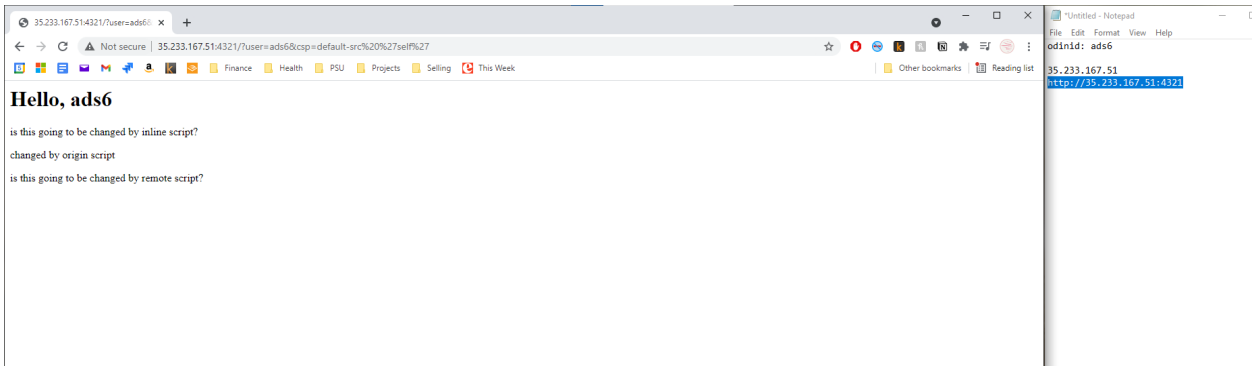
Example #3

- Take a screenshot of the console output showing the scripts that have been blocked. Explain why these results differ from the previous example.

The Content-Security-Policy header was set to self, so scripts from the same location as the file are allowed.



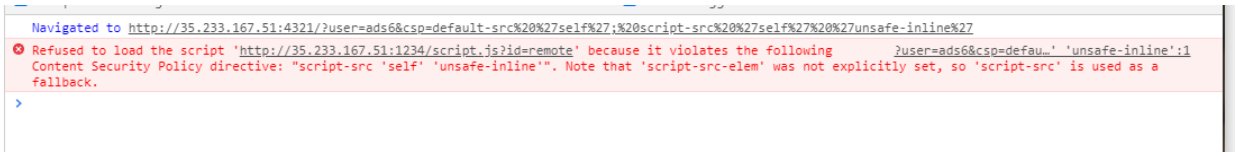
- Take a screenshot of the page result that includes the URL in the browser



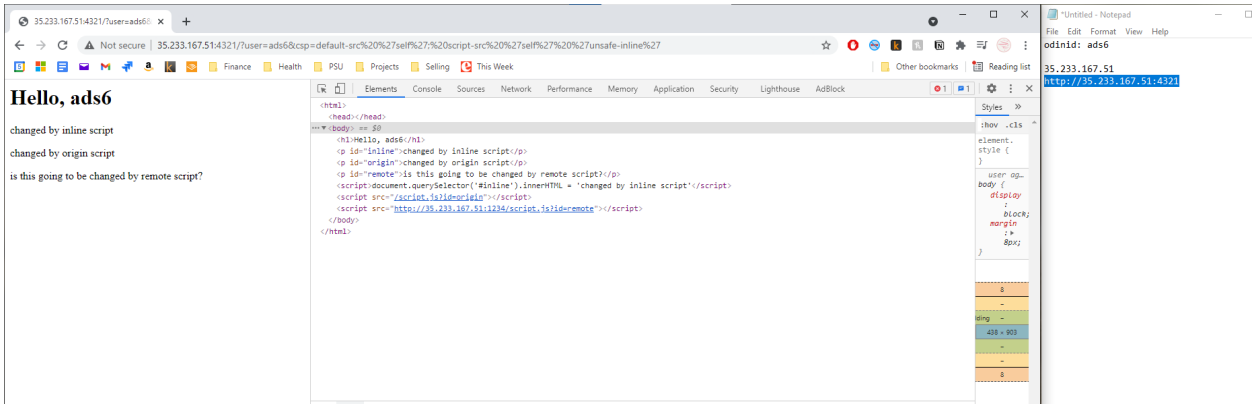
Example #4

- Take a screenshot of the console output showing the scripts that have been blocked. Explain why these results differ from the previous example based on the additional options given.

  The inclusion of the unsafe-inline for script-src suffices to permit the inline script to execute, as was hinted by the error message for the inline node in the above examples.



- Take a screenshot of the page result that includes the URL in the browser



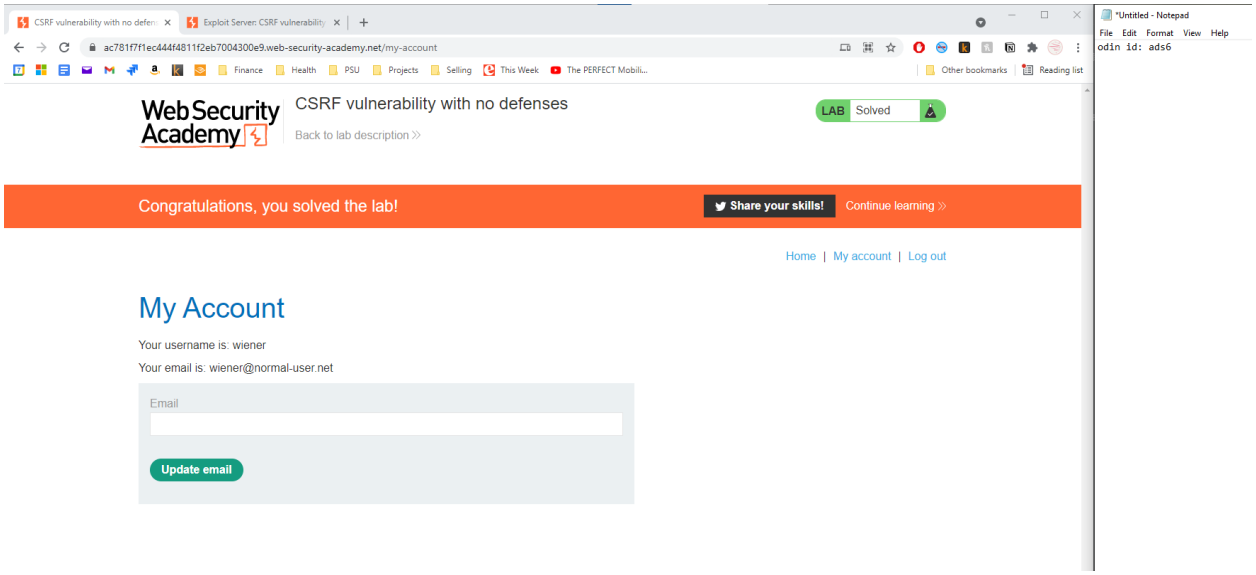
Lab 3.3

1. csrf (1)

- Take a screenshot showing completion of the level that includes your OdinId

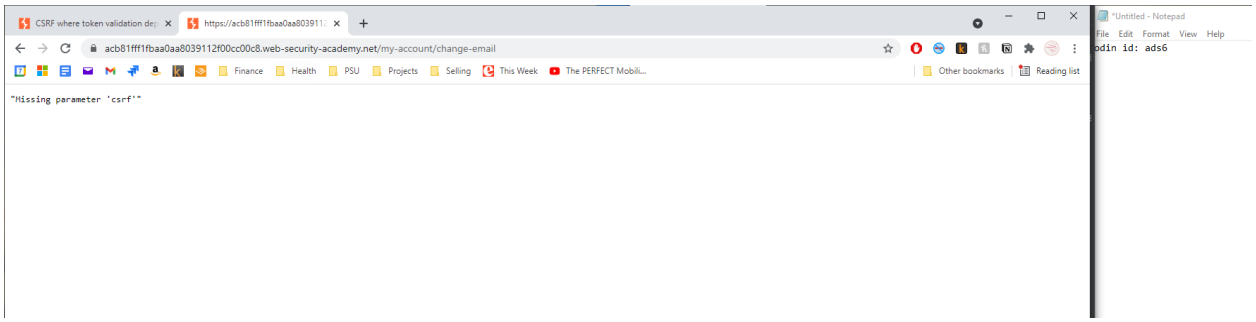
Vulnerability: The submission url takes no precautions against unauthorized requests, allowing the adversary to submit any email change requests they desire.

Remediation: Use a CSRF token to ensure the request has come from the associated, validated, user.



3. csrf (2)

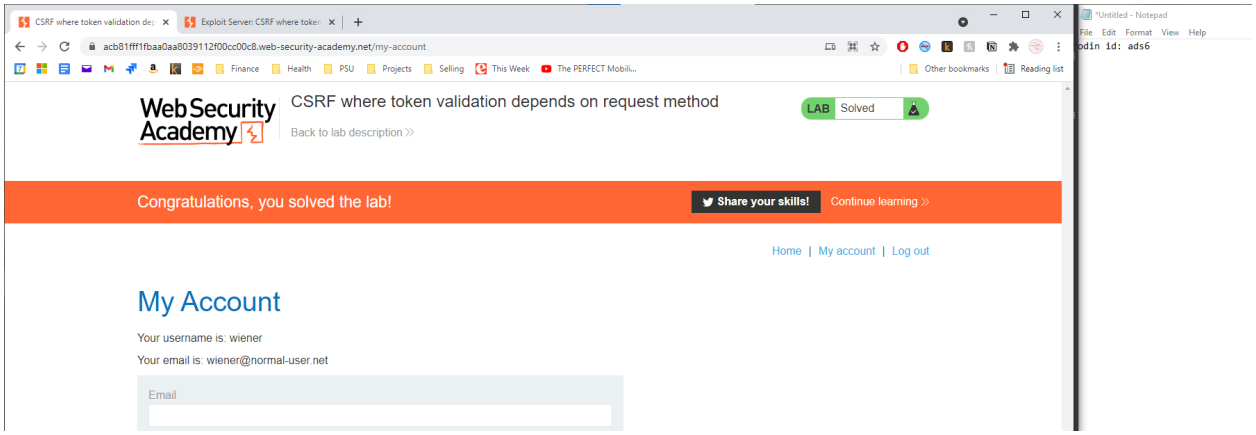
- Take a screenshot of the result showing that the exploit has failed



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: Although a CSRF token is now in place, the GET route is not protected by the token, permitting the adversary to make the same attack as before.

Remediation: Protect the GET route with a CSRF token.

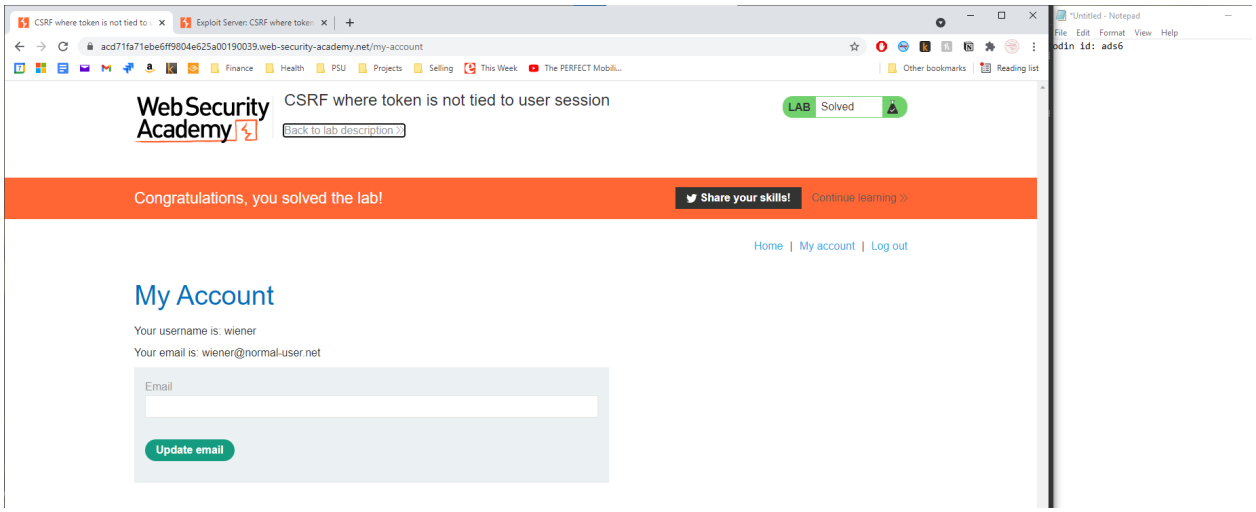


4. csrf (3)

- Take a screenshot showing completion of the level that includes your OdinId

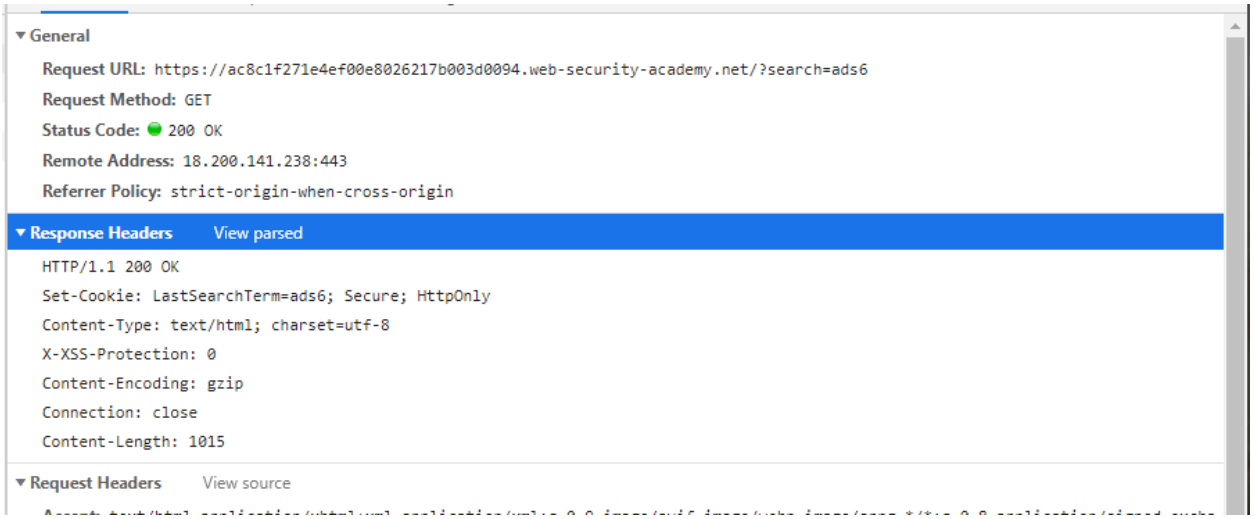
Vulnerability: Although a CSRF token is now in place, it is not tied to a user session, allowing the adversary to retrieve a token and embed it into their attack site form--which enables them to repeat the same attack as above.

Remediation: Ensure that the CSRF token is associated with a user session, and verify that connection before accepting the form submission.



5. csrf (4)

- Take a screenshot of the entire HTTP response header that includes your OdinId



- How many cookies are returned? What are their names?

One Cookie: LastSearchTerm.

- How have foo and bar been interpreted?

As a new line in the http response header.

- How many cookies have been returned?

Two Cookies.

- Explain the results. Give one reason why a developer would choose to implement CSRF protection in this manner

The server simply compares the csrf in the form to the csrf in the cookie and checks if they're the same. Probably implemented this way because it's (1) easier to program, (2) allows for multiple simultaneous tabs with the same user; and (3) requires no back-end storing of csrf.

- What status code is returned? What is the value of the `csrf` field in the HTML form that is given back as a response?

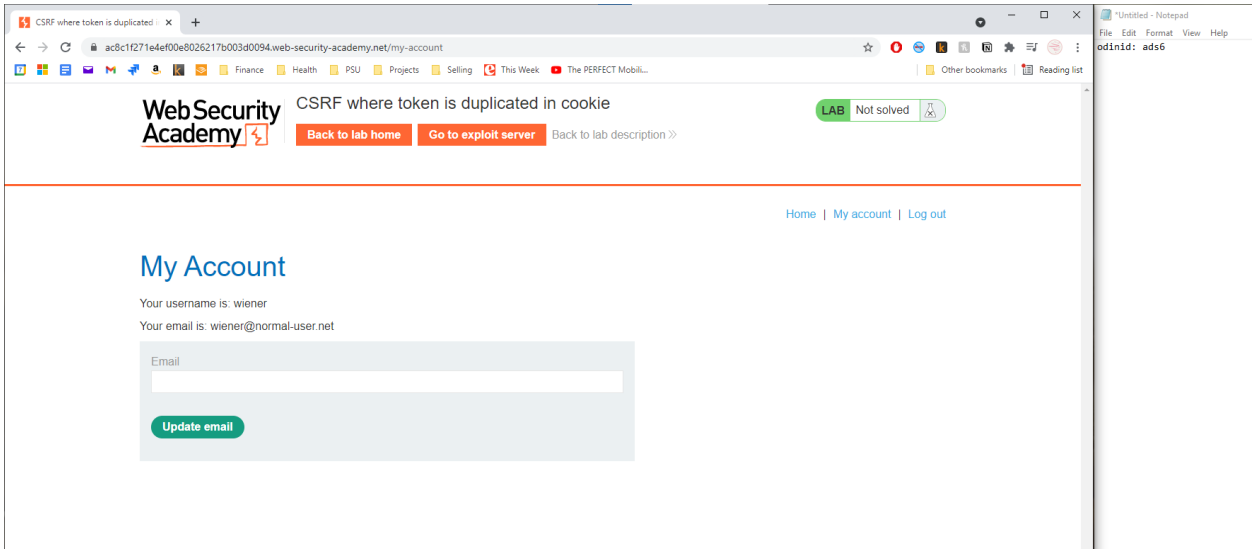
HTTP status code 200

CSRF token in HTML response is ads6

- How might a developer use a keyed hash function on the server to prevent this request from succeeding without being forced to store each token?

Make the csrf a hash of the session id and the time?

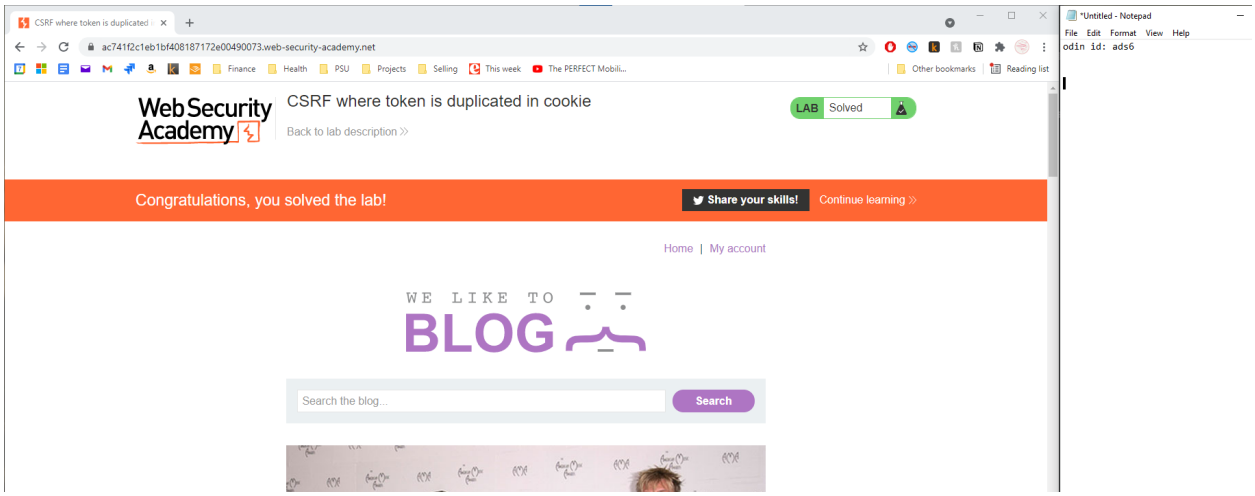
- Take a screenshot of the page that you are sent to for your lab notebook



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The csrf uses the double submit method, which the adversary may bypass by choosing any string for both the form csrf and the cookie csrf. This, combined with the improper use of unsanitized user input in the http header, permits the adversary to execute a XSS attack when the authenticated user visits a site controlled by them.

Remediation: Authenticate the csrf against something that isn't client-supplied; and sanitize the user input before incorporating it in the http header.



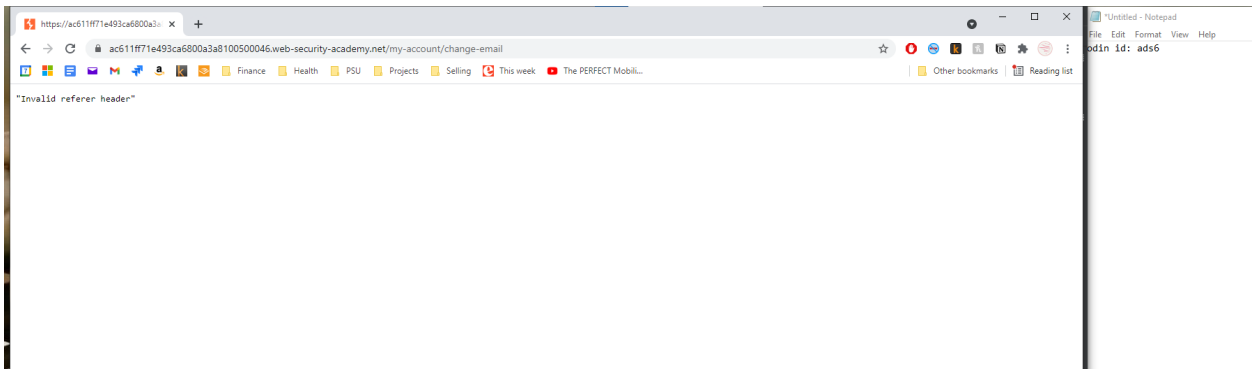
9. csrf (5)

- What status code and response text is returned?

HTTP status code: 400 with response text "Invalid referer header"

200

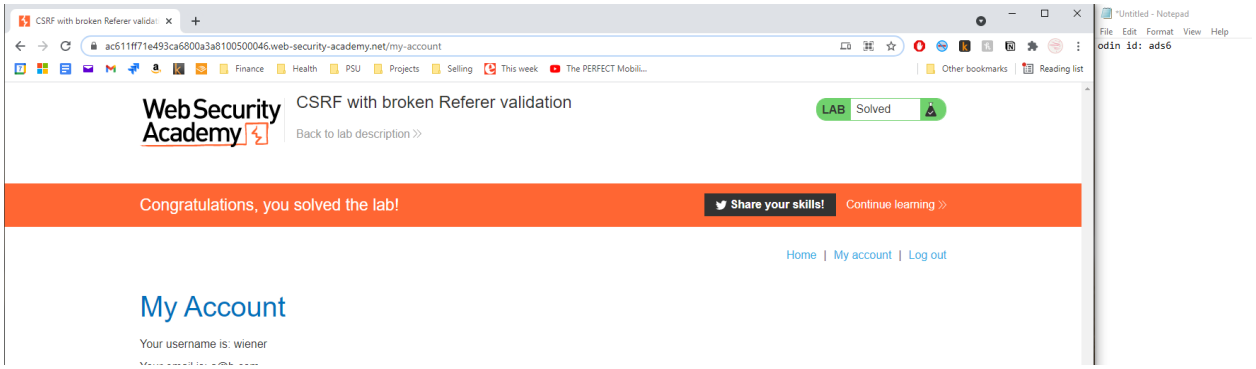
- Take a screenshot of the page that is returned including its URL



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The server tries to prevent csrf attacks using the referrer header for validation; unfortunately, the adversary may bypass this check by simply adding the site to the client's history via the history.pushState() command.

Remediation: Use a server-side hashing or storage check to validate the csrf; don't rely on client-supplied input.

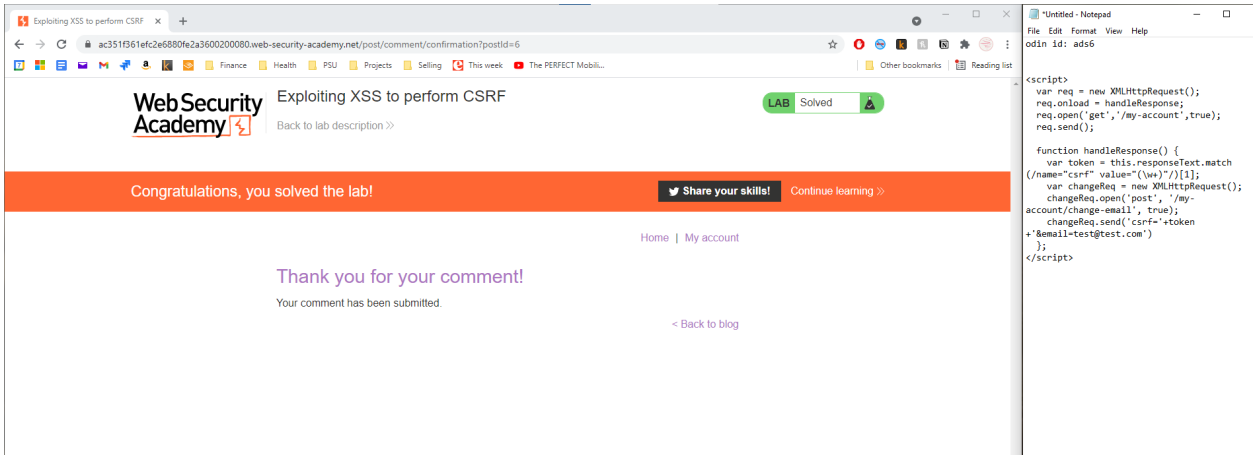


11. cross-site-scripting/exploiting

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The csrf is adequately checked; but a XSS vulnerability in the comments feature allows the adversary to embed their attack page inside the site itself.

Remediation: Sanitize user comments before incorporating them into the HTML.



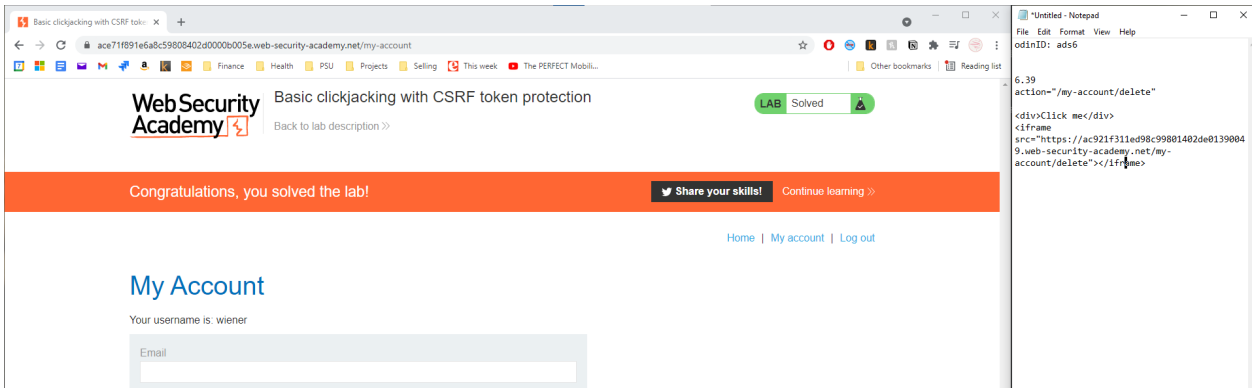
Lab 3.4

1. clickjacking (1)

- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: The adversary has embedded the target site in an iframe on a site they control, allowing them full access to the form contents before they are submitted.

Remediation: Employ the X-Frame-Options: deny option in the response header.

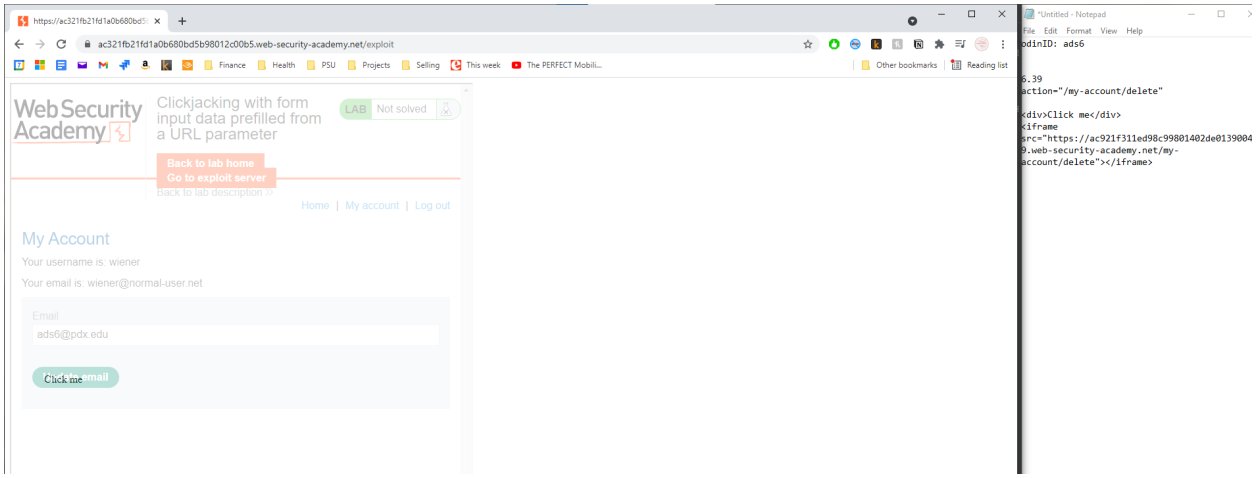


4. clickjacking (2)

- In the form's HTML, how many fields are present? Which one has been prefilled with a value?

Two fields are present; one (CSRF) has been pre-filled.

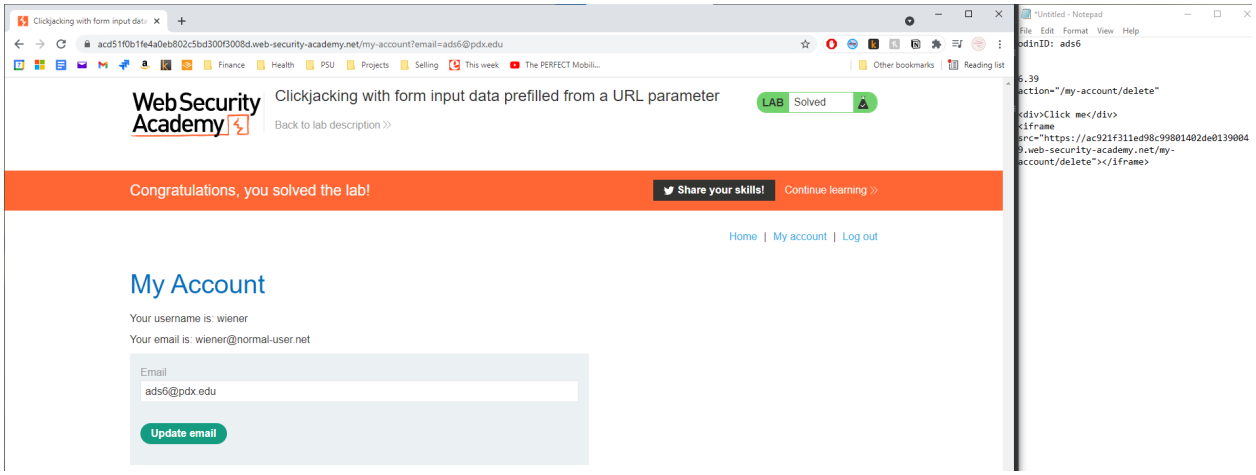
- Take a screenshot showing that the form in the transparent <iframe> has been prefilled similar to below



- Take a screenshot showing completion of the level that includes your OdinId

Vulnerability: Similar to above, but this time the adversary also exploits the inclusion of the email= query string parameter to pre-fill the victim's form before click-jacking the submission.

Remediation: Employ the X-Frame-Options: deny option in the response header.



5. clickjacking (3)

- What is the name of the HTML element that has been updated after the form has been submitted?

feedbackResult

- What is the name of the function that is registered as an event listener for when the page is loaded?

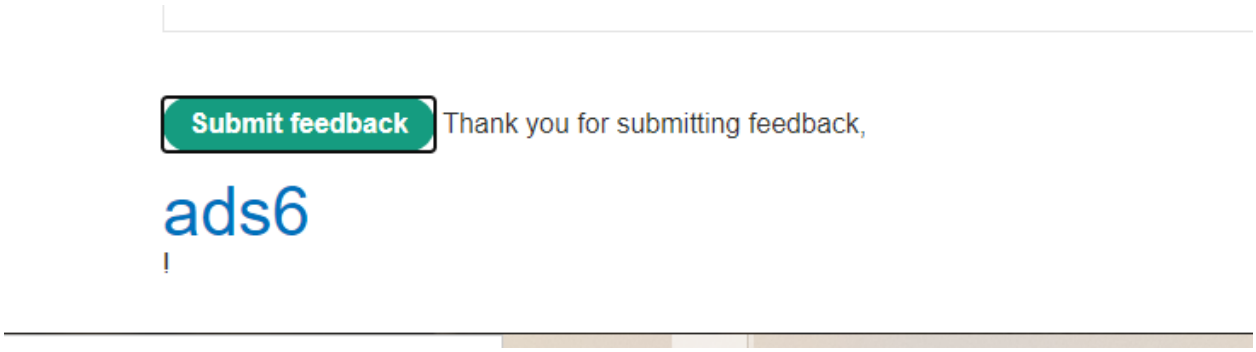
displayFeedbackMessage()

- Show the vulnerable line of code in this function and explain why it is subject to an XSS attack.

  This line of code uses unsanitized client input from the form. (I could also show the place where this function is invoked: there are two places where the client input might have been sanitized, but was not.)

```
(this.status === 200) {
 feedbackResult.innerHTML = "Thank you for submitting feedback" + (name ? ", " + name : "") + "!";
 feedbackForm.reset();
```

- Take a screenshot that includes the URL demonstrating successful injection



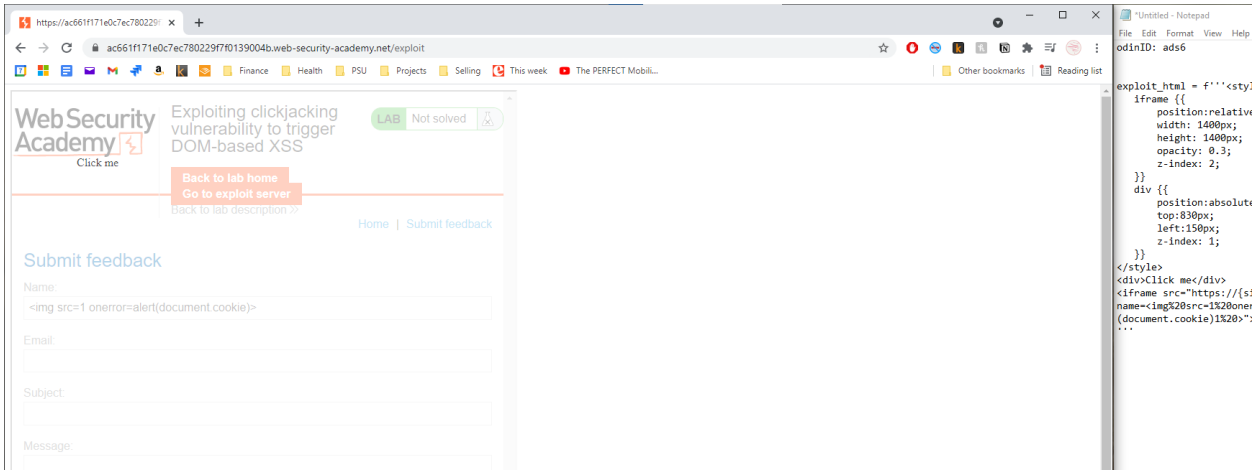
- Did the payload get successfully returned into the page?

  yes

- Did the payload execute? If not, what might be the reason?

The script was added via the DOM /after/ the page was loaded…

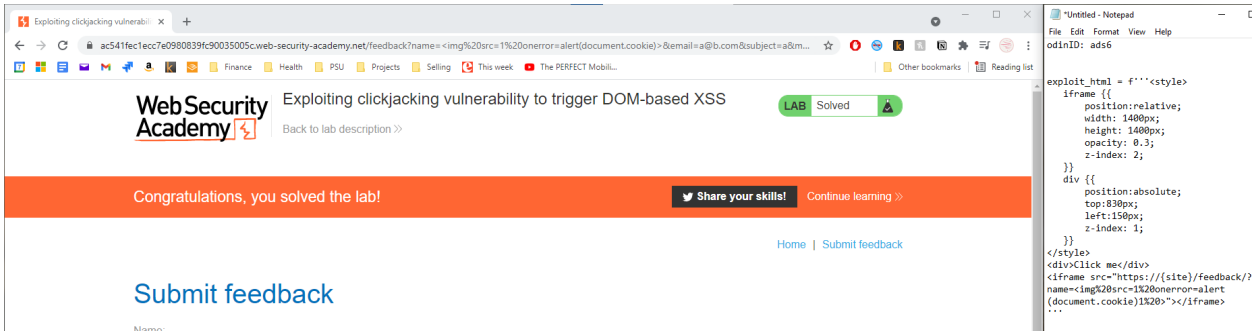
- Take a screenshot showing the location of the "Click me" element on the underlying feedback page.



- Take a screenshot showing completion of the level that includes your OdinId

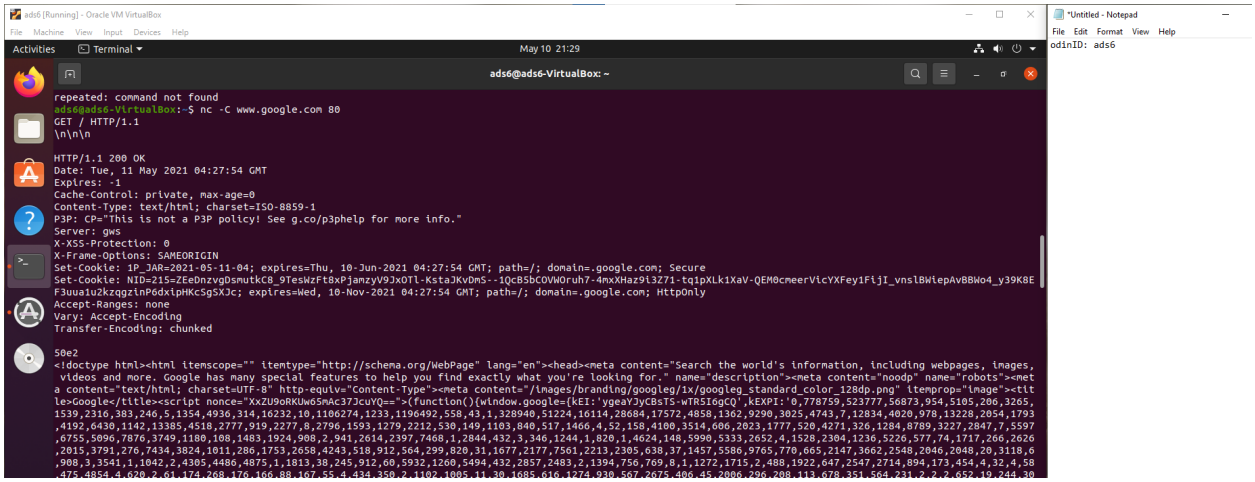
Vulnerability: Similar to above, but this time the adversary injects a cross-site script through unsanitized client input vulnerability in the comment function, allowing them to execute arbitrary code on the client when the user clicks.

Remediation: Employ the X-Frame-Options: deny option in the response header. And html-encode the user input before incorporating it into the DOM.

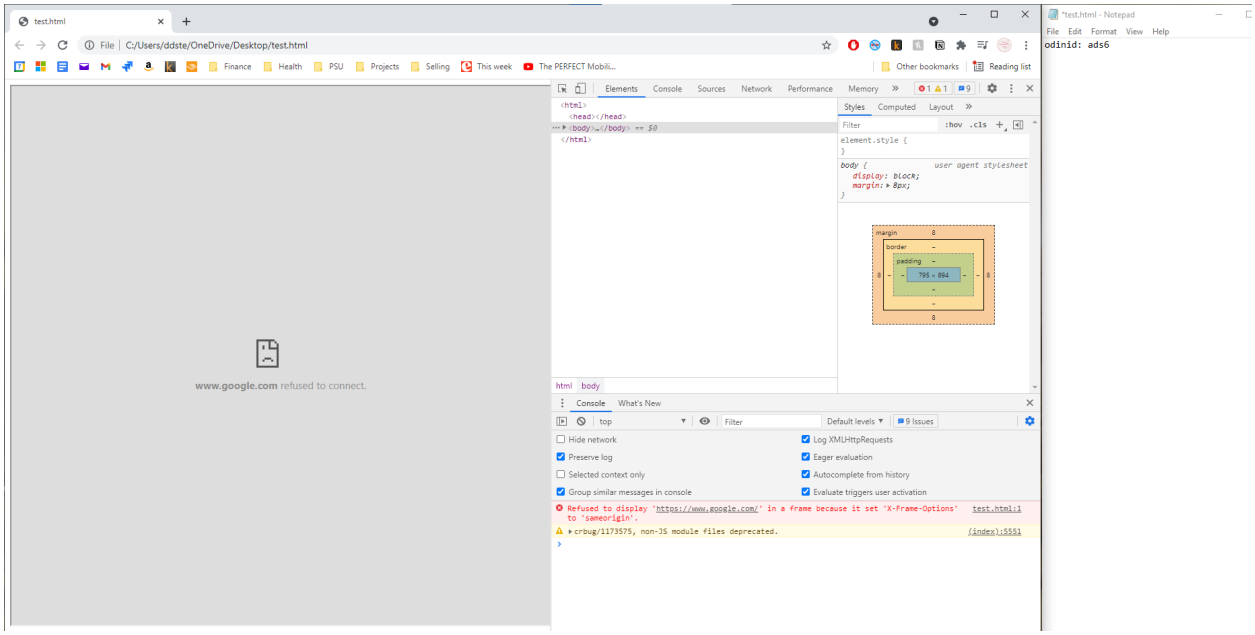


8. Prevention (X-Frame-Options)

- Show a screenshot of what Google passes back with in its X-Frame-Options: header.

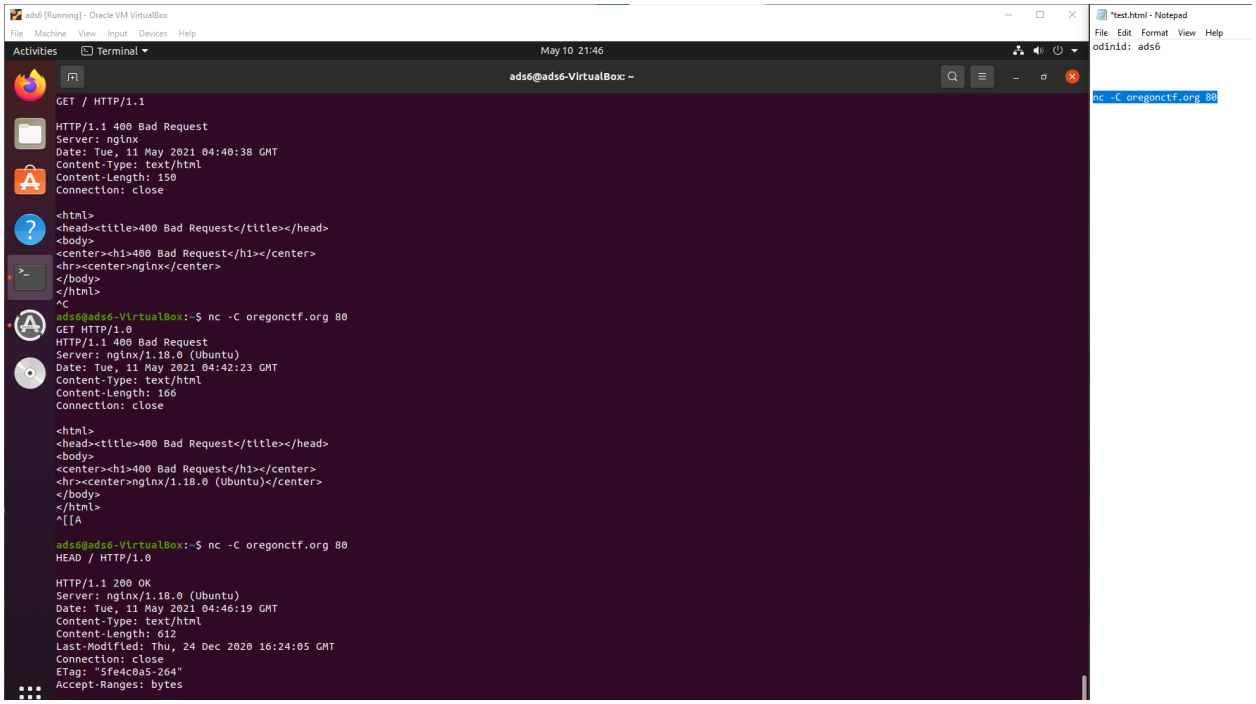


- If not, show the error in the console and explain the results.



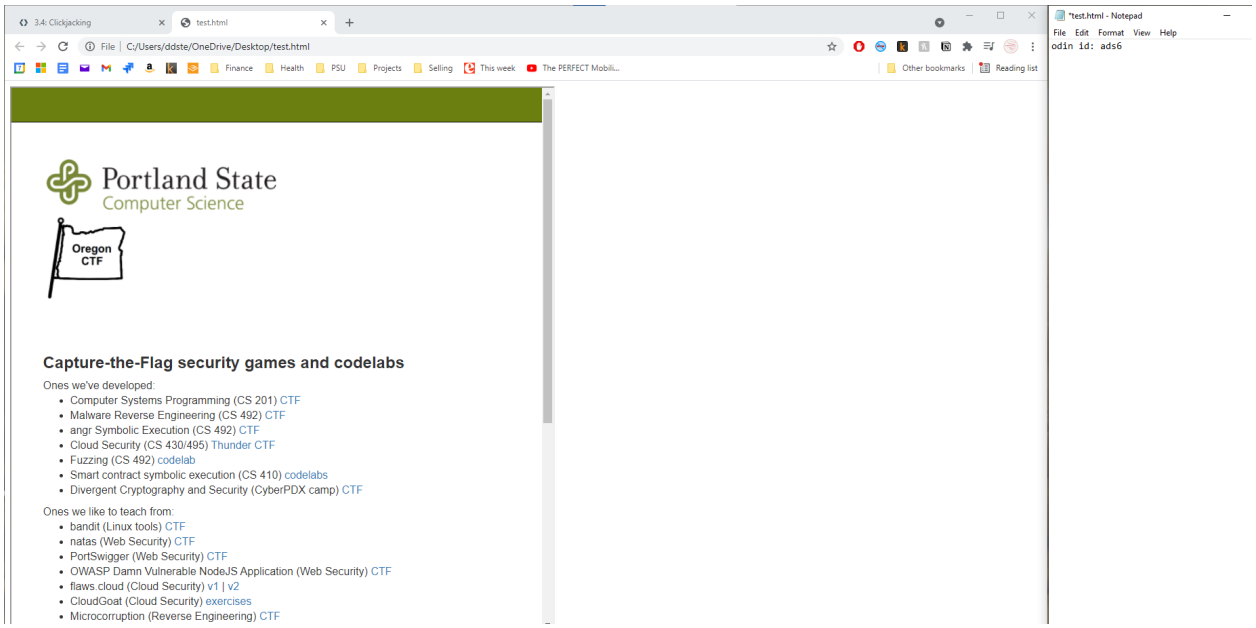
- Show a screenshot of what OregonCTF passes back. Is there a X-Frame-Options: header?

No, it does not.



- If not, show the error in the console and explain the results.

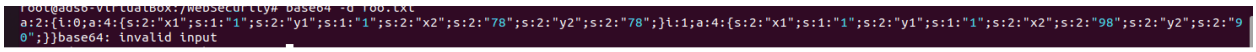
Yep, it loads…



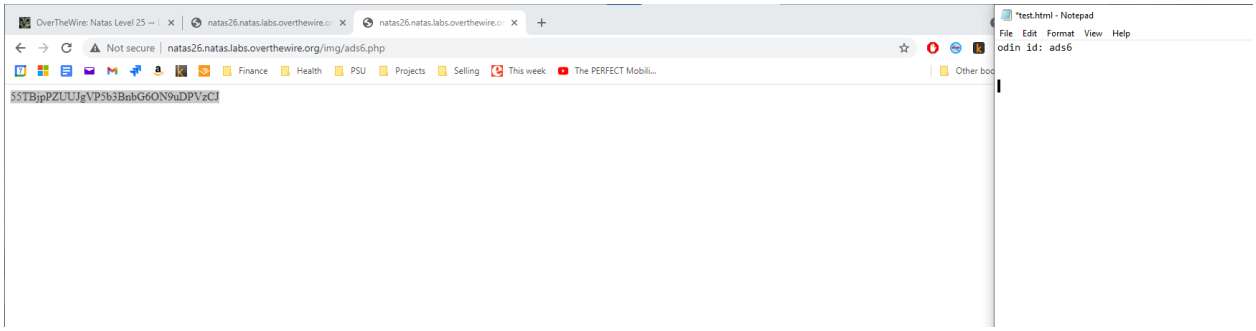
Lab 3.5

- Highlight the coordinates you entered previously via the UI in the decoded output for your lab notebook

  (highlighted in blue)



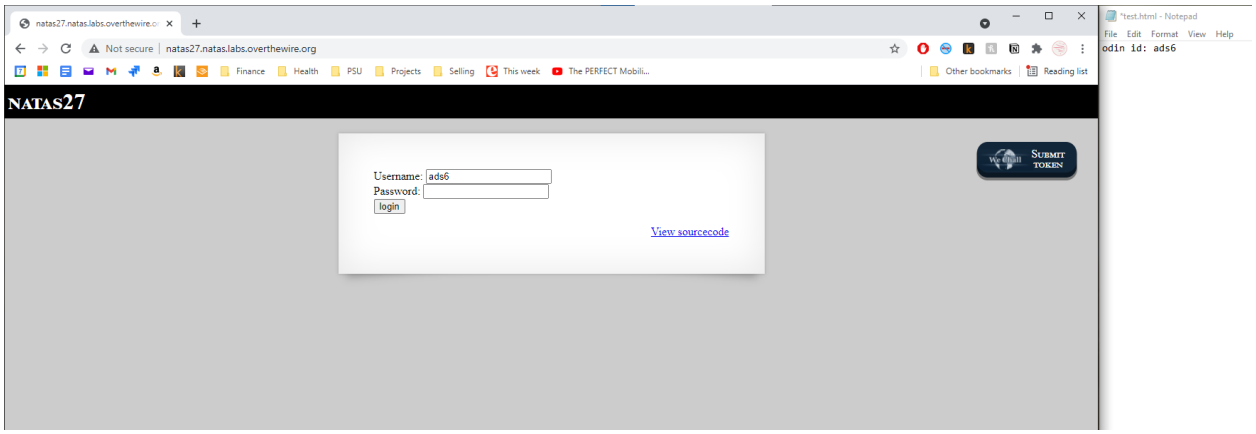
- Take a screenshot of it with your OdinID to include in your lab notebook.



- Take another screenshot with your OdinID in the Username field for your lab notebook.

Vulnerability: The cookie is deserialized before being validated, allowing the adversary to inject arbitrary php code into the flow of program execution. In this case, we demonstrate that vulnerability by forcing the OS to create a file that will print out a secret code to anyone who accesses it.

Remediation: Properly validate the contents of the cookie before deserializing it.



Lab 3.6