# Practical demonstration - Main application hacking



## Introduction

When we hunt, it's important to look at every target in it's own right. We are going to look at the OWASP juice shop. In this demonstration you will be show all the topic we went over and which parameters we will be using to test. Not all of our tests will lead to existing issues but still we **Have** to do all these tests. We are no longer practicing right now, this is bug bounties.

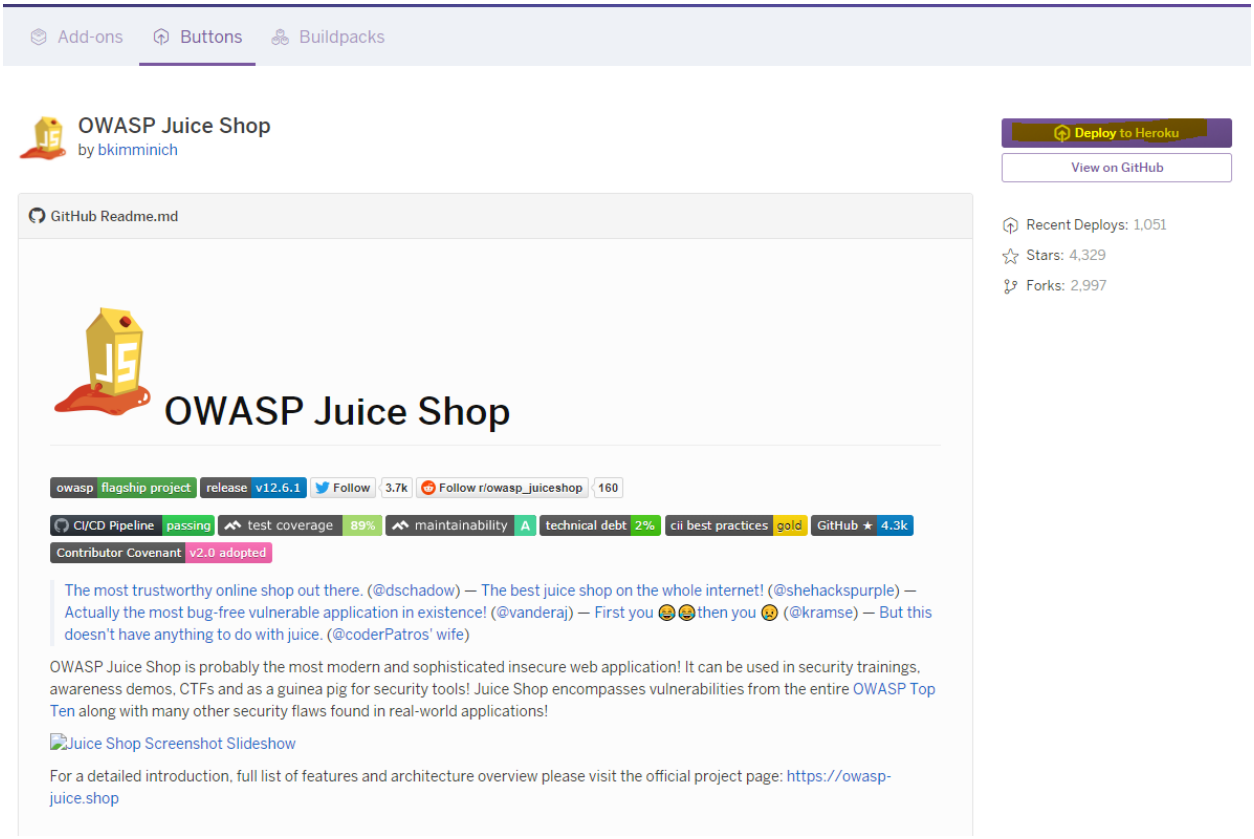
## Owasp Juice Shop

### Installation

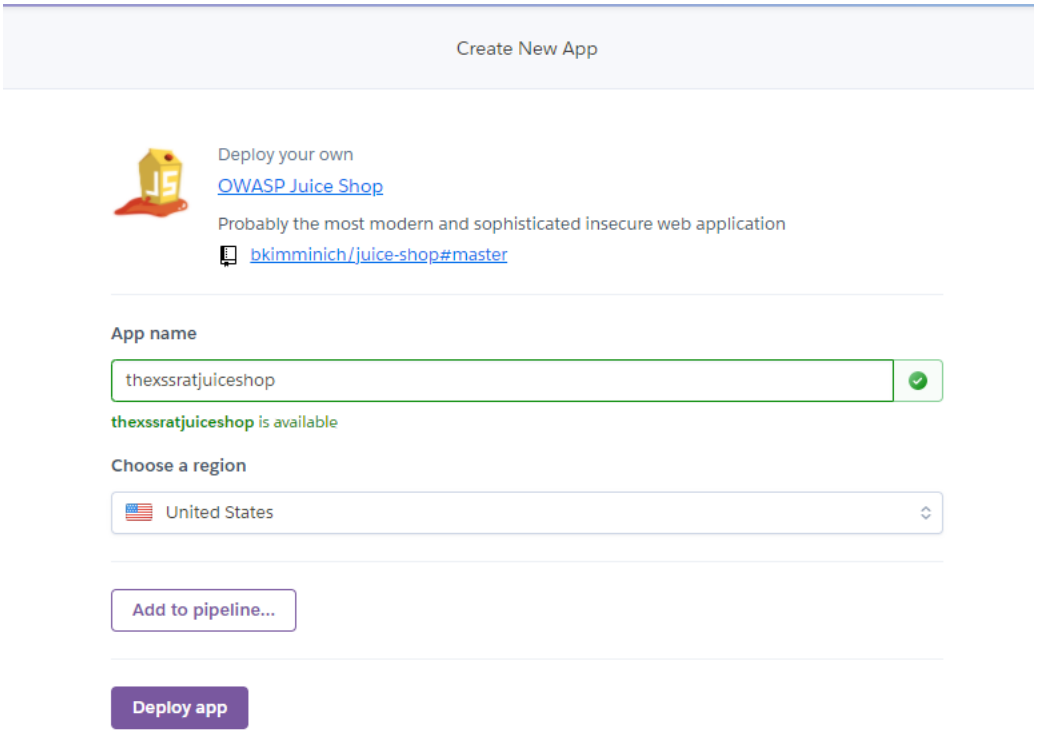
Before we can hack on the OWASP juice shop, we ofcourse need to install it first. I chose to use herokuapp for this as it's free but you can also pick the docker image.

First navigate to https://dashboard.heroku.com/apps if you need to create an account, do that first. You will now see the option to create a new app, select this and pick the option to create a new app.

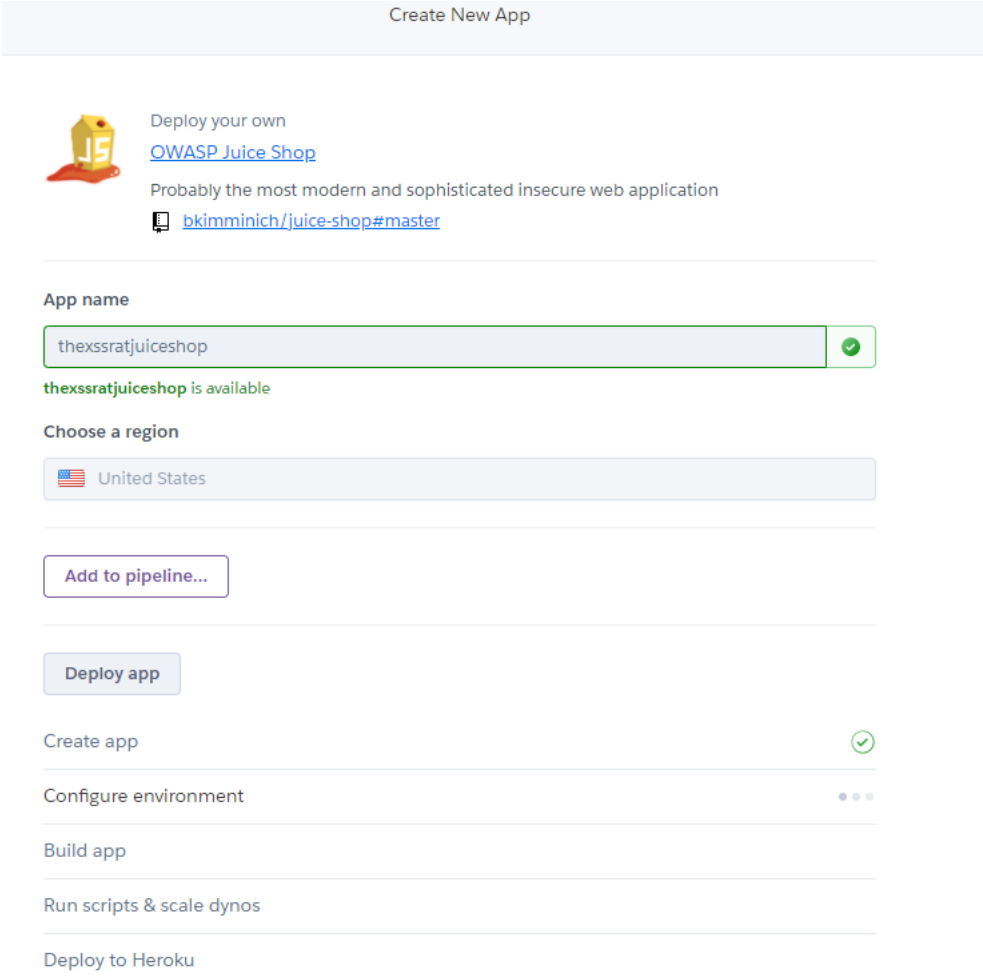
Next navigate to https://elements.heroku.com/buttons/bkimminich/juice-shop and click the "Deploy to heroku" button.

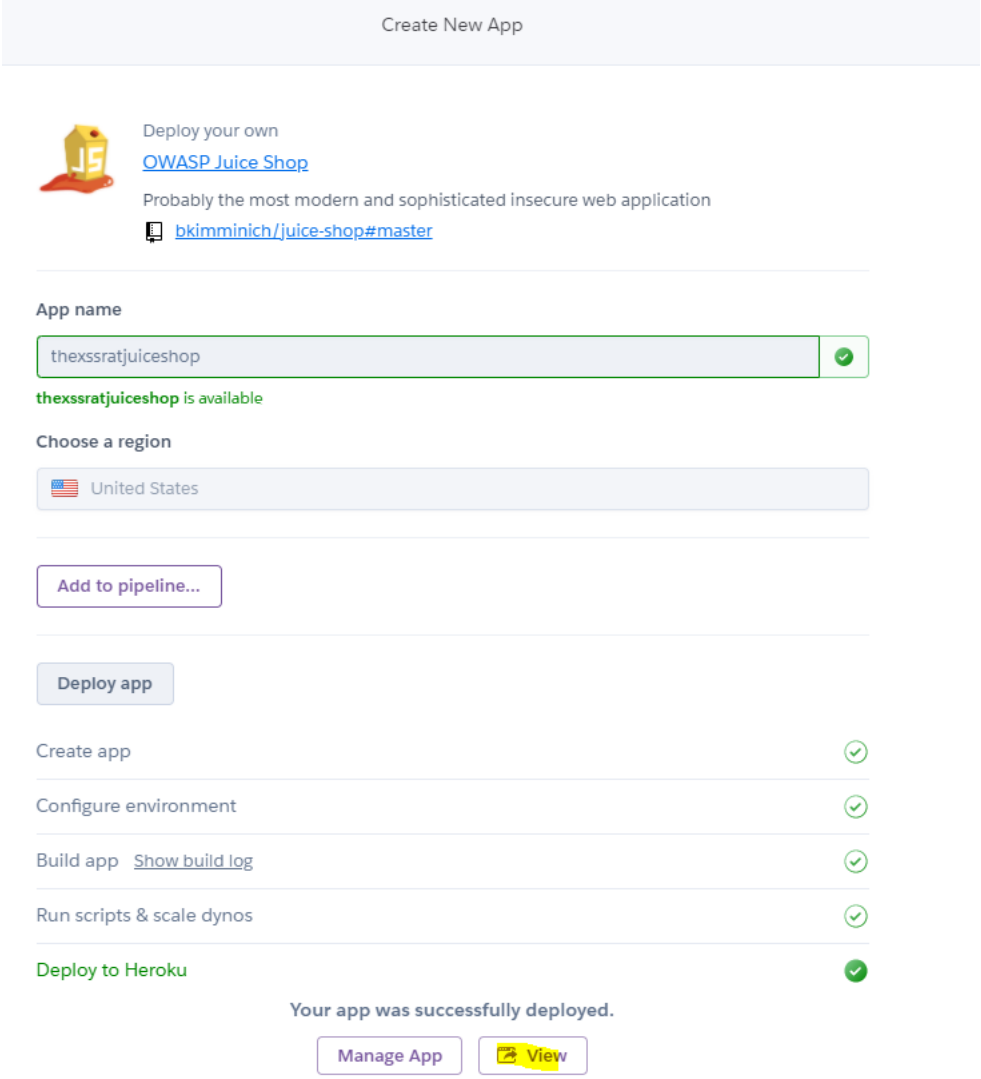



Pick a unique name for you app, it doesn't matter which it is but it has to be unique.



Click the "Deploy app" button , the app will now start deploying. It will take a while before the app is deployed so give it some take, sit back and make yourself a good coffee or tea with a nice piece of cheese.






We can now view our app 😉

## Judging our target

We recognize this is a webshop so to hack this target properly we will have to make a small investment. Since this is a webshop we want to test for at least the following functions:

- Registration
- Login
- Buying an item
- Possibly returning an item
- Our wallet functionality if exists
- Logic flaws
- XSS
  - Stored
  - Reflected

- Basket functionality
- Adresses
- IDORs
- CSRF
- Broken access control if we can get to admin functions

  This is an initial judgement and we might add to this as we explore the website and find more functionality.

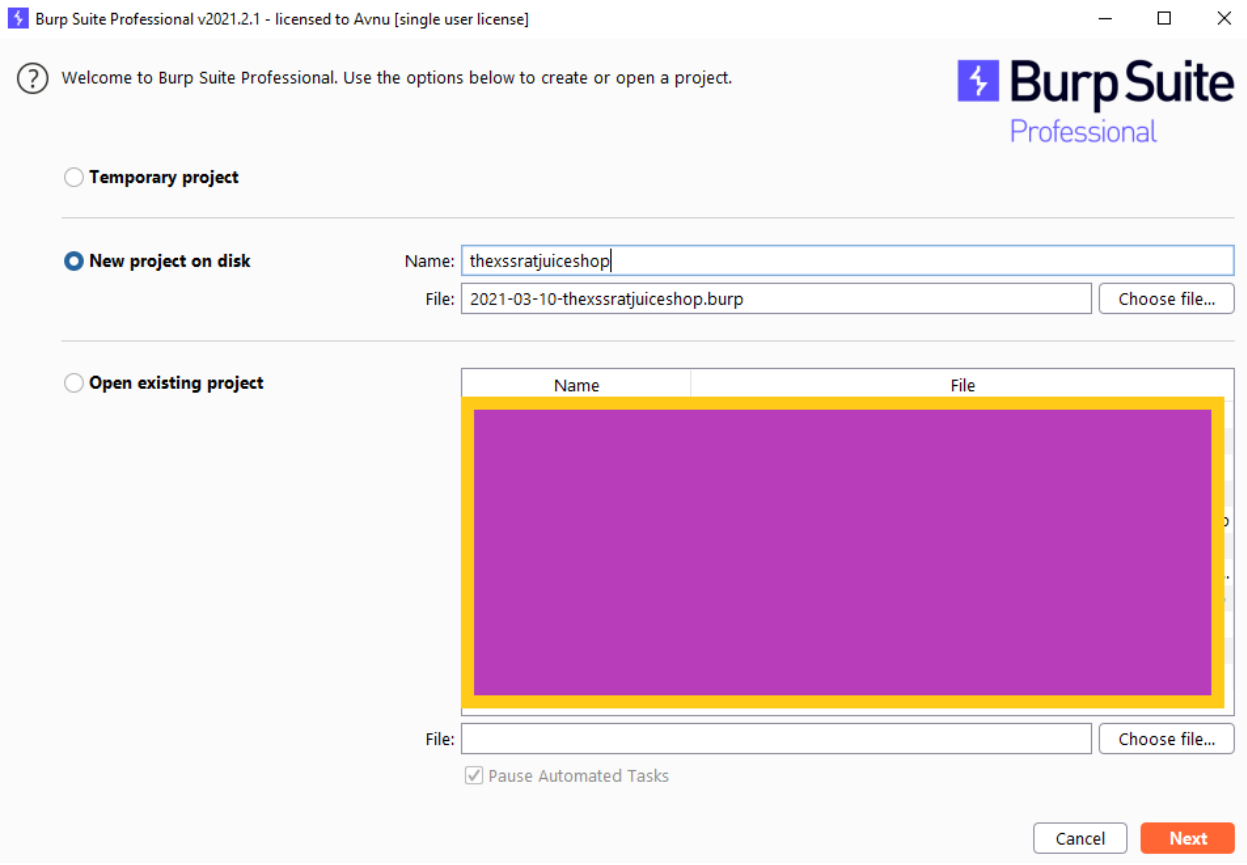
## Exploring the application

First of all we need to know what functionality exists before we can start attacking our target properly. We need to fill up our site map in burp and we need to be able to epxlore the paramatirised requests. To do this, we need to set up burp properly first. This includes setting up our scope and setting the options that we need. In this course we will use burp suite but feel free to use any other MiTM proxy with the same functionality.

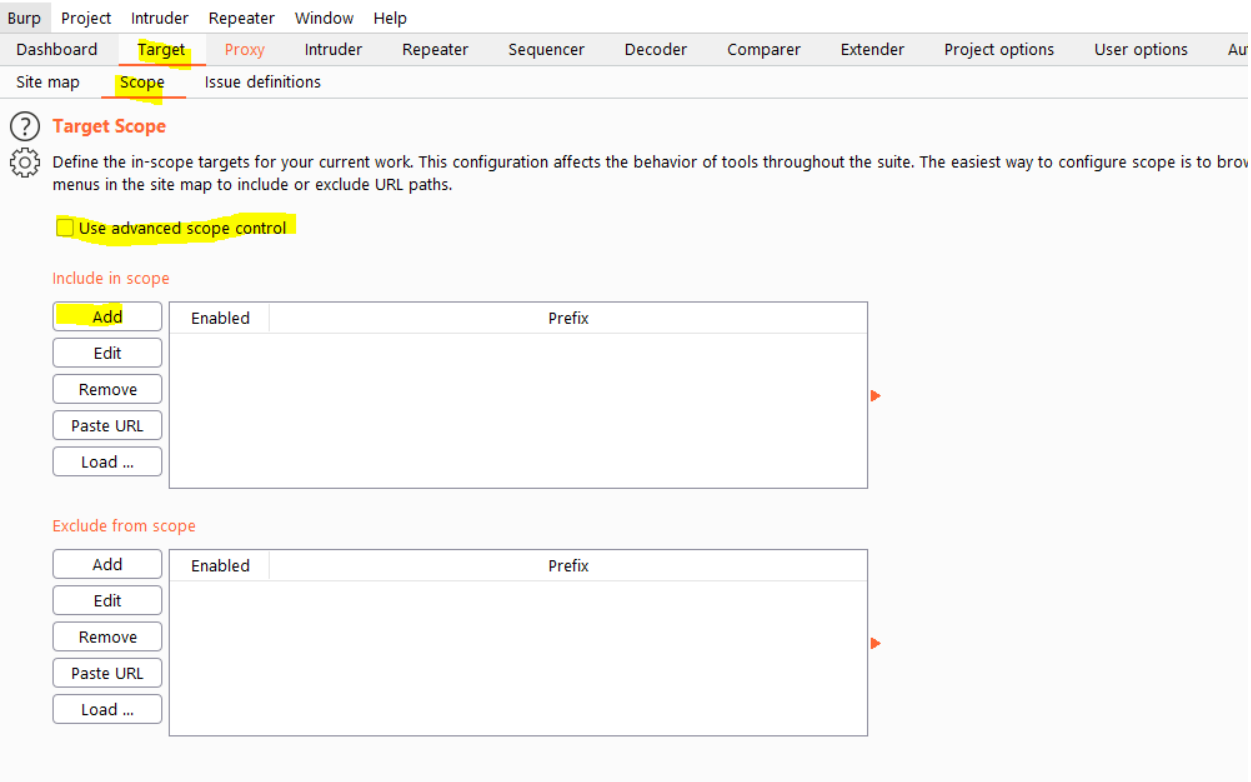
### Setting up burp suite

It really helps to have burp suite pro, you don't have to but the fact that you can save a project is a major plus for me. I can only hunt in bursts of 1 to 3 hours so i have to revisit my target often. This means two things.

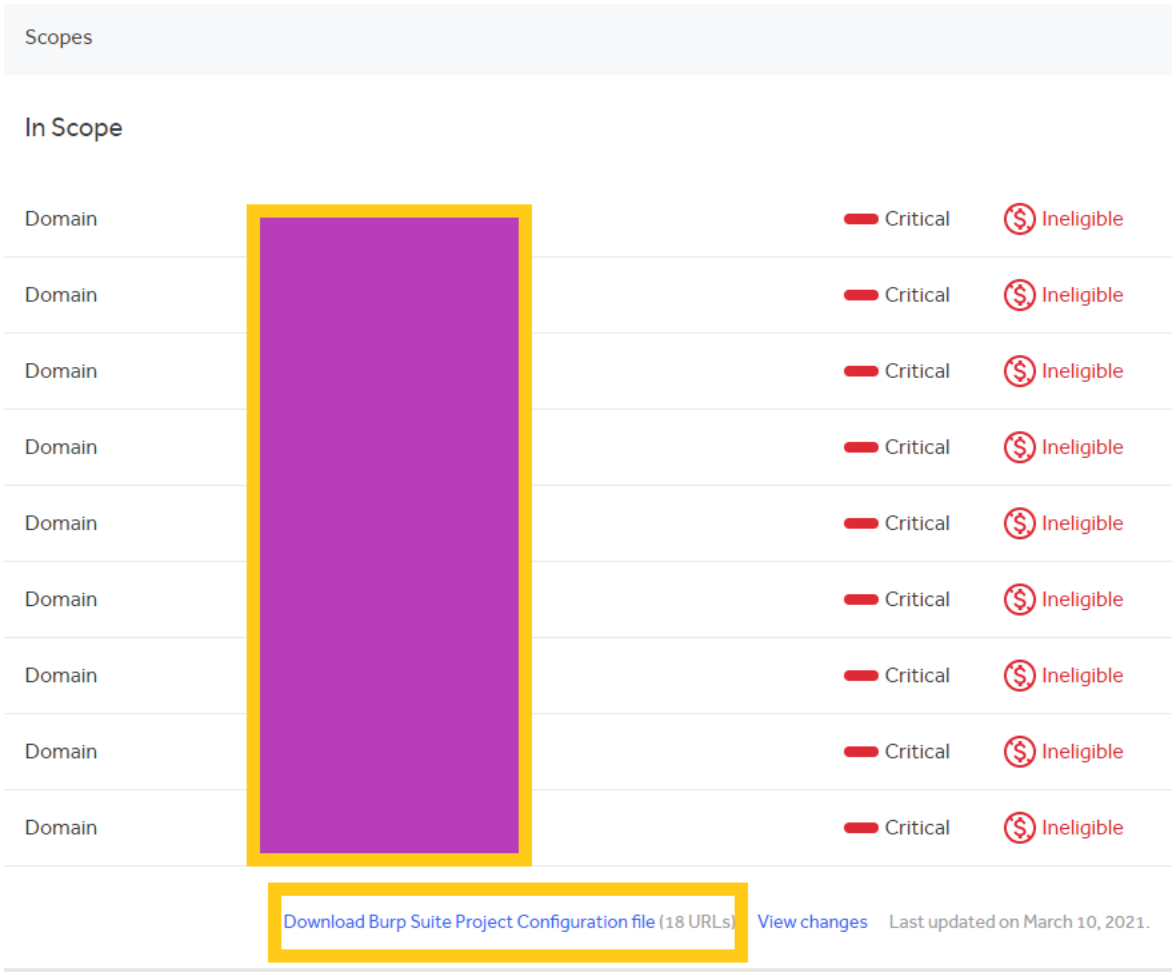
- I have to take very dilligent notes so i don't retest things 10 times needlesly and so that i make sure i do test all of my functionality. Part of this documentation is the "Judging our target" section.
- If u can set up my project settings in burp suite, save them and reload them whenever you want, that is a major plus. The biggest part of any activity is getting yourself to do it and if you can skip part of the setup, that will help you get started. If you are doing something time seems to fly but if you are sitting in your sofa it takes tremendous power to get yourself up and go hunting. Anything you can do to make this easier is a major win.



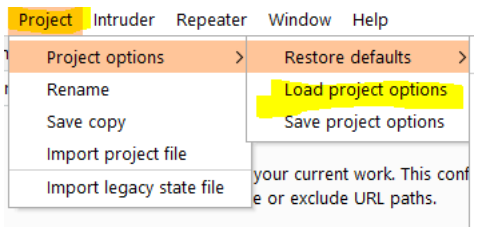
First i like to setup my scope, make sure you add the propper URLs that are in scope.

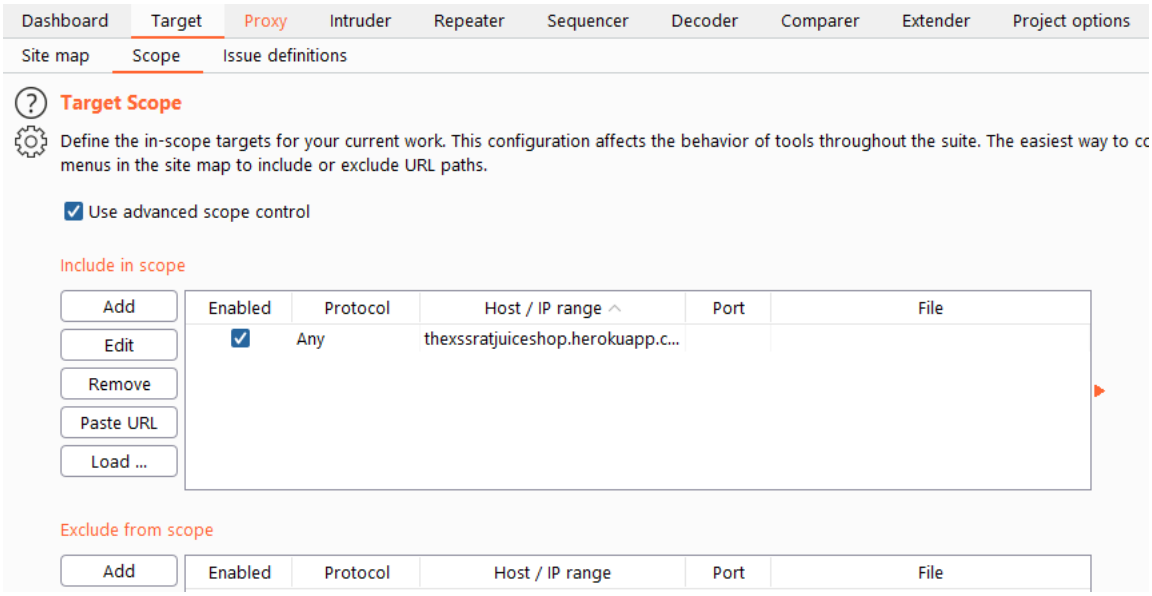



Hackerone has configuration files for burp you can download.

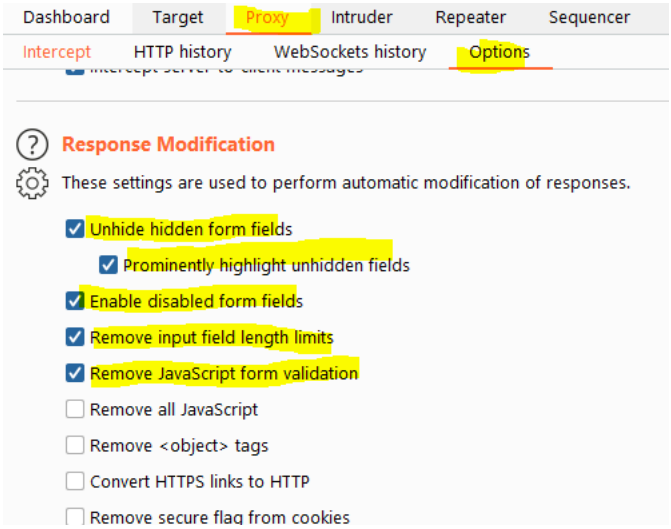



You can then import this file via the project options import functionality under "Project > Project options > Load Project options"






After setting up our scope we will move on to setting up our proxy options. I always configure several options to make it easier for myself to see things like hidden fields and to remove any javascript validation from my responses.




This will make any hidden fields easier to see as it will unhide them and draw a big red square around them. If you see this big red square you know you are looking at a hidden field.


Now that we have burp suite set up in the background we can start exploring our application.

### Manually walking the application

Just because we are manually walking the application does not mean we should not be hacking. This is the most important phase in bug bounties and most of you will know it as the recon phase.

In this phase we want to get to know our application. We want to start by exploring the functionality and as we do that we want to take note of our prvilidge levels. Even though it might not seem like it, since we don't have access to the admin functionality (yet), but there are different levels of priviledges.

- Unauthenticated accounts (not logged in)
- Authenticated accounts

As we hack our application, there might be more levels we can add to this such as administrators.

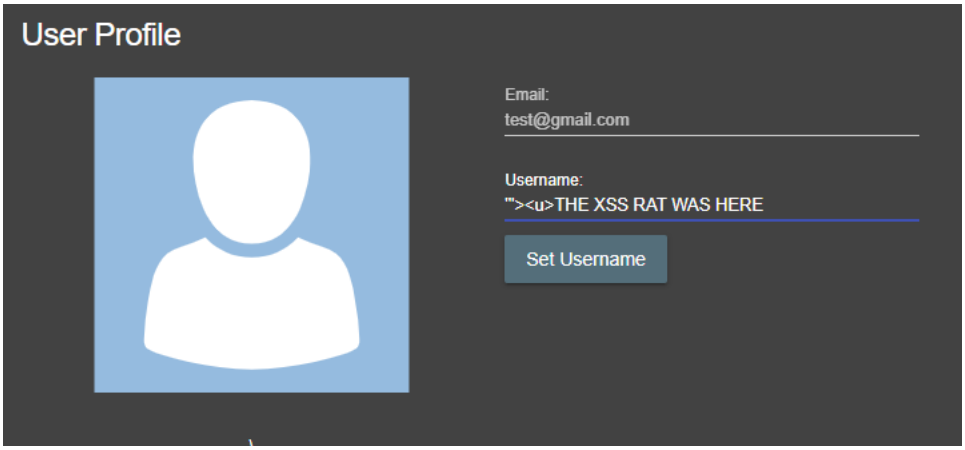
I myself use excel to make a quick mindmap but you can use whatever tool suits you best.

| | Authenticated | Unauthenticated | Admin?? |
|---|---|---|---|
| View items | | | ? |
| Add items to cart | | | ? |
| Buy items | | | ? |
| View reviews | | | ? |
| Add reviews to items | | | ? |
| ,,, | ? | ? | ? |

When i register my account, I register using an attack vector that automatically tests for JS XSS, HTML injection and HTML tag attribute injection.

```
'"><u>THE XSS RAT WAS HERE
```

I use this attack vector wherever possible when attacking my target.

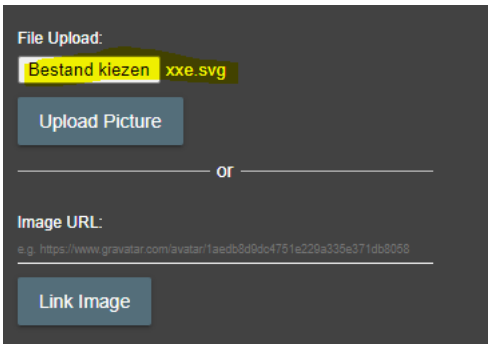



This way, whenever the application uses my username anywhere, i am automatically testing for all of the described attack vectors. If my username is reflected in the JS context anywhere, the "' will try to break out of the context it is being reflected into. If the username is reflected in a tag, like it is in the picture above. In this picture, when we save the username, we can see that we have broken out of the VALUE attribute of the input tag. From here, we can try to insert our own javascript.

```
<input class="form-control mdl-textfield__input" id="usernam
e" type="text" name="username" value="'">THE XSS RAT WAS HER
E" style="color: #FFFFFF;" placeholder="e.g. SuperUser" aria-
label="Text field for the username"> == $0
```

I will try this WHEREVER i can, this includes adresses, nicknames, ...

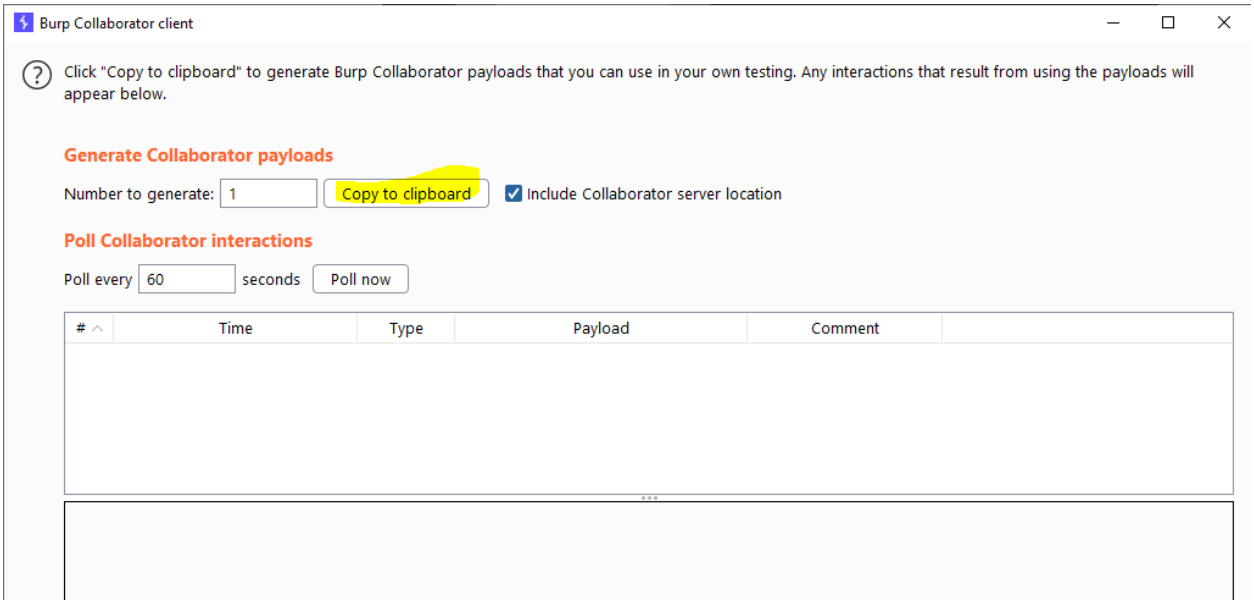
On the profile page however i can still see some more things i can test for. I see a profile picture so i can test for XXE via SVG here.




I also see a link option here. This link will resolve to a picture which gives the option for SSRF. To test for this, i start my burp collaborator and grab a URL that i can insert into this field. Running a public burp collaborator server is a premium option and only available in the paid version of burp. If you don't have the paid version of burp you can use:

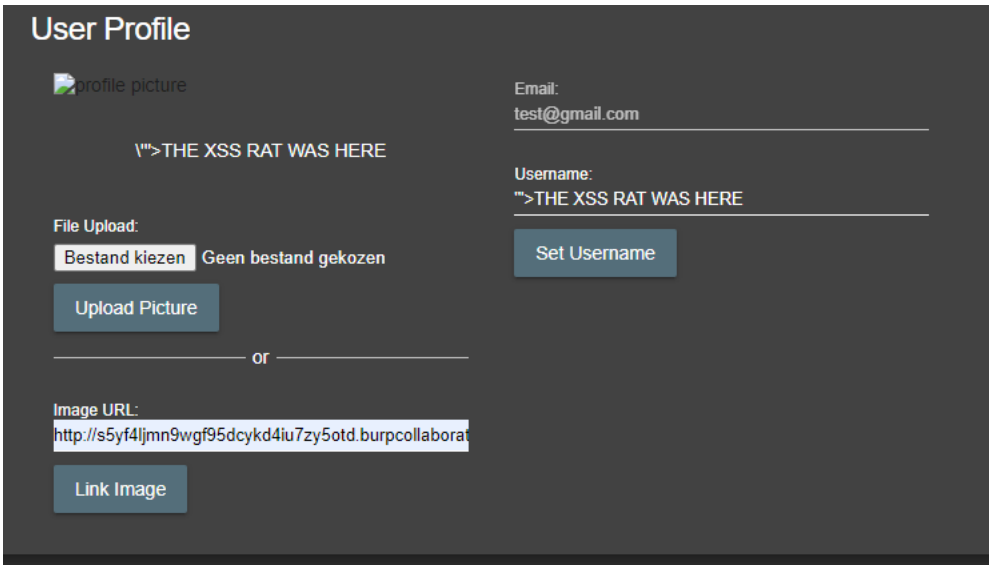
- Your own webserver
  - If you don't configure this properly, you can only capture HTTP requests
- A public burp collaborator
  - This may suffer from availability issues as it's shared between all users

    Whichever option we go for, we need to copy our payload to the clipboard and paste it in our URL field that the server tries to resolve.

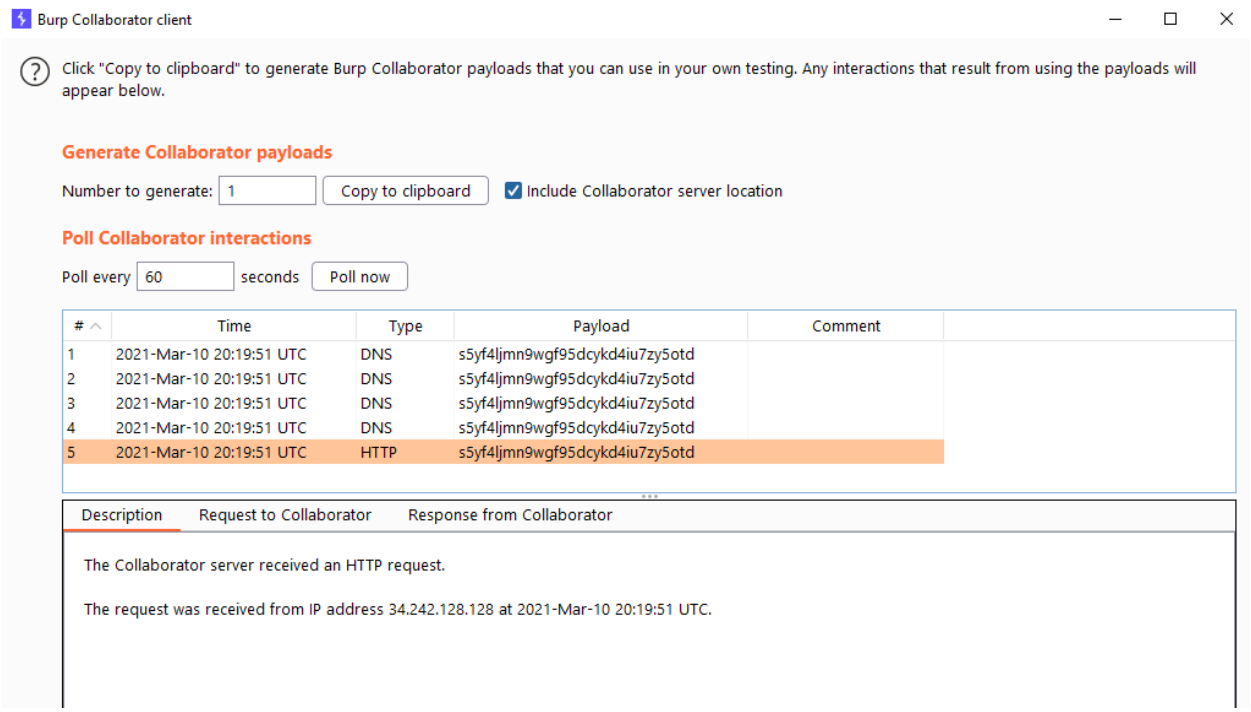


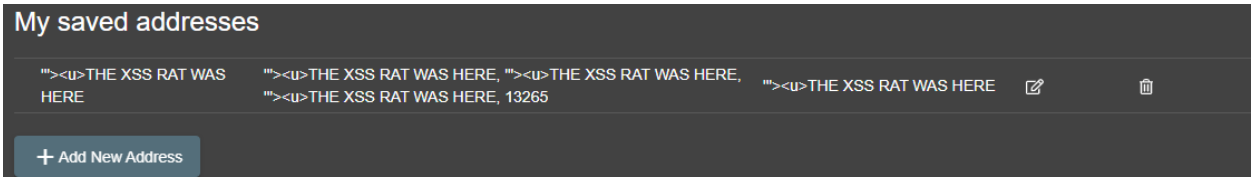

We might need to put HTTP:// in front of our URL if the server checks for syntax.

http://s5yf4ljmn9wgf95dcykd4iu7zy5otd.burpcollaborator.net




If you a lot of DNS requests coming into your burp collaborator but no HTTP requests, than there probably is no way to pull off SSRF is there is a possible egress filter in place. Egress filters can stop certain types of outgoing traffic. As of this writing OWASP juice shop does not have any SSRF vulnerabilities but if we'd find one here, we would continue on our SSRF path.



I will try the same XSS technique for adress any stored or reflected fields i can find. If i suspect a server side template engine is being used, i will add an SSTI attack vector.

```
'"><u>THE XSS RAT WAS HERE${7'*7'}
```

if this resolves to 7777777 or 49 i will investigate further into SSTI or CSTI attack techniques.

### SQLi

The previous attack vector also automically tests for SQLi as well

```
'"
```

These special characters will also test for SQLi since these are special characters also used in SQL statements. If we get a SQL error, i usually run a tool like SQLmap to better investigate what SQLi i have on my hands if any. See SQLi section.

### Testing for IDORs

To test for IDOR's i will have to create a second account. In this case i can test for IDORs in several ways. First of all i really need that second account. I am going to copy his session cookie or authorisation header. This may seem confusing as you can't copy your victims cookie or headers in production **but** this is not our goal. We just need that cookie or header to automate our IDOR search. How to do this has been demonstrated in the tools section.

### Testing for vertical priviledge escalation

When we want to test for vertical priviledge escalation we do need accounts of different priviledge levels. Since we don't have any admin accounts yet, we can't test for this yet. If we did have different priviledge levels, we would create accounts of all different priviledge levels and test for BAC. Example:

- Administrator
- Content editor
- Customers

Again, how to test for this specifically has been illustrated in the tools section.

### The login section

In the login section we can play around with the requests a little bit. For example if we request a password reset link, and append our own email adress to the request, the server might send the password reset link of the victim to the email adress of the attacker.

```
POST /api/resetPassword HTTP/1.1
Host: ferretshop.herokuapp.com
Connection: close
sec-ch-ua: ";Not A Brand";v="99", "Chromium";v="88"
Accept: application/json, text/plain, */*
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTcsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://ferretshop.herokuapp.com/
Accept-Encoding: gzip, deflate
Accept-Language: nl-NL,nl;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: language=en; welcomebanner_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6M
Content-Length: 24

{email:"victim@gmail.com"}
```

Might turn into

```
POST /api/resetPassword HTTP/1.1
Host: ferretshop.herokuapp.com
Connection: close
sec-ch-ua: ";Not A Brand";v="99", "Chromium";v="88"
Accept: application/json, text/plain, */*
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTcsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ0
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://ferretshop.herokuapp.com/
Accept-Encoding: gzip, deflate
Accept-Language: nl-NL,nl;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: language=en; welcomebanner_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6M
Content-Length: 24

{email:"victim@gmail.com",
email:"attacker@gmail.com"}
```

This might prompt the "GenerateLink" server to generate a password reset link for the first email address but the "SendLink" server might send the link to the attacker.

### CSRF

Whenever i see a CSRF token, i will try to replace it:

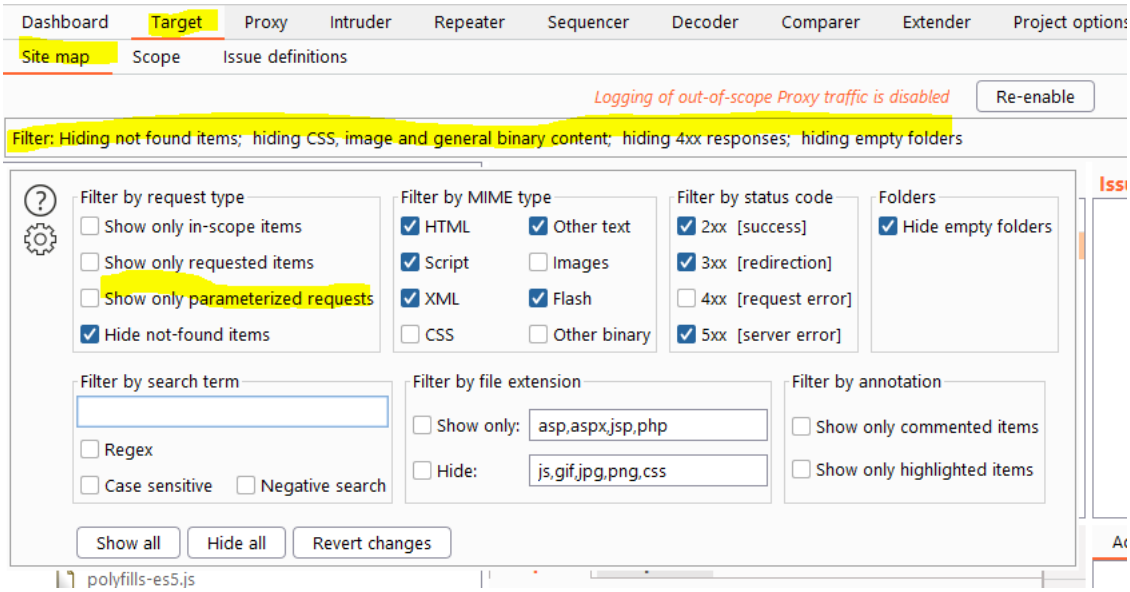
- with an empty parameter (CSRF=)
- with a parameter of the same restrictions (like same length and alphanumeric) (CSRF=2475455dfs1)
- CSRF=1
- A CSRF token that does not belong to that account

I can use the tools "Match and replace" or "Autorepeater" for this although the "Autorepeater" extensions seems to have broken with the latets burp update though this might get fixed later on.

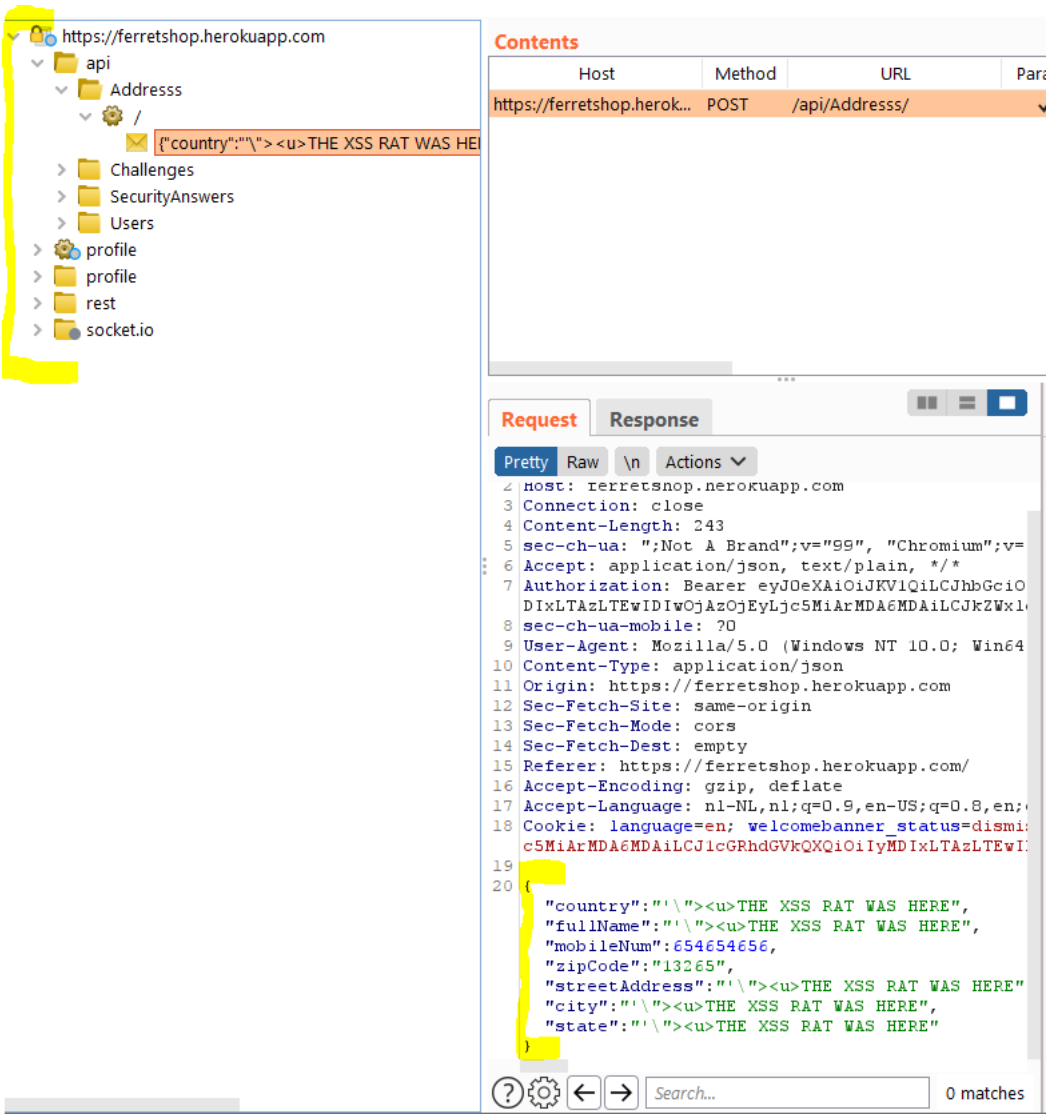
See the tools section.

### Exploring the requests

Now comes the fun part, we are going to look at all the requests and parameters in their own right. To do this we need to go back to burp suite and look at our site map. This has been filling up in the background while we click around.

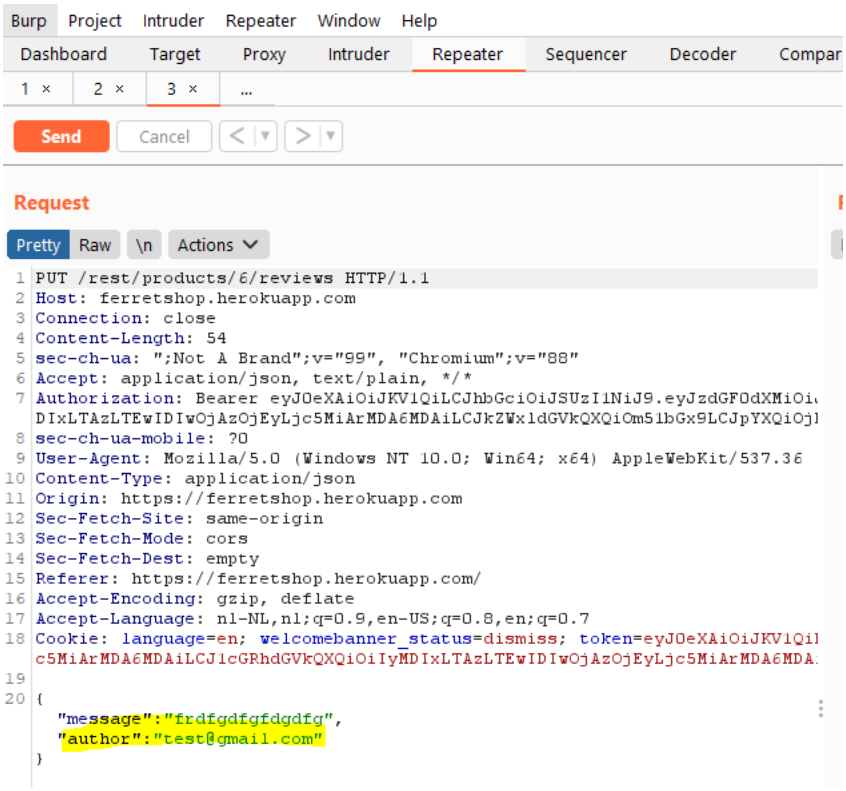


Now we can see the parameterised requests, we don't really care about the static requests.




I am going to look at all of these requests and see if i can find some parameters i can manipulate that i should not be able to manipulate such as:

- {userType:"User"} > {userType:"Admin"}
- {accountType:"Basic"} > {accountType:"Advanced"} (might be more expensive)
- {rating:5} > {rating:-5000} (might be able to negatively affect the rating of a video on youtube)
- ... Use your imagination




In this case for example, we can change the author of a review which should not be possible.

### LFI/RFI

Whenever a parameters shows that it is grabbing a file from the local file system or whenever it seems to be from a remote location, i will try either LFI or RFI respectively. See those sections to learn more about them.

- GET /avatar.php?file=image1.png
- GET /avatar.php?file=s3.bucket.org/image1.png

LFI/RFI in and off- itself is usually not that impactful. If we find this issue we should try to find files on the system that we should not be able to access like private pictures of other users or we should try to include a file which executes remote code execution. This can cause use to create a reverse shell allowing us access on the server. If you ever achieve this, you should stop and report, don't explore a production server with the risk of seeing data you should not or crashing the production server.

### OS Command injection

This is the last vulnerability type i check for. The only way to check for this issue type is to fuzz all of those parameters with your fuzzing list that you created in the OS command injection section. To do this I use the intruder tool that's built into burp.