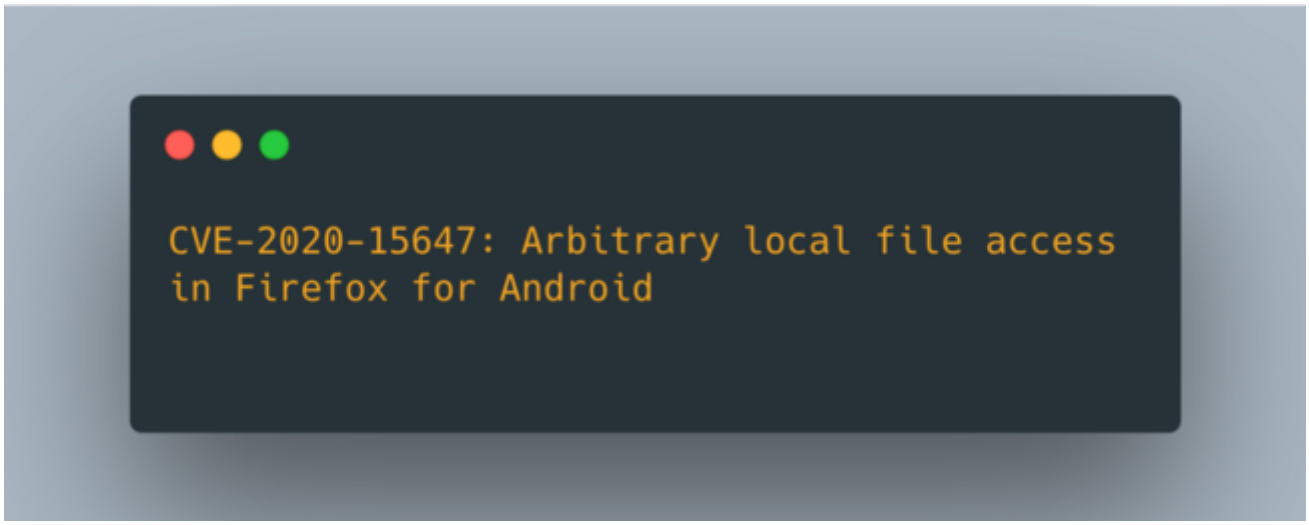
Firefox: How a website could steal all your cookies

Pedro Oliveira

1 day agoÃ‚Â·6 min read



This is a write up for CVE-202015647, explaining how webpages are capable of stealing files from your Android device, including but not limited to cookies from any visited website.

Introduction

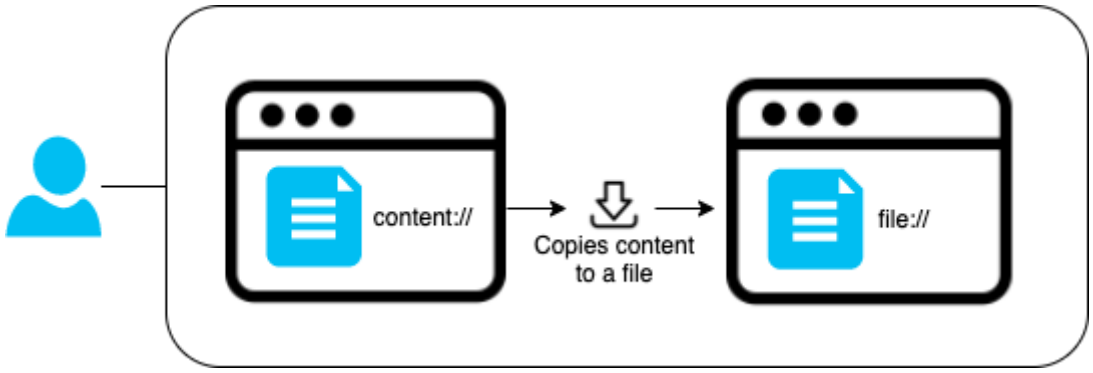
In mid-2020, I started checking Android browsers for multiple types of vulnerabilities; while reviewing v68.9.0 of Firefox for Android, I noticed it displaying strange behaviour when browsing `content://` URIs.

For context, Content URIs in Android identify data in a content provider; they can represent multiple forms of information, such as files or database information.

Most browsers support the parsing and processing of both `file://` and `content://` URI schemes. If you try to open a local HTML file in your browser, it will most likely use a `content://` URI created by the file browser you used when opening the file.

Testing the content:// URI

When I tested Firefoxs use of `content` URIs, I noticed the address bar was changing while rendering the URI, redirecting me to a `file://` URI. It appeared that Firefox was saving the content to a file, and then redirecting me to that created file - the file was being saved in the internal temporary folder `/data/data/org.mozilla/firefox/cache/contentUri/`.



Firefox file download/redirect routine

I also noticed the file that was created had the same name as the display name (`_display_name`) returned by the provider, with Firefox not changing the name (and therefore overwriting the file) if it already existed.

Permissions

The thing with content providers is that generally, applications require specific URI permissions in order to fetch contents from other applications and providers. This prevents applications from accessing files from other providers unless they have been explicitly granted permission (which is the case when tapping Open with or Share with and choosing an application to access a file). However, an application does not need to go down this route when accessing URIs from their own content providers.

Firefox had a file content provider whose authority was `org.mozilla.firefox.fileprovider`, and had the following configuration:

```
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <root-path name="root" path="."/>
</paths>
```

This is an issue, due to the use of the root-path configuration; I could open virtually any file in Firefox, provided it had access.

Showing contents of a private Firefox file

Now that we knew we could open any file, lets try to expose file contents to the outside.

Taking advantage of SOP for file://

The way Firefox deals with cross-domain requests for `file://` is pretty well documented. For the purpose of this exploit, the only thing you need to know is that a file can access its own contents (i.e. via `new XMLHttpRequest().open("GET", window.location, true)`) because the origin is the same.

Now, if a file can request its own contents, and these contents can be replaced, I might be able to spoof the contents of a file with another file using the same name, right

I started off with simple testing I needed to retrieve the contents of a private file by opening a file from the external directory. In this case I chose `/data/user/0/org.mozilla.firefox/files/mozilla/profiles.ini`; this file contains information on where the cookie database is stored in the device.

To retrieve this file, I need to create a file with the same name, saving it in `/sdcard/Download/profiles.ini`. Like in the first paragraph, I'll be using `XMLHttpRequest` to retrieve `window.location`.

```
/sdcard/Download/profiles.ini
```

To load this script, well also create an iframe in the same file which loads a `content://` URI pointing to the file we are actually trying to read; by opening with a `content://` URI, we will leverage Firefox's copying of the file to another location and accessing it via `file://`.

```
/sdcard/Download/profiles.ini
```

> Showing contents of /data/user/0/org.mozilla.firefox/files/mozilla/profiles.ini

To summarise the above, by opening the file weve created:

1. Firefox opens `content://org.mozilla.firefox.fileprovider/root/sdcard/Download/profiles.i`, which is the content URI representation of the file `/sdcard/Download/profiles.ini`
2. Firefox fetches the content given by its provider and saves it in `/data/data/org.mozilla.firefox/cache/contentUri/profiles.ini`
3. Firefox redirects the user to the file in step 2, prefixing it with the `file://` URI
4. The HTML page, once rendered, has the iframe request the `content://` URI in its `src` field, which has the same name as the currently opened file (albeit in a different location), `content://org.mozilla.firefox.fileprovider/root/data/user/0/org.mozilla.fi`

5. Firefox replaces the contents of the file opened in step 3 with the contents of the file in step 4
6. After a short delay, the page alerts its own contents (which has now been changed to the contents of the other file in step 4

We now have a cross-domain issue, as a `file://` URI can access contents from another `file://` URI if the files share the same names. Furthermore, because we are using Firefox's provider; that provider's `root-path` configuration means we could exploit this to access any file on the device.

Escalating to remote

The next question was checking if it was possible to convert this vulnerability into a remotely-executable proof of concept, using `android-app` Intent URIs.

Intent URIs are commonly used by deep linking features; you can read more about them here and here.

Ill setup a Python server to allocate this next exploit. The first thing we need is a file with the same name that we need to steal. In that case, our remote webpage should be able to trigger a download of the `profiles.ini` file we created.

Once the file has been downloaded we need to open it; well use a deep linking content URI to continue making use of Firefox moving the downloaded file to its internal directory:

This should do as before; it will request Firefox (`org.mozilla.firefox` part of the `android-app` URI) to open the content URI `content://org.mozilla.firefox.fileprovider/root/sdcard/Download/profiles.ini` with action `android.intent.action.VIEW`, downloading its contents to `/data/data/org.mozilla.firefox/cache/contentUri/profiles.ini` and opening it with the `file://` URI.

Remotely accessing contents of profiles.ini

It works the sensitive files contents have been printed. You can check out the full code here.

Conclusion and final thoughts

As demonstrated, it was possible to steal files from the device solely by having the victim visit a webpage. In a real attack scenario, the malicious file would send the

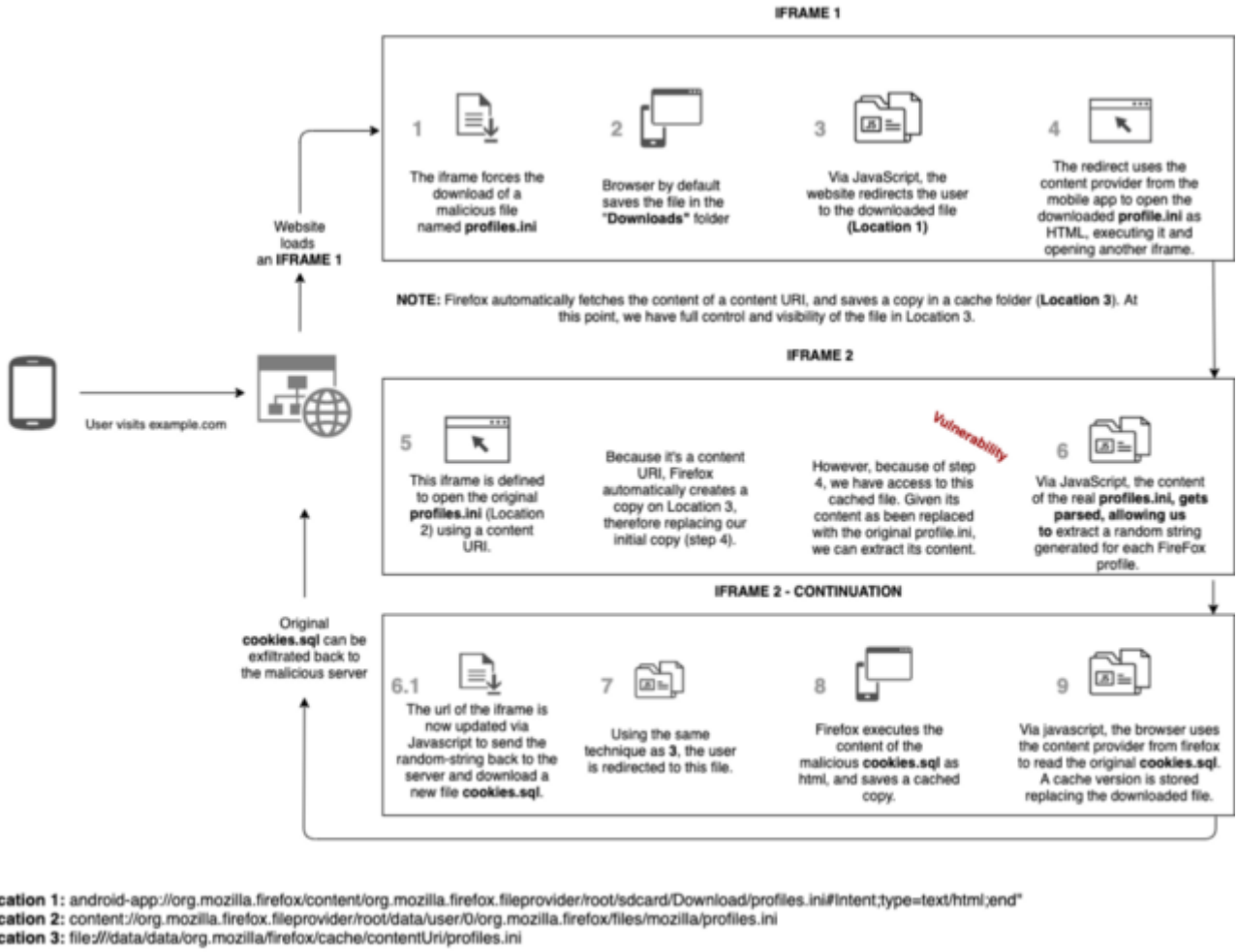
contents read to an attacker-controlled server, rather than outputting the contents in an alert modal.

When I submitted the vulnerability, the proof of concept I presented highlighted the ability to steal Firefoxs cookie database, which was an sqlite database containing all cookies from visited domains.

Below is an example of the PoC:

> Remotely accessing contents of cookies.sql

This diagram shows how the exploit works:



> Diagram of full PoC, sending users cookies to a malicious server

At the time of reporting, Firefox Fennec (v68.9.0) was nearing end-of-life and would have been replaced with Fenix which wasnt vulnerable. Even so, Firefox considered this a critical issue and patched the vulnerability in a short timespan, pushing the version to (v68.10.1). This just shows how seriously these guys take security issues in their platforms. They were also very professional and easy to communicate with.

This writeup is a part of series of writeups I found in 2020 on Android browsers. Stay

tuned for future writeups on other browsers (such as Brave and Samsung Browser)!

Timeline

20200620 Issue reported to Mozilla

20200622 Internal investigation started

20200625 Issue confirmed and addressed (on the same day)

20200706 Fixed version released on Play Store (v68.10.1)

20200707 Bounty assigned ($5000)

Twitter: @kanytu

LinkedIn: www.linkedin.com/in/kanytu

HackerOne: https://hackerone.com/kanytu

Special thanks to @heydean for reviewing this writeup and to @fabiopirespt for the help with the full PoC diagram.