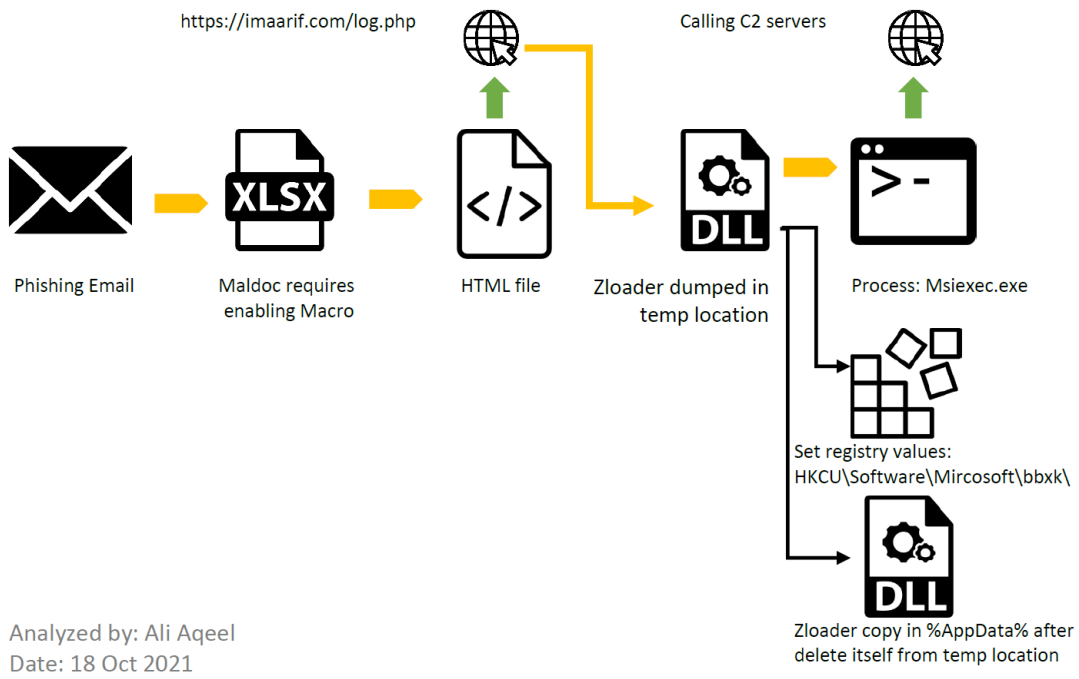
Zloader Reversing

aaqeel01.wordpress.com/2021/10/18/zloader-reversing/

Ali Aqeel October 18, 2021



Aka: ZeusLoader, Deloader, Terdot, Zbot is a malware family that downloads Zeus OpenSSL. Parts of the source code of Zeus were leaked back in 2010 [1] and since couple of versions been forked. Each of the version has its malicious capabilities, but all in common do info stealing specially banking information. Zeus in its core does wild stuff from stealing HTTPS session before being encrypted; to split stolen data and send it in multiple channels over different C2 server based on the stolen info-type [2]. The sent data is being encrypted using RC4 algorithm. Given that major parts of the Zeus being well known and very detectable by almost every AV; Zloader is not just a loader/packer to Zeus core functionality. There are some complicated obfuscation techniques and visual encryption implemented on every single unpacked version of Zloader that bypass security and difficulty extracting configuration. Uncommon attack vector like using Google AdSense has been observed lately [3] also attacker signs Zloader with a certificate compromised from legitimate software in order to evade detection. In this post, we gonna take a look of common Zloader *123 botnet* attack that uses maldoc vector. Quickly analyze maldoc, downloader, and the well known unpacking technique with observed behavior which simple and not quite interesting. However, the second part is going to be deep dive into analyzing and reversing techniques of Zloader unpacked version.

Maldoc

SHA256 500856ee3fc13326cad564894a0423e0583154ef10531de4ab6e6d5df90d4e31

File Type Office Open XML Spreadsheet

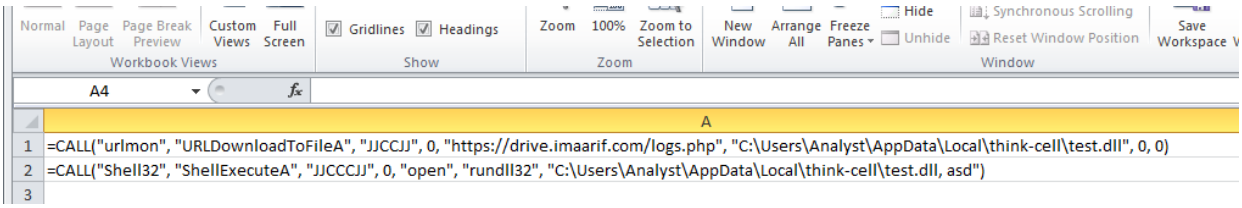
Name tn4598151.xlsm

Size 182.62 KB (187002 bytes)

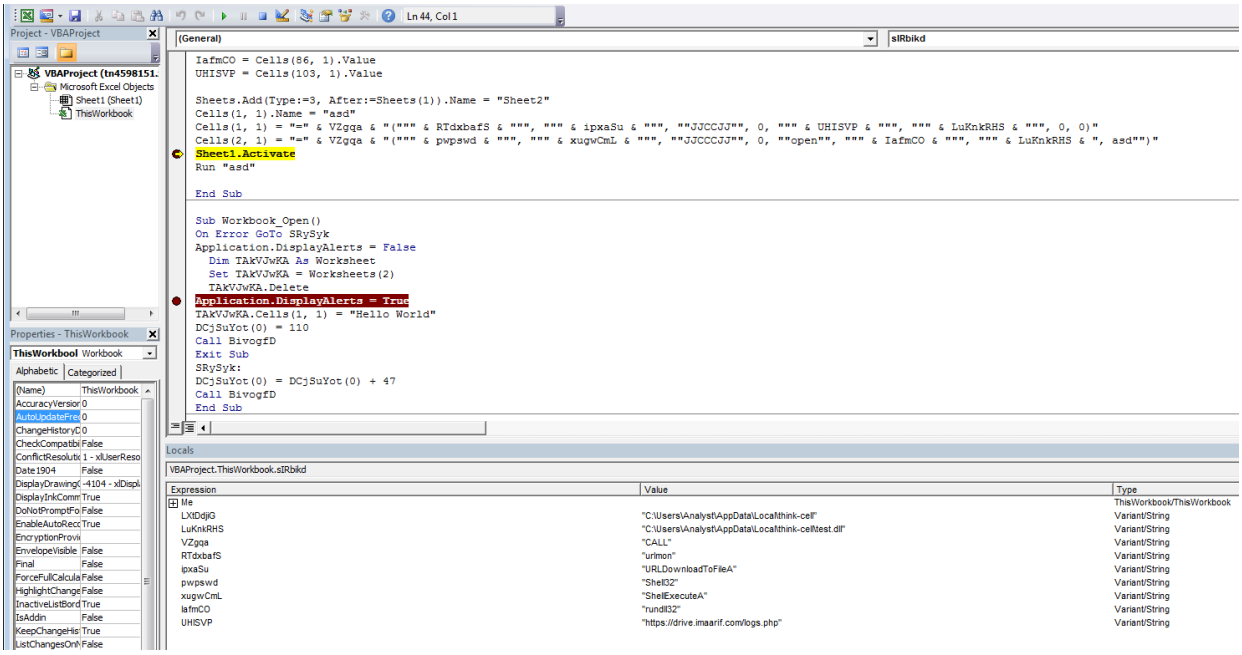
Creation Time 2021-10-04 13:17:51

Links MalwareBazaar, VirusTotal, Any.run

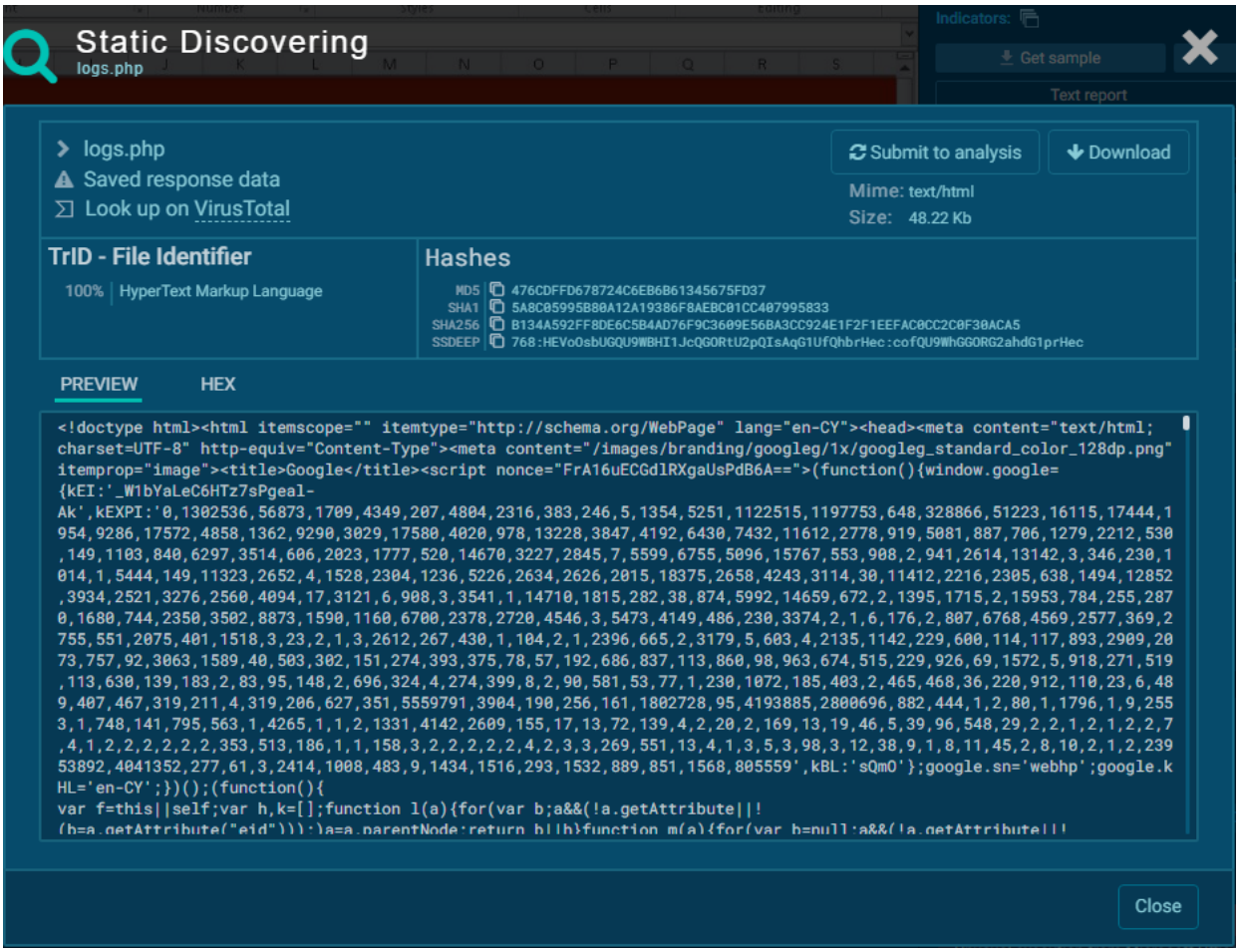
In clear text on sheet2, the maldoc give away downloader URL, directory where it's been dropped, and shell command to run a Dll which is the Zloader.



Enable macro is required to run the above in VBA script.



Downloaded *test.dll* is just an HTML! that downloads *logs.php* which is a Zloader Dll file!



DII Zloader

SHA256 c4ab81d7b7d44dd6dfc4f2b69dbe3f22fbf23c1ae49ab8edac2d26f85ae4514d

File Type Win32 DLL

Names suqyatda.dll, ewviv.dll, ehev.dll, cyvi.dll, logs.php

Size 1.13 MB (1189888 bytes)

Compiler Time-stamp Mon Sep 23 01:29:14 2019

First Submission 2021-10-04 18:23:00

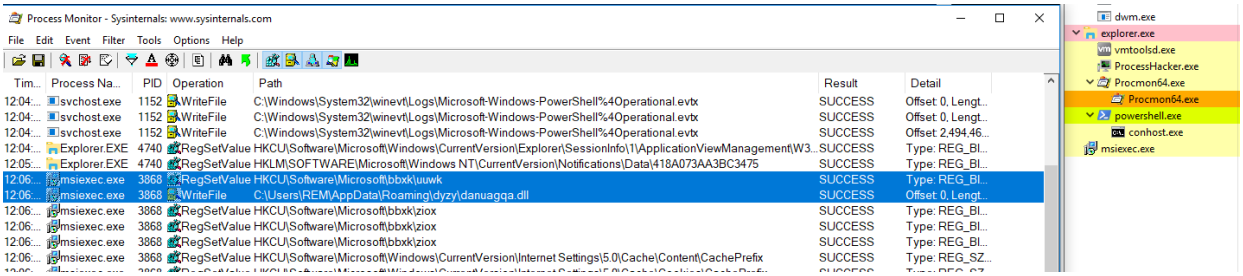
Links MalwareBazaar, VirusTotal, Tria.ge

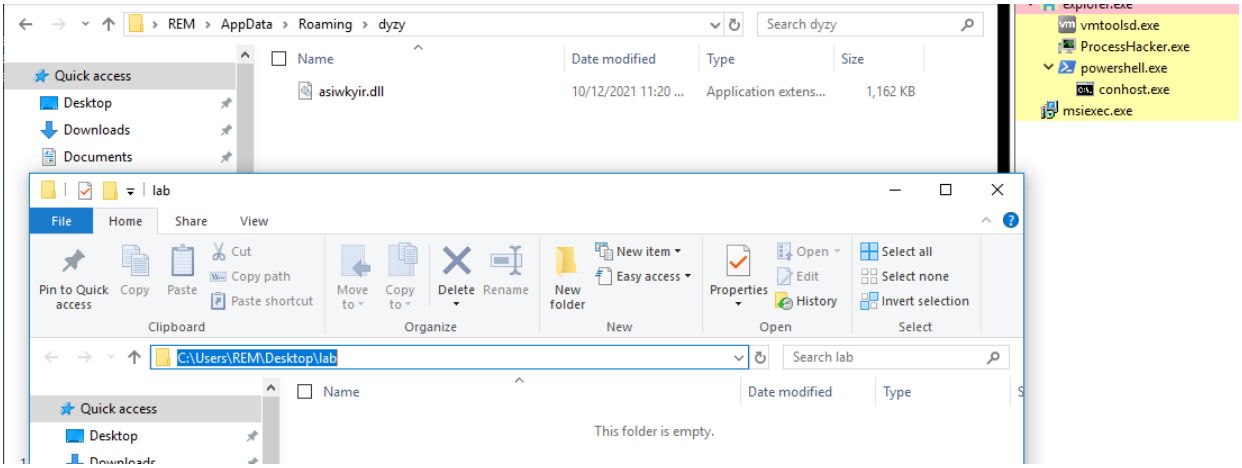
Zloader Dll file is been downloaded and runned in temp location. After Zloader runs:

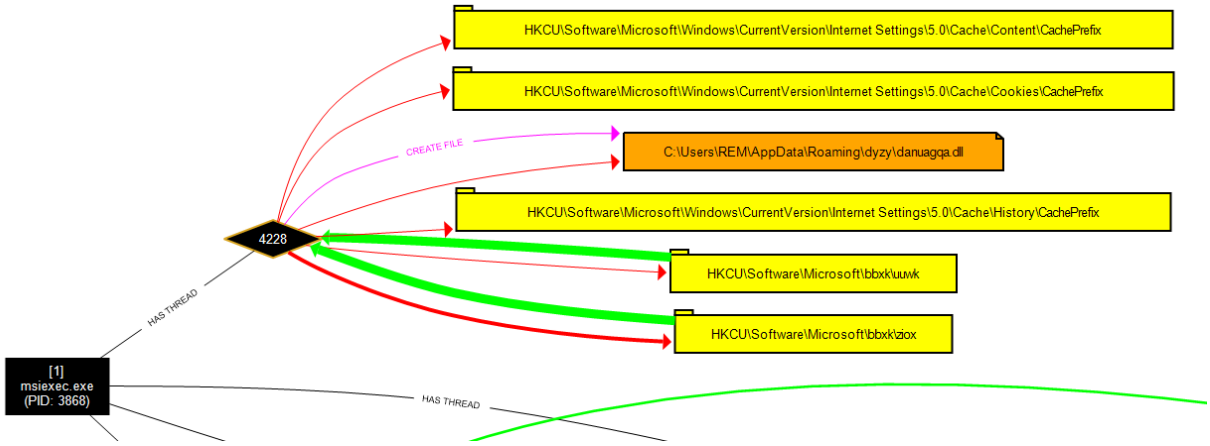
1. Create new process *msiexec.exe* and inject its loader in it.
2. Loader sets new registry values using random hive and key names in:
    - HKCU\Software\Mircosoft*bbxk*\uuwk
    - HKCU\Software\Mircosoft*bbxk*\ziox

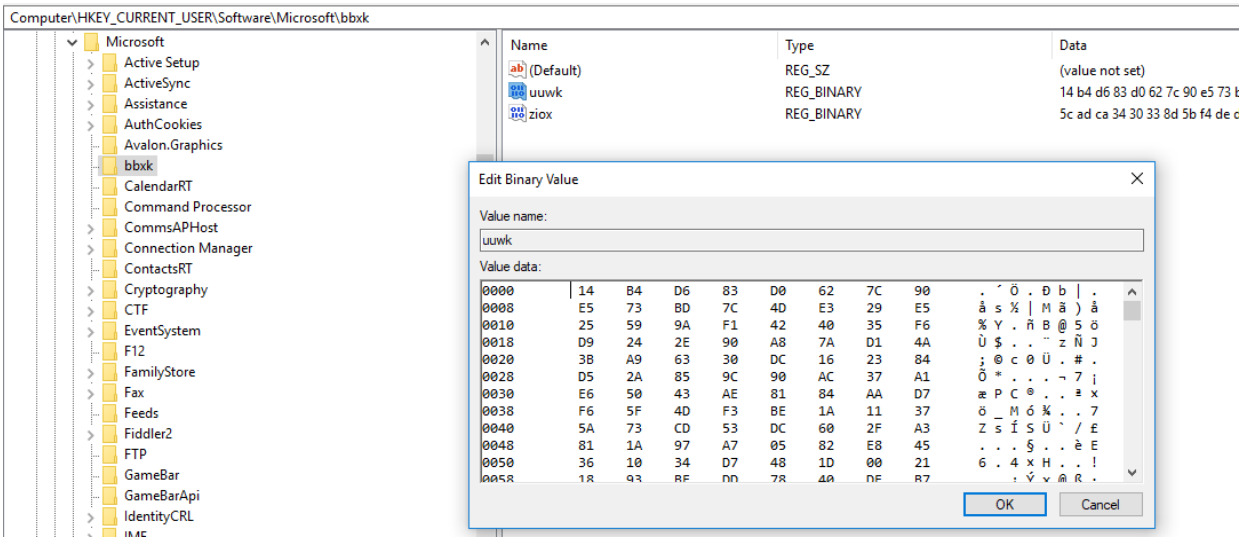
3. Deletes original downloader and copy itself to %AppData%\Roaming*random name*.dll

The registry value calls the new directory for persistent in case of host rebooted. Both registry values are encrypted with RC4 but more to that in next section



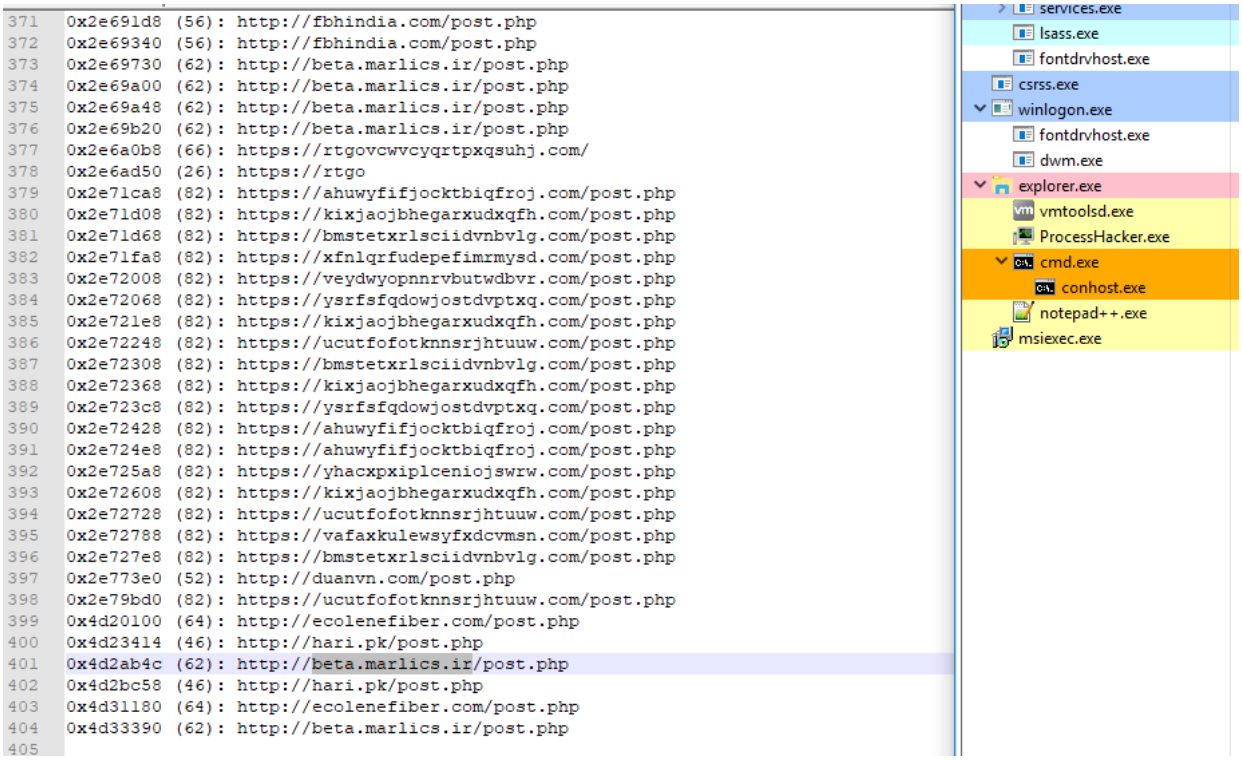


- 

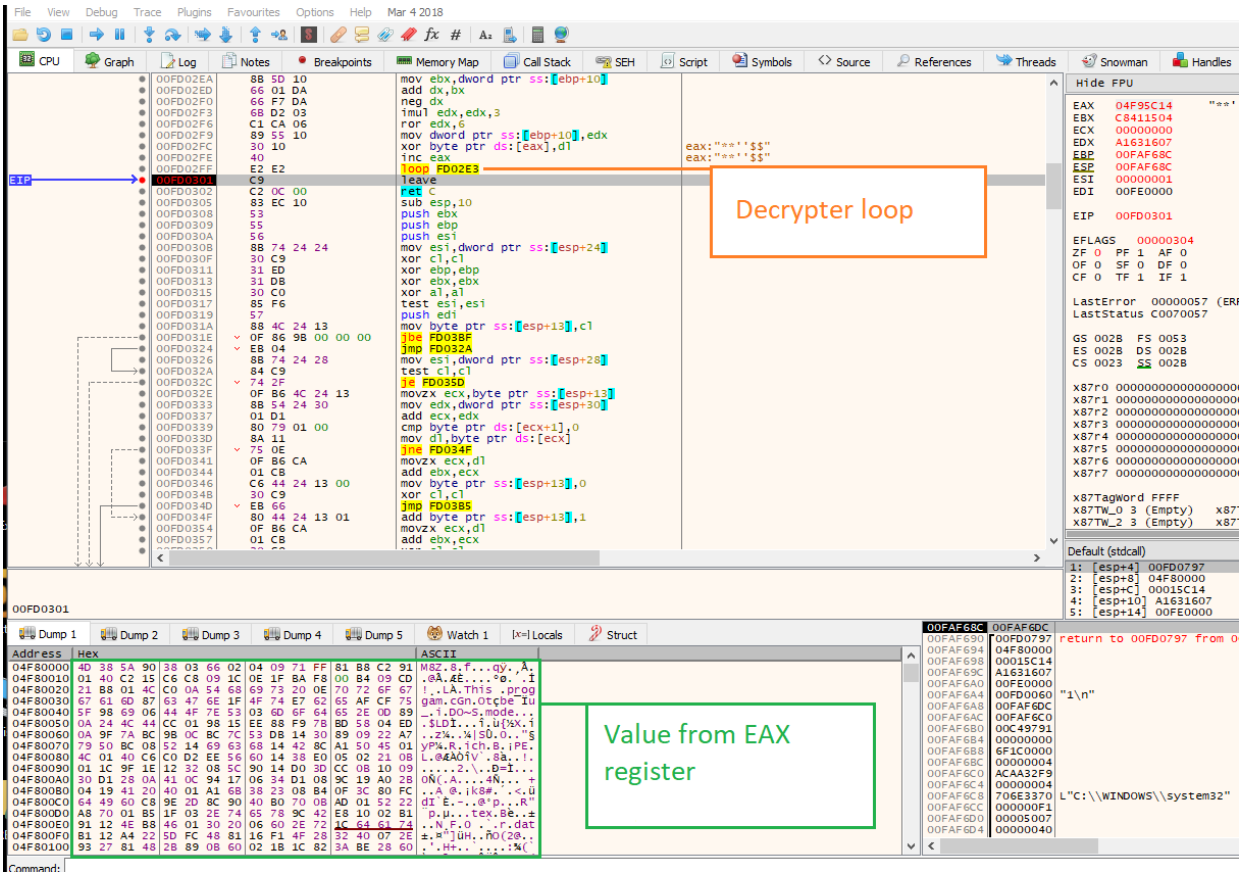


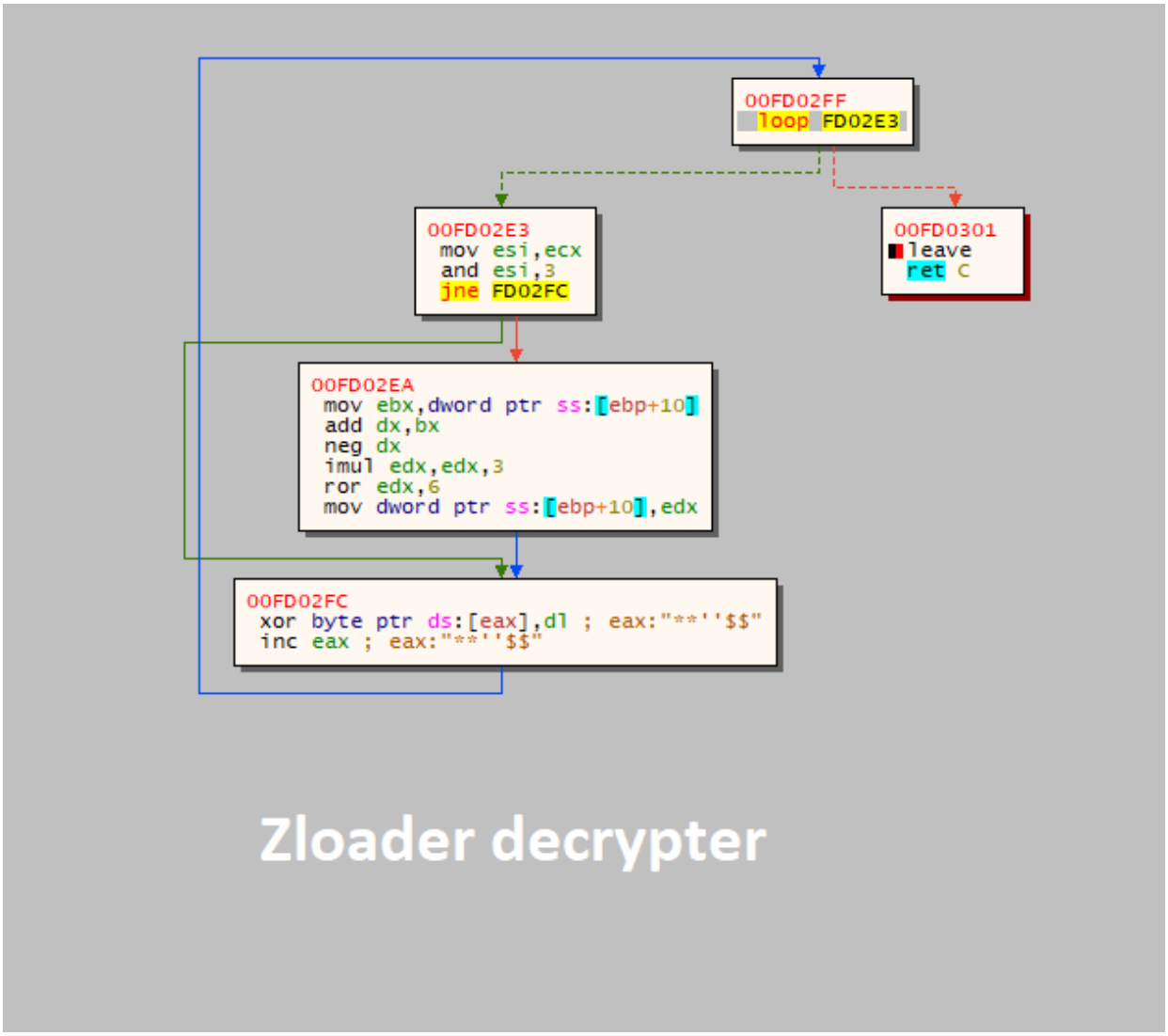
Running Process and Registry Value change

When checking memory strings for any forensics it spells out great number of C2 values. Noticed that 20 URLs has random name with fixed length. Those are called Domain Generated Algorithm DGA, unlike hardcode C2 URLs those are queried during running. More to that later in next section.



Using *pe-sieve64* tool is good way to dump the unpacked Zloader from the running process which is valid PE file to be analyzed. However, just a quick debugging would give same result. In SquirrelWaffle and QakBot recent analysis [4] [5] it's been observed that Zloader among other malwares are using same crypters/decryptor for unpacking mechanism for their loaders before injecting them in process. Following the same debugging method in [4] would reveal the packed Zloader.


Decrypter loop
Value from EAX register



Unpacked Zloader

SHA256
3A4CA58B0A2E72A264466A240C6636F62B8742FFBC96CE14E2225F0E57012E96

File Type Win32 DLL

Name unpacked_zloader_21_10_4.dll,

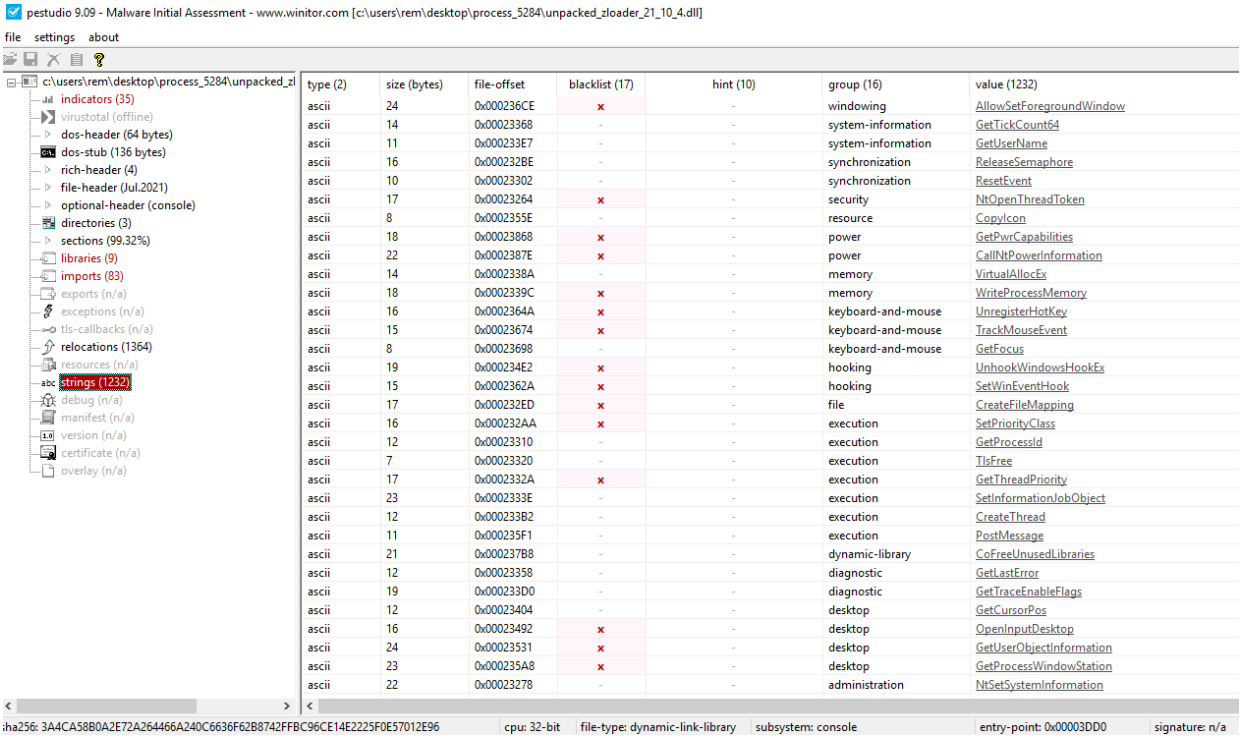
Size 146.00 KB (149504 bytes)

Compiler Time-stamp Wed Jul 14 08:04:16 2021

First Submission 2021-10-18 15:32:37

Links MalwareBazaar, VirusTotal, Tria.ge

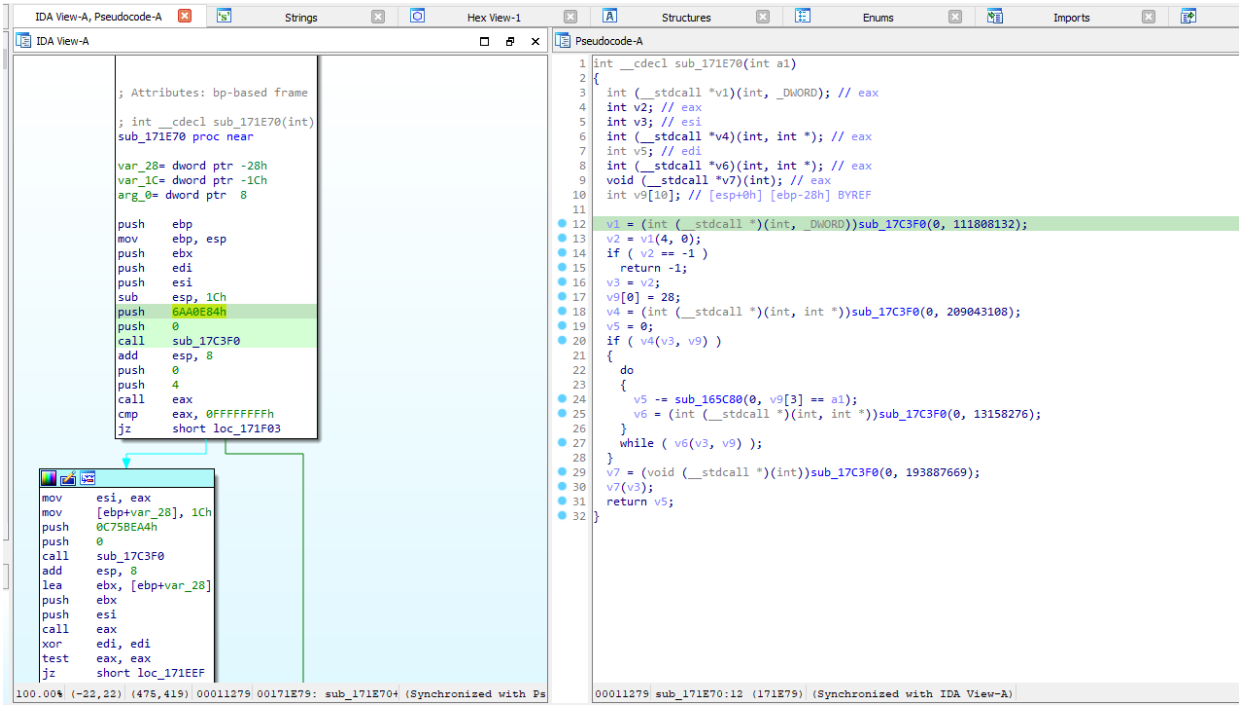
The unpacked Zloader is a master piece of obfuscated functions that waste lots of analysis time to dig into. API strings among other static indicators would not be a good clue for analyzing Zloader. Beside, this malware family is known for API hashing, Visual Encryption using XOR, and RC4 encryption to encrypt strings.

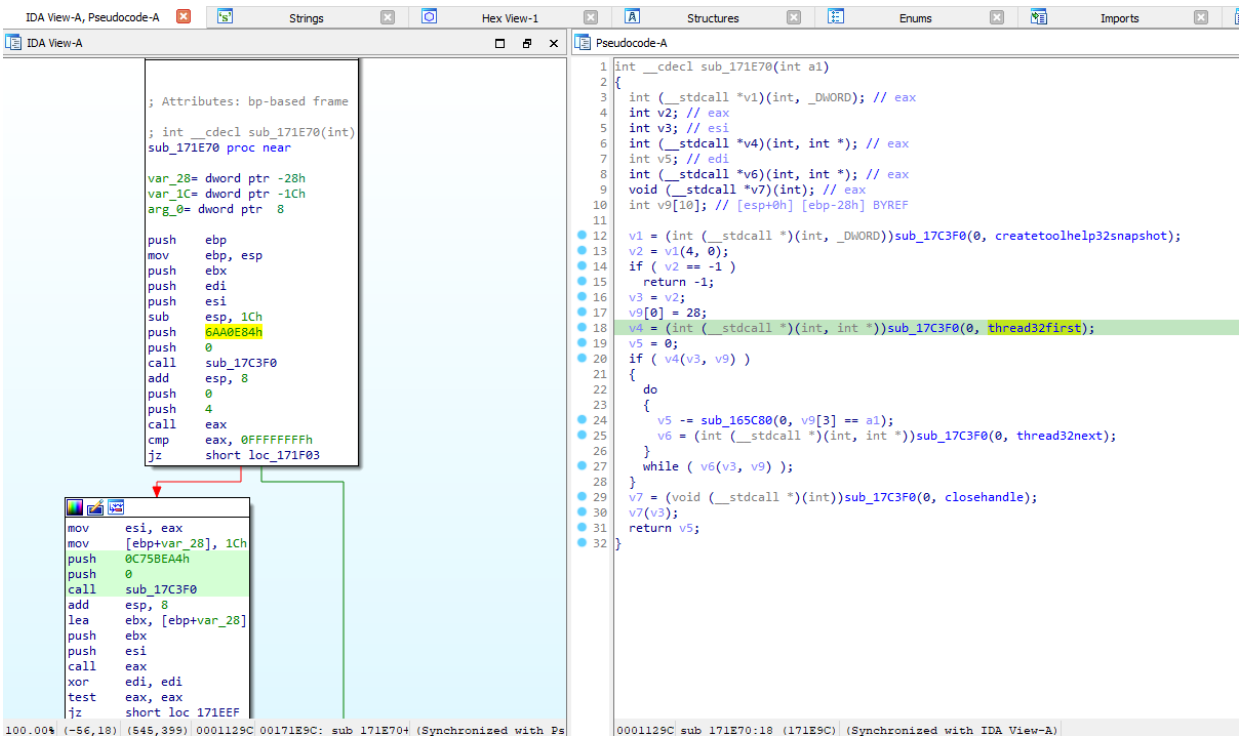


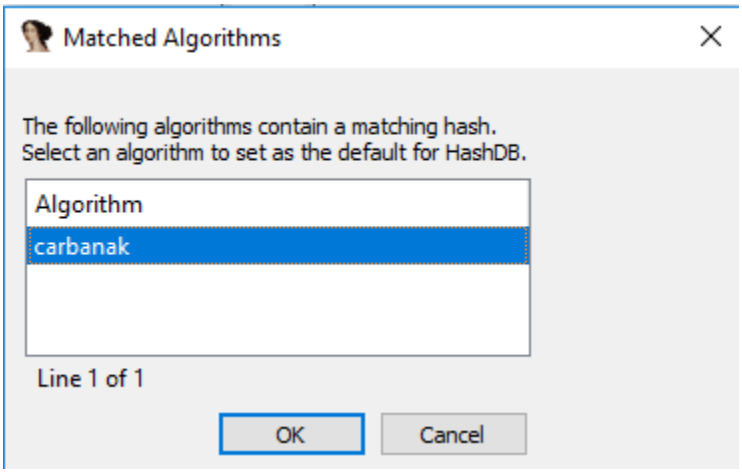
There are five main topics we are going to discuss in this section when reversing Zloader: API hashing, XORing string, extracting Configuration, DGA routine, and Zeus function.

API Hashing:

Statically analyzing Zloader is a bit of a challenge. However, with a new amazing IDA plugin called *HashDB* from OpenAnalysis Labs [6] it's amazing how much obfuscated strings get out the way when reversing Zloader. Just to show a case of what HashDB can do before and after shots of hashed values in a random function. The hashes been checked among large database of hashes with good prediction of hashing algorithms been used.


IDA View-A, Pseudocode-A
Strings
Hex View-1
Structures
Enums
Imports
IDA View-A
; Attributes: bp-based frame
; int __cdecl sub_171E70(int)
sub_171E70 proc near
var_28= dword ptr -28h
var_1C= dword ptr -1Ch
arg_0= dword ptr 8
push ebp
mov ebp, esp
push ebx
push edi
push esi
sub esp, 1Ch
push 6AA0E84h
push 0
call sub_17C3F0
add esp, 8
push 0
push 4
call eax
cmp eax, 0FFFFFFFFh
jz short loc_171F03
mov esi, eax
mov [ebp+var_28], 1Ch
push 0C75BEA4h
push 0
call sub_17C3F0
add esp, 8
lea ebx, [ebp+var_28]
push ebx
push esi
call eax
xor edi, edi
test eax, eax
jz short loc_171EEF
100.00% (-22,22) (475,419) 00011279 00171E79: sub_171E70+ (Synchronized with Ps
Pseudocode-A
int __cdecl sub_171E70(int a1)
{
int (__stdcall *v1)(int, _DWORD); // eax
int v2; // eax
int v3; // esi
int (__stdcall *v4)(int, int *); // eax
int v5; // edi
int (__stdcall *v6)(int, int *); // eax
void (__stdcall *v7)(int); // eax
int v9[10]; // [esp+0h] [ebp-28h] BYREF
v1 = (int (__stdcall *)(int, _DWORD))sub_17C3F0(0, 111808132);
v2 = v1(4, 0);
if (v2 == -1)
return -1;
v3 = v2;
v9[0] = 28;
v4 = (int (__stdcall *)(int, int *))sub_17C3F0(0, 209043108);
v5 = 0;
if (v4(v3, v9))
{
do
{
v5 -= sub_165C80(0, v9[3] == a1);
v6 = (int (__stdcall *)(int, int *))sub_17C3F0(0, 13158276);
}
while (v6(v3, v9));
}
v7 = (void (__stdcall *)(int))sub_17C3F0(0, 193887669);
v7(v3);
return v5;
}
00011279 sub_171E70:12 (171E79) (Synchronized with IDA View-A)


IDA View-A, Pseudocode-A
Strings
Hex View-1
Structures
Enums
Imports
IDA View-A
; Attributes: bp-based frame
; int __cdecl sub_171E70(int)
sub_171E70 proc near
var_28= dword ptr -28h
var_1C= dword ptr -1Ch
arg_0= dword ptr 8
push ebp
mov ebp, esp
push ebx
push edi
push esi
sub esp, 1Ch
push 6AA0E84h
push 0
call sub_17C3F0
add esp, 8
push 0
push 4
call eax
cmp eax, 0FFFFFFFFh
jz short loc_171F03
mov esi, eax
mov [ebp+var_28], 1Ch
push 0C75BEA4h
push 0
call sub_17C3F0
add esp, 8
lea ebx, [ebp+var_28]
push ebx
push esi
call eax
xor edi, edi
test eax, eax
jz short loc_171EEF
100.00% (-56,18) (545,399) 0001129C 00171E9C: sub_171E70+ (Synchronized with Ps
Pseudocode-A
int __cdecl sub_171E70(int a1)
{
int (__stdcall *v1)(int, _DWORD); // eax
int v2; // eax
int v3; // esi
int (__stdcall *v4)(int, int *); // eax
int v5; // edi
int (__stdcall *v6)(int, int *); // eax
void (__stdcall *v7)(int); // eax
int v9[10]; // [esp+0h] [ebp-28h] BYREF
v1 = (int (__stdcall *)(int, _DWORD))sub_17C3F0(0, createtoolhelp32snapshot);
v2 = v1(4, 0);
if (v2 == -1)
return -1;
v3 = v2;
v9[0] = 28;
v4 = (int (__stdcall *)(int, int *))sub_17C3F0(0, thread32first);
v5 = 0;
if (v4(v3, v9))
{
do
{
v5 -= sub_165C80(0, v9[3] == a1);
v6 = (int (__stdcall *)(int, int *))sub_17C3F0(0, thread32next);
}
while (v6(v3, v9));
}
v7 = (void (__stdcall *)(int))sub_17C3F0(0, closehandle);
v7(v3);
return v5;
}
0001129C sub_171E70:18 (171E9C) (Synchronized with IDA View-A)

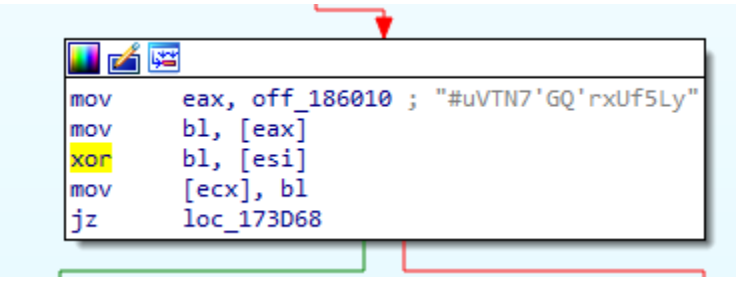


XORing Strings:

With API hashing out of the way. It's important to get reversing tricks to dig into the main functions and extract configurations. There're very limited hardcode strings in Zloader that can be clues like those.

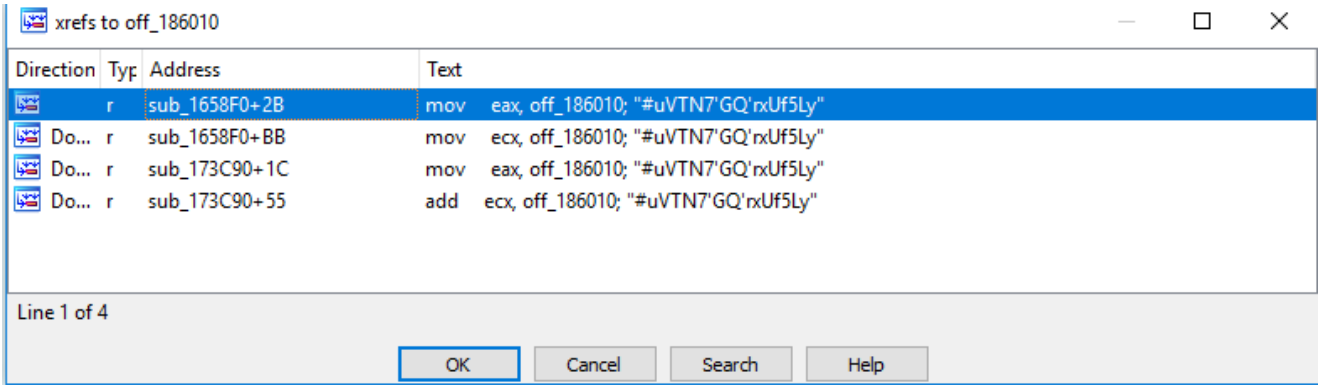
```
.text:00161EDE    call    sub_170ED0
.text:00161EE3    push    offset aQhpacozsstaznu ; "qhpacozsstaznupphhedjtuoww"
.text:00161EE8    push    offset unk_184404
.text:00161EED    call    sub_1656B0
.text:00165911    cmp     [ebp+arg_4], 0
.text:00165915    jz      loc_165A46
.text:0016591B    mov     eax, off_186010 ; "#uVTN7'GQ'rxUf5Ly"
.text:00165920    movzx   ebx, word ptr [esi]
.text:00165923    movsx   esi, byte ptr [eax]
.text:00165926    mov     dword ptr [ebp+var_10], ebx
```

First, let's look at **Off_186010** which is an offset of an offset of a memory location **rdata:00183D80** with literal string (#uVTN7'GQ'rxUf5Ly). When cross referencing this offset it's been used 4 times in two different functions. And there's some sort of XOR function in both subroutines which reveal this is the key literal string could be an XOR key value.



```
loc_165990:
push    0A95168F5h
push    edi
call    sub_161DE0
add     esp, 8
lea     edi, [eax+56AE970Ch]
mov     esi, eax
mov     edx, 0F0F0F0F1h
mov     ecx, off_186010 ; "#uVTN7'GQ'rxUf5Ly"
mov     eax, edi
mul     edx
shr     edx, 4
mov     eax, edx
shl     eax, 4
add     eax, edx
neg     eax
lea     eax, [esi+eax+56AE970Ch]
movsx   eax, byte ptr [ecx+eax]
mov     ecx, [ebp+arg_0]
movzx   ecx, word ptr [ecx+esi*2-52A2D1E8h]
mov     ebx, ecx
xor     ebx, eax
push    eax
push    ecx
call    sub_161D70
add     esp, 8
mov     ecx, [ebp+arg_4]
test    bx, bx
mov     [ecx+esi*2-52A2D1E8h], bx
jz      short loc_165A48
```

```
34    if ( (_WORD)v7 )
35    {
36      v8 = v7;
37      v9 = 0;
38      while ( 1 )
39      {
40        v13 = sub_167E80(12429);
41        v14 = -sub_163120(-(__int16)v8, -v13);
42        sub_167E80(12429);
43        if ( (unsigned __int16)v14 >= 0x5Fu )
44        {
45          if ( (unsigned __int16)v8 > 0xDu )
46            break;
47          v15 = 9728;
48          if ( !_bittest(&v15, v8) )
49            break;
50        }
51        v10 = sub_161DE0(v9, -1454282507);
52        v9 = v10 + 1454282508;
53        v11 = v10;
54        v12 = off_186010[v10 - 17 * ((v10 + 1454282508) / 0x11u) + 1454282508];
55        v8 = v12 ^ (unsigned __int16)a1[v11 - 693201140];
56        sub_161D70(a1[v11 - 693201140], v12);
57        v3 = a2;
58        a2[v11 - 693201140] = v8;
59        if ( !(_WORD)v8 )
60          return v3;
61      }
62      return a1;
63    }
64  }
65  else
66  {
```

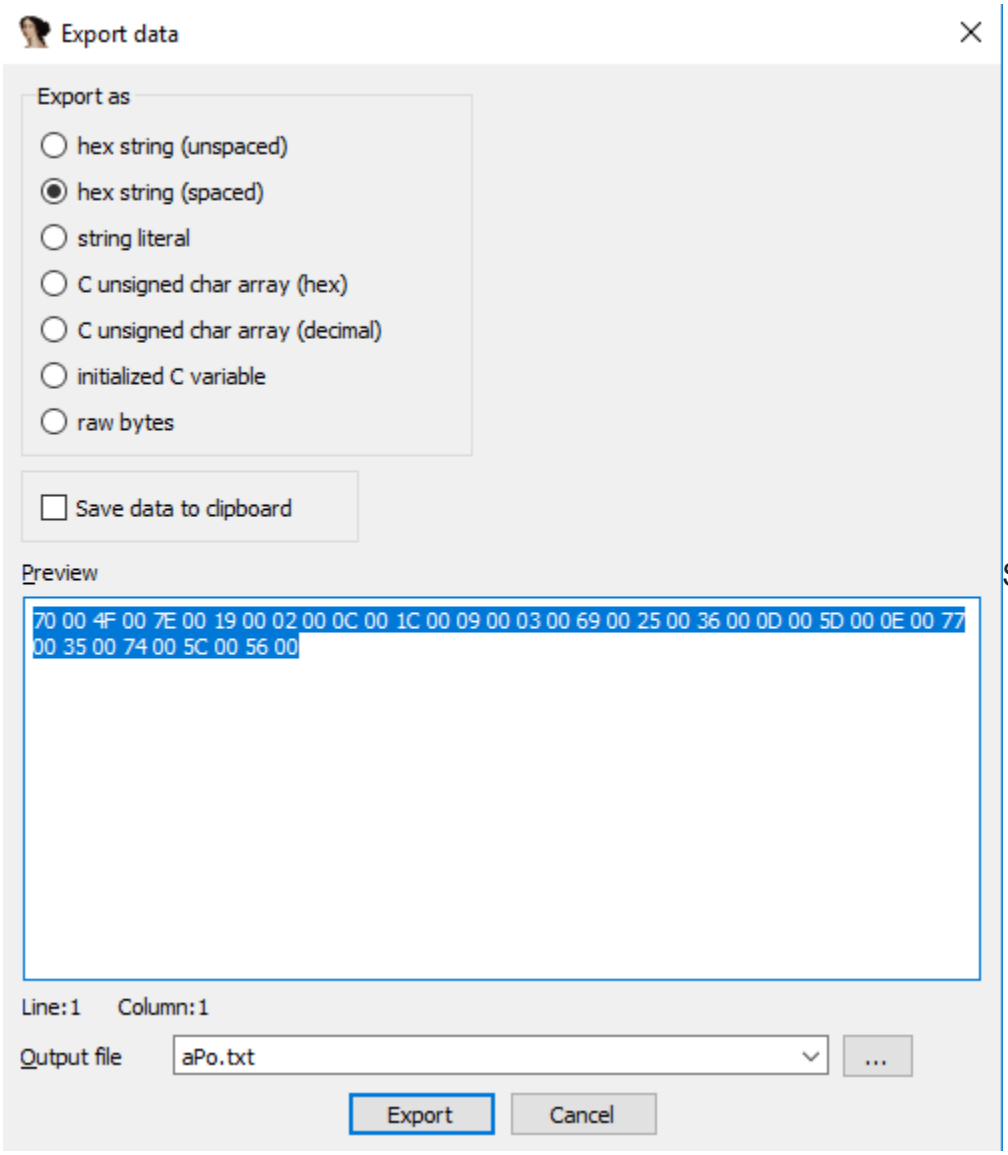


Cross referencing both **sub_1658F0** and **sub_173C90** routines would shows that over 120 times those functions has been called. Randomly checking any of the cross referencing like below

….skipped lines……..
text:001610E0 push offset unk_183E8E
.text:001610E5 call sub_1658F0
….skipped lines……..
text:0016444E push offset unk_184310
text:00164453 call sub_173C90
….. skipped lines……
text:00165685 push offset unk_184040
text:0016568A call sub_1658F0

We noticed both subrouties been called after a push of unknow offsets

Let’s use XOR key (#uVTN7’GQ’rxUf5Ly) with **offset unk_184040** value.



Shift+E over unknow offset location

Hex: 70 00 1A 00 30 00 20 00 39 00 56 00 55 00 22 00 0D 00 6A 00 1B 00 1B 00 27 00 09 00 46 00 23 00 1F 00 57 00 29 00 56 00

key: #uVTN7'GQ'rxUf5Ly

Result: SuLT~7.Gh'$x.f.Lt#.VON,'`Q.r>UE5Sytu.T.7

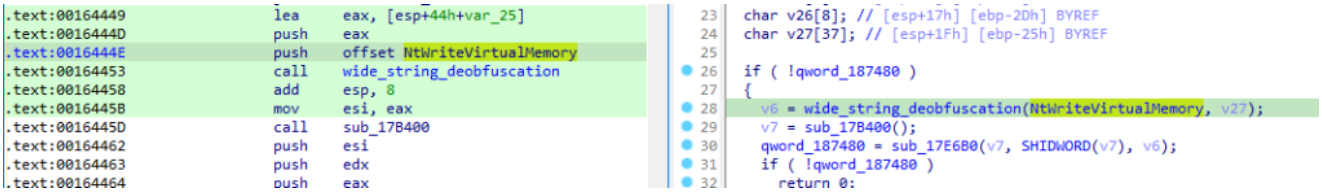
The XORed value/result doesn't make sense. If anything noticeable that the Hex values has zeros in sequence. Which indicate **sub_1658F0** is for wide character and this makes and **sub_173C90** for normal character. let's try again deleting all repeated zeros and XOR with the key

Hex: 701A3020395655220D6A1B1B270946231F572956

key: #uVTN7'GQ'rxUf5Ly

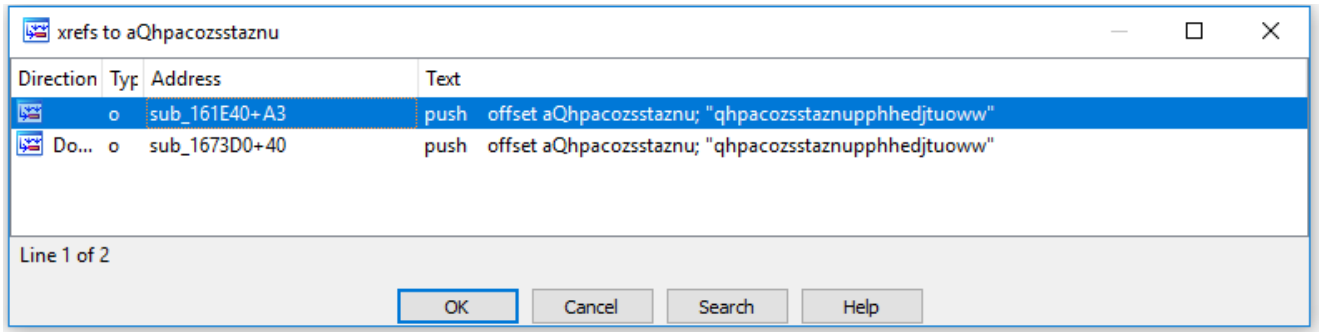
Result: Software\Microsoft\

It’s not just strings that been obfuscated, some API calls been XORed too. Almost 120 offset being pushed in stack which means 120 strings are being XORed and to make it readable; *Appendix – A* contains all the strings with addresses after been XORed.



Configuration:

The other string ‘qhpacozsstaznupphhedjtuoww’ is 26 length. It’s crossed referenced twice in two separate routines.



snipped assembly from **sub_161E40** and **sub_1673D0** routines

.text:00161EE3 push offset aQhpacozsstaznu ; “qhpacozsstaznupphhedjtuoww”
.text:00161EE8 push offset unk_184404
——-skipped lines—
text:001673D0 sub_1673D0 proc near ; CODE XREF: sub_171400+40↓p
.text:001673D0 push ebp
.text:001673D1 mov ebp, esp
.text:001673D3 push edi
.text:001673D4 push esi
.text:001673D5 mov esi, ecx
.text:001673D7 call sub_1809A0
.text:001673DC mov edi, [eax+30h]
.text:001673DF mov ecx, esi
.text:001673E1 call sub_1809A0
.text:001673E6 add eax, edi
.text:001673E8 push 36Fh
.text:001673ED push offset unk_184404
.text:001673F2 push eax
.text:001673F3 call sub_171D80
.text:001673F8 add esp, 0Ch
.text:001673FB mov ecx, esi
.text:001673FD call sub_1809A0

.text:00167402 mov edi, [eax+34h]
.text:00167405 mov ecx, esi
.text:00167407 call sub_1809A0
.text:0016740C add eax, edi
.text:0016740E push 64h ; ‘d’
.text:00167410 push offset aQhpacozsstaznu ; “qhpacozsstaznupphhedjtuoww”

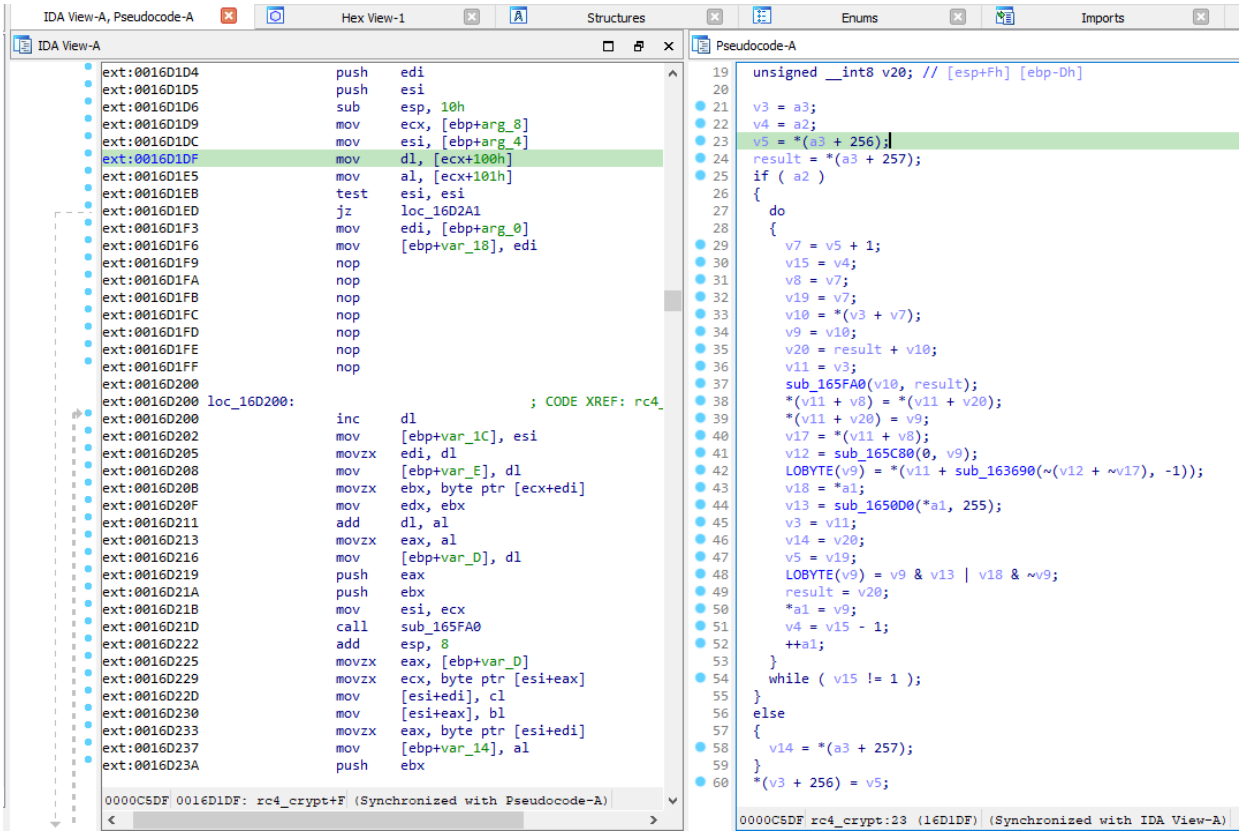
In both routines notice a repeated push to an offset **unk_184404**. This offset contains configurations. Noticed that both offset passed into a function **sub_1656B0** (*name decrypting_rc4*)

Enums | Imports | Exports

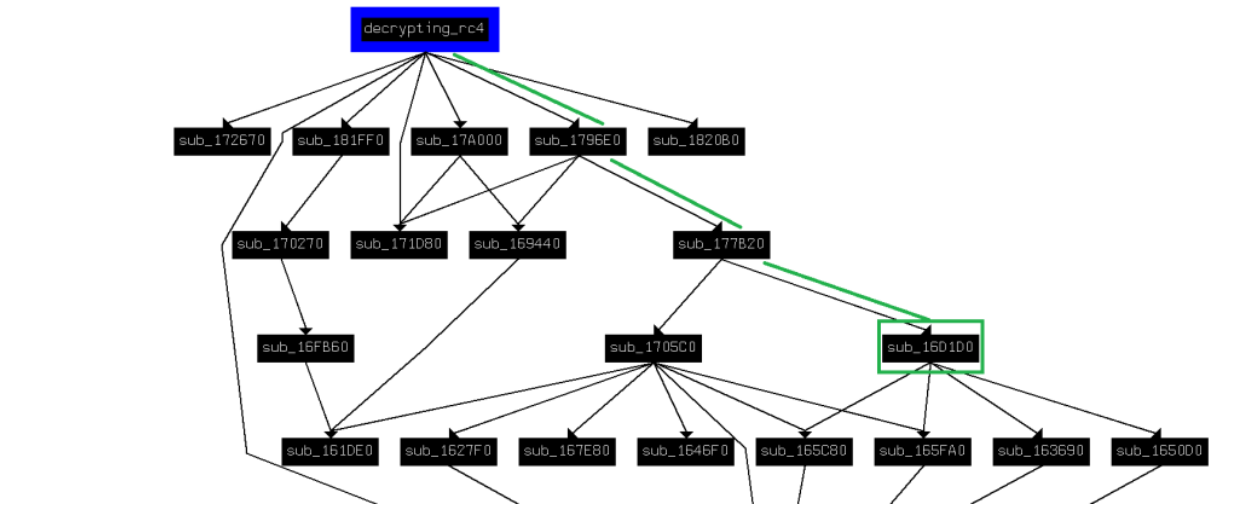
Pseudocode-A

```
1  char __usercall C2@<al>(int a1@<ebx>)
2  {
3    unsigned __int8 *v1; // eax
4    unsigned int v2; // eax
5    void (__cdecl *v3)(int (__stdcall *)(int)); // eax
6    unsigned int v4; // eax
7    void (__cdecl *v5)(int); // eax
8    char v7[13]; // [esp+3h] [ebp-Dh] BYREF
9
10   if ( !sub_16EDD0() )
11     return 0;
12   v1 = wide_string_deobfuscation(kernel32_dll_0, v7);
13   if ( !sub_175E70(lpLibFileName, v1) )
14     return 0;
15   if ( !LoadLibraryA(lpLibFileName) )
16     return 0;
17   call_get_proc_heap();
18   v2 = sub_167890(-1514247953);
19   v3 = Resolve_api(0, v2);
20   v3(exit_thread);
21   v4 = sub_167890(-1432131992);
22   v5 = Resolve_api(0, v4);
23   v5(32775);
24   call_internet_set_option();
25   sub_174FA0();
26   sub_170ED0();
27   decrypting_rc4(&config, "qhpacozsstaznupphhedjtuoww");
28   sub_16F5D0(a1);
29   if ( !call_getModule_name() || !call_get_length_sid() || !sub_168830() )
30     return 0;
31   call_getcurrentprocessid();
32   sub_174E80();
33   return 1;
34 }
```

Pseudo code from **sub_1656B0** routine
Decrypting_rc4 function calls multiple function and those are calling other functions. What we are looking here is RC4 algorithm.

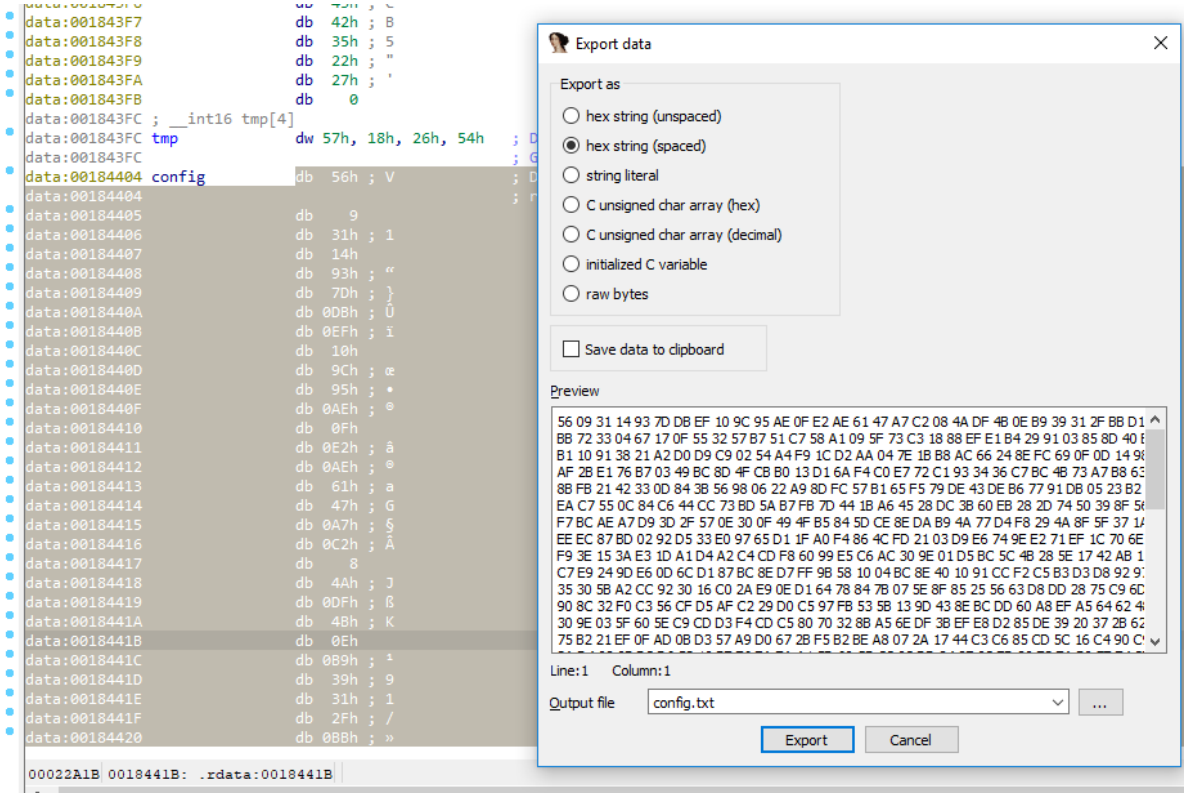


To have mind map where RC4 algorithm location lets Xref-from Decrypting_rc4 function where Config strings and key retrieved



Xref_from **sub_1656B0** (decrypting_rc4)

Now let's go back to the configuration 'config' offset in data block and copy its hex value to CyberChef and use RC4 algorithm to decrypt it with the key (qhpacozsstaznupphhedjtuoww)



Notice three things: got C2 URLs, list in Table-1, and 123 which is ID for this variant of Zloader, and at the tail there's this value (djfsf02hf832hf03) which is another RC4 key that decrypt the registry values in \HKEY_CURRENT_USER\Software\Microsoft\bbxk and also encrypt decrypt traffic with C2 [7].

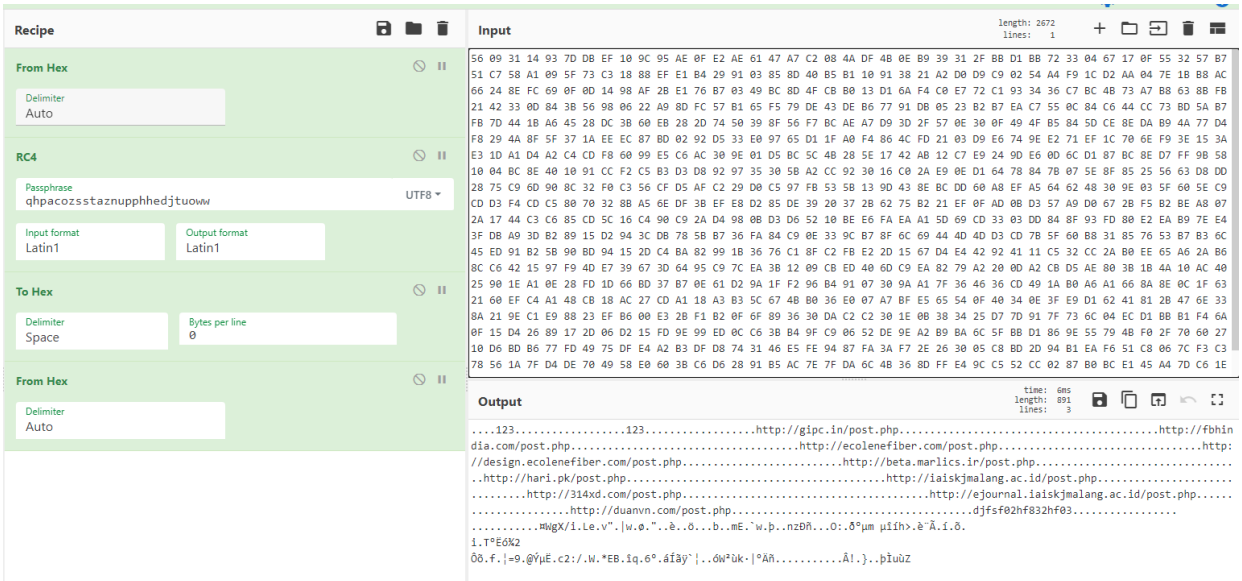
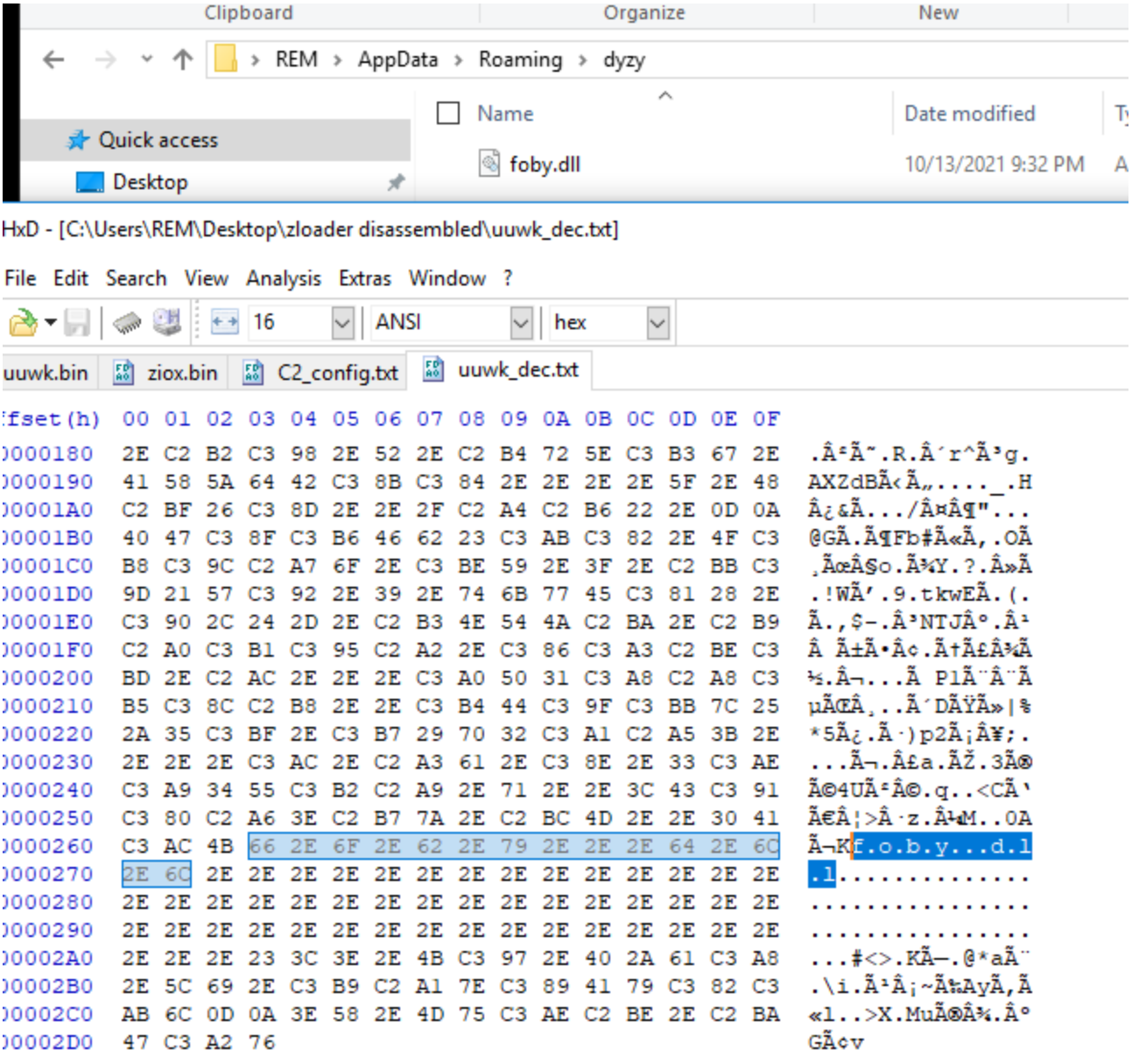


Table-1

123
hxxp://gipc.in/post[.]php
hxxp://fbhindia.com/post[.]php

hxxp://ecolenefiber.com/post[.]php
hxxp://design.ecolenefiber.com/post[.]php
hxxp://beta.marlics.ir/post[.]php
hxxp://hari.pk/post[.]php
hxxp://iaiskjmalang.ac.id/post[.]php
hxxp://314xd.com/post[.]php
hxxp://ejournal.iaiskjmalang.ac[.]id/post.php
hxxp://duanvn.com/post[.]php
djfsf02hf832hf03

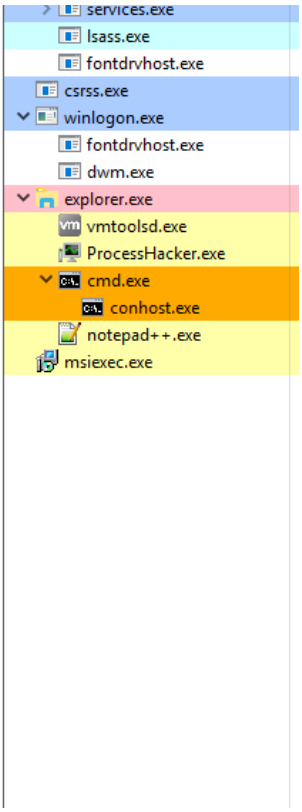
Decrytped registry key value contains host name and the Zloader in %AppData% directory.



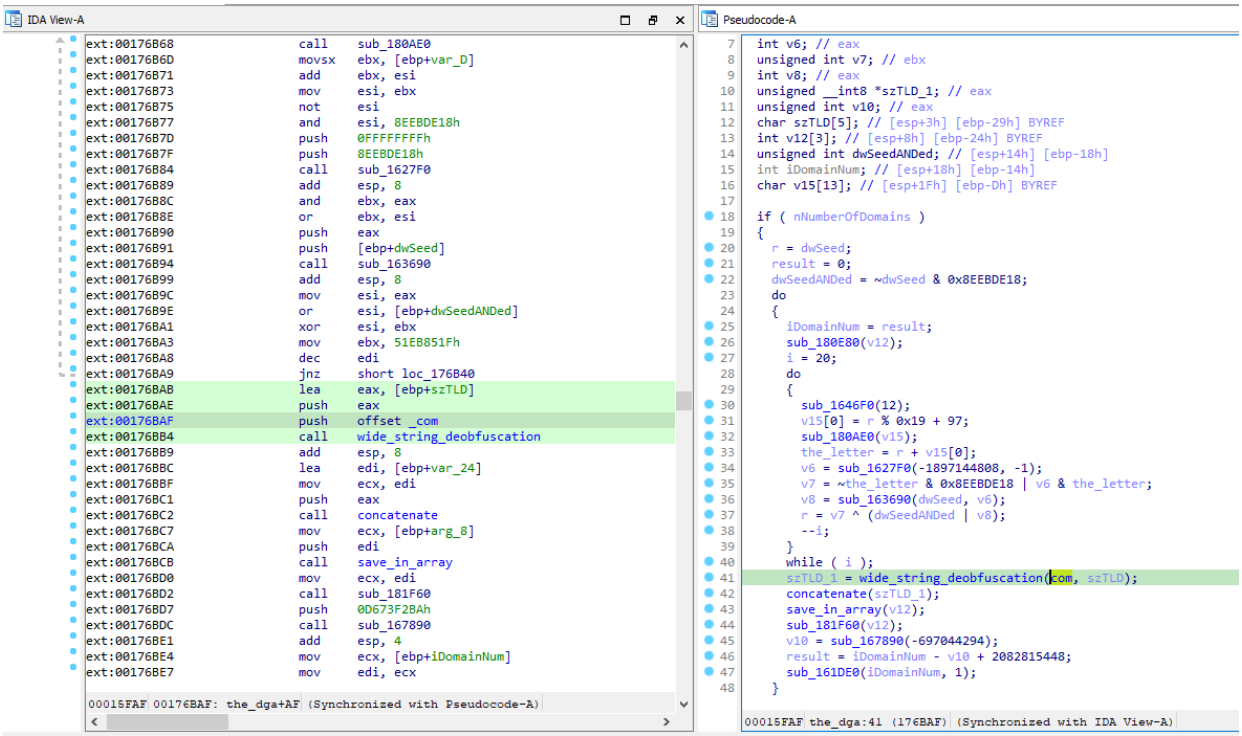
DGA:

Zloader know for using DGA algorithm and we notice above some of the generated 32 character length URLs. To find the DGA function in this we can look for *.com* or *post.php* strings that been deobfuscatd in the previous section of XORing strings.

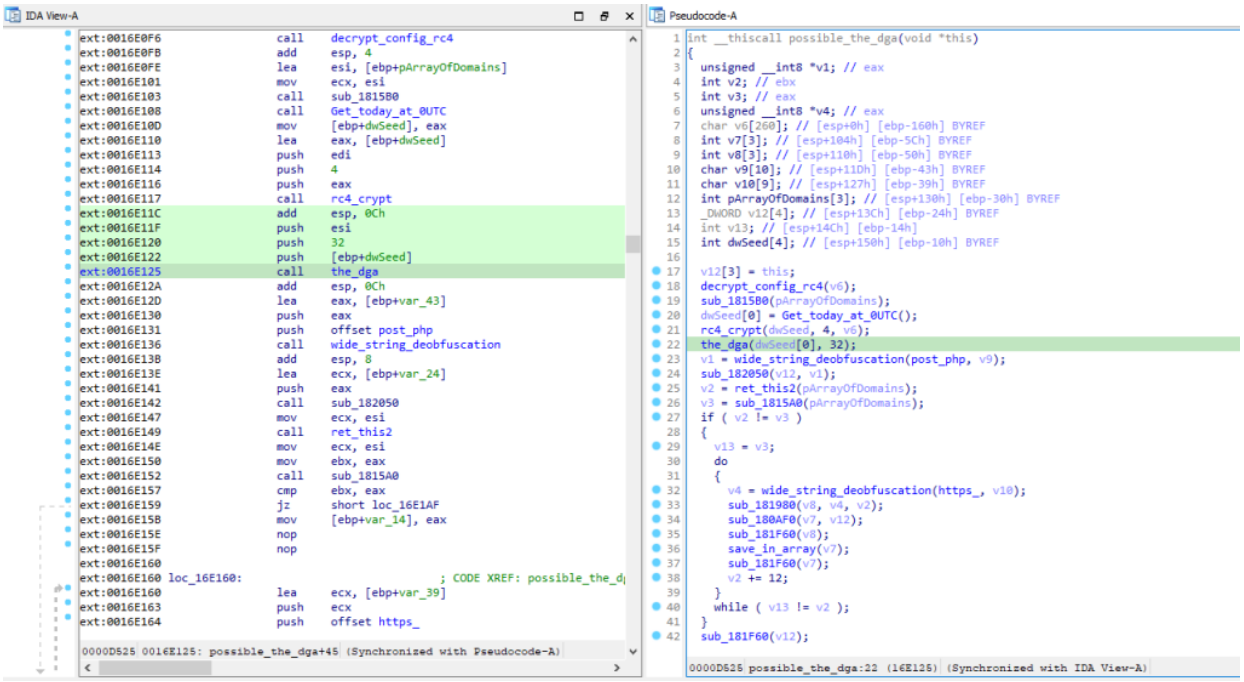
```
0x2e691d8 (56): http://fbhindia.com/post.php
0x2e69340 (56): http://fbhindia.com/post.php
0x2e69730 (62): http://beta.marlics.ir/post.php
0x2e69a00 (62): http://beta.marlics.ir/post.php
0x2e69a48 (62): http://beta.marlics.ir/post.php
0x2e69b20 (62): http://beta.marlics.ir/post.php
0x2e6a0b8 (66): https://rtgovcwvcyqrtpxqsuhj.com/
0x2e6ad50 (26): https://rtgo
0x2e71ca8 (82): https://ahuwyfifjocktbiqfroj.com/post.php
0x2e71d08 (82): https://kixjaojbhegarxudxqfh.com/post.php
0x2e71d68 (82): https://bmstetxrlsciidvnbvlg.com/post.php
0x2e71fa8 (82): https://xfnlqrfudepefimrmysd.com/post.php
0x2e72008 (82): https://veydwyopnnrvbutwdbvr.com/post.php
0x2e72068 (82): https://ysrfsfqdowjostdvptxq.com/post.php
0x2e721e8 (82): https://kixjaojbhegarxudxqfh.com/post.php
0x2e72248 (82): https://ucutfofotknnsrjhtuuw.com/post.php
0x2e72308 (82): https://bmstetxrlsciidvnbvlg.com/post.php
0x2e72368 (82): https://kixjaojbhegarxudxqfh.com/post.php
0x2e723c8 (82): https://ysrfsfqdowjostdvptxq.com/post.php
0x2e72428 (82): https://ahuwyfifjocktbiqfroj.com/post.php
0x2e724e8 (82): https://ahuwyfifjocktbiqfroj.com/post.php
0x2e725a8 (82): https://yhacxpxiplceniojswrw.com/post.php
0x2e72608 (82): https://kixjaojbhegarxudxqfh.com/post.php
0x2e72728 (82): https://ucutfofotknnsrjhtuuw.com/post.php
0x2e72788 (82): https://vafaxkulewsyfxdcvmsn.com/post.php
0x2e727e8 (82): https://bmstetxrlsciidvnbvlg.com/post.php
0x2e773e0 (52): http://duanvn.com/post.php
0x2e79bd0 (82): https://ucutfofotknnsrjhtuuw.com/post.php
0x4d20100 (64): http://ecolenefiber.com/post.php
0x4d23414 (46): http://hari.pk/post.php
0x4d2ab4c (62): http://beta.marlics.ir/post.php
0x4d2bc58 (46): http://hari.pk/post.php
0x4d31180 (64): http://ecolenefiber.com/post.php
0x4d33390 (62): http://beta.marlics.ir/post.php
```



when cross referencing *.com* from **rdata:001849B4** location we find that it's been called by one function and let's name that function the_dga



The_dga function has been called one by another function. Based on [8], the caller of DGA routine does it math calculating values called Seed based on time and RC4 key (djfsf02hf832hf03) (second key). So the values generated are much different each day passed in used **GetLocalTime** and **SystemTimetoFileTime** APIs. Notice that the_dga function has passed value of 32 which is the same length of the URL string with Seed value which in this case makes the entire caller function to calculate Seed value. followed by *post.php* and *https* while loop. The caller function got many obfuscated function that slows down analysis and it get complicated calculating generated domains manually.



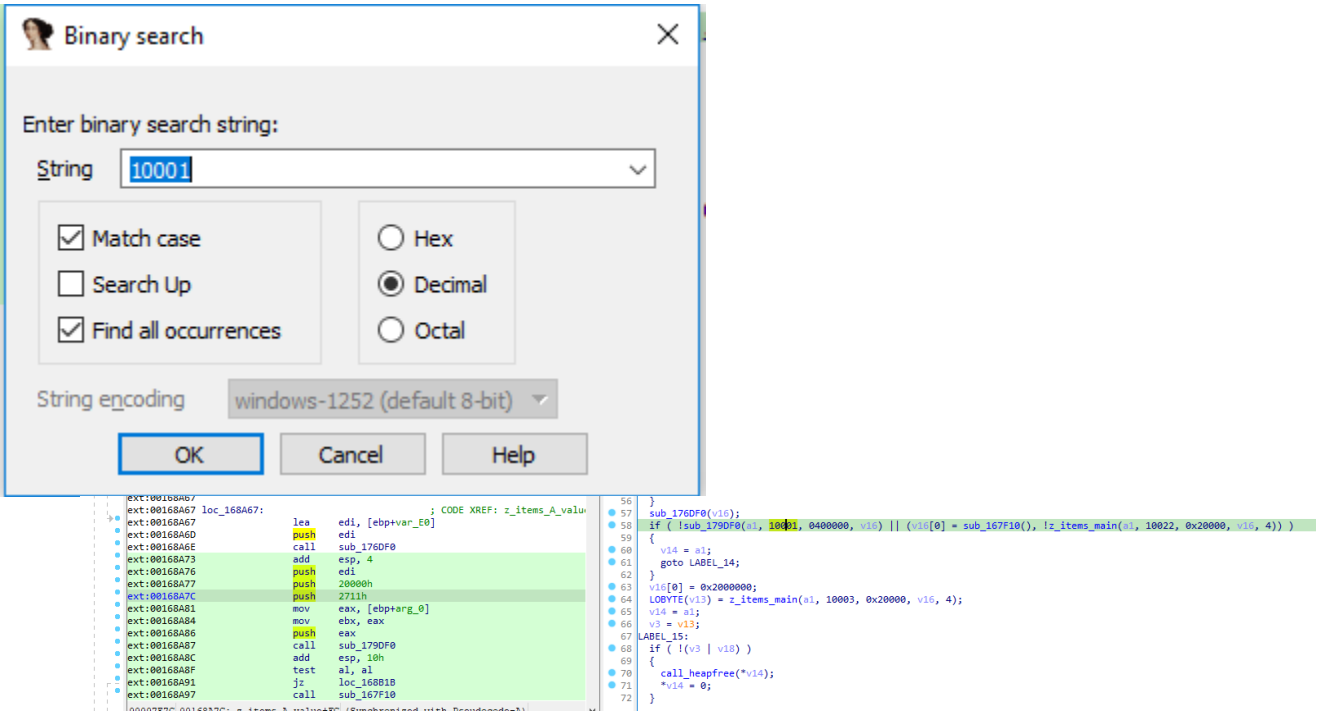
Zeus Items:

Zeus uses item ID as list below which is the main one, there are more extended list based on Zloader version [1] [2] [7]. Each ID passed into a function and dissect information from victim machine. When that information stored in attacker SQL filed it show retrieved info about the host.

Item ID	Value
10001	SBCID BOT ID
10002	SBCID BOTNET
10003	SBCID BOT VERSION
10005	SBCID NET LATENCY
10006	SBCID TCPPORT S1
10007	SBCID PATH SOURCE

Item ID	Value
10008	SBCID PATH DEST
10009	SBCID TIME SYSTEM
10010	SBCID TIME TICK
10011	SBCID TIME LOCALBIAS
10012	SBCID OS INFO
10013	SBCID LANGUAGE ID
10014	SBCID PROCESS NAME
10015	SBCID PROCESS USER
10016	SBCID IPV4 ADDRESSES
10017	SBCID IPV6 ADDRESSES
10018	SBCID BOTLOG TYPE
10019	SBCID BOTLOG

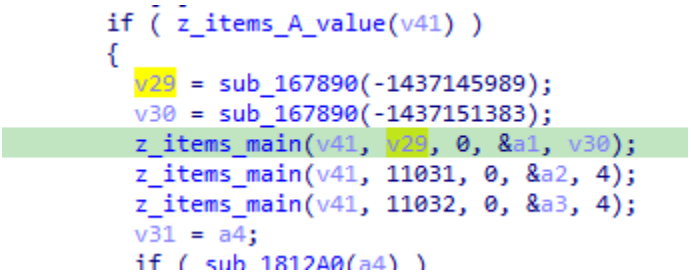
To get to Zeus item values and function we need to search strings in IDA to find one the common ID values since they are constant.



Notice that most the calls are **sub_1657B0**, let's call it z_items_main, it's been crossed referenced 17 times. List of Zeus items being found in 123 variant.

Item ID	Value
10001	SBCID BOT ID
10003	SBCID BOT VERSION
10006	SBCID_PING
10007	SBCID PATH SOURCE
11014	SBCID_GET_FILE
11015	SBCID_GET_FILE_VER
11031	SBCID_LOG_ID_EXT
11032	SBCID_LOG_ERR_CODE
11033	SBCID_LOG_MSG
10022	SBCID_DEBUG
10025	SBCID_MARKER
20001	CFGID_LAST_VERSION
20000	SBCID_BOTLOG
20005	CFG_HTTP_FILTER
20006	CFGID_HTTP_POSTDATA_FILTER
20008	CFGID_DNS_LIST

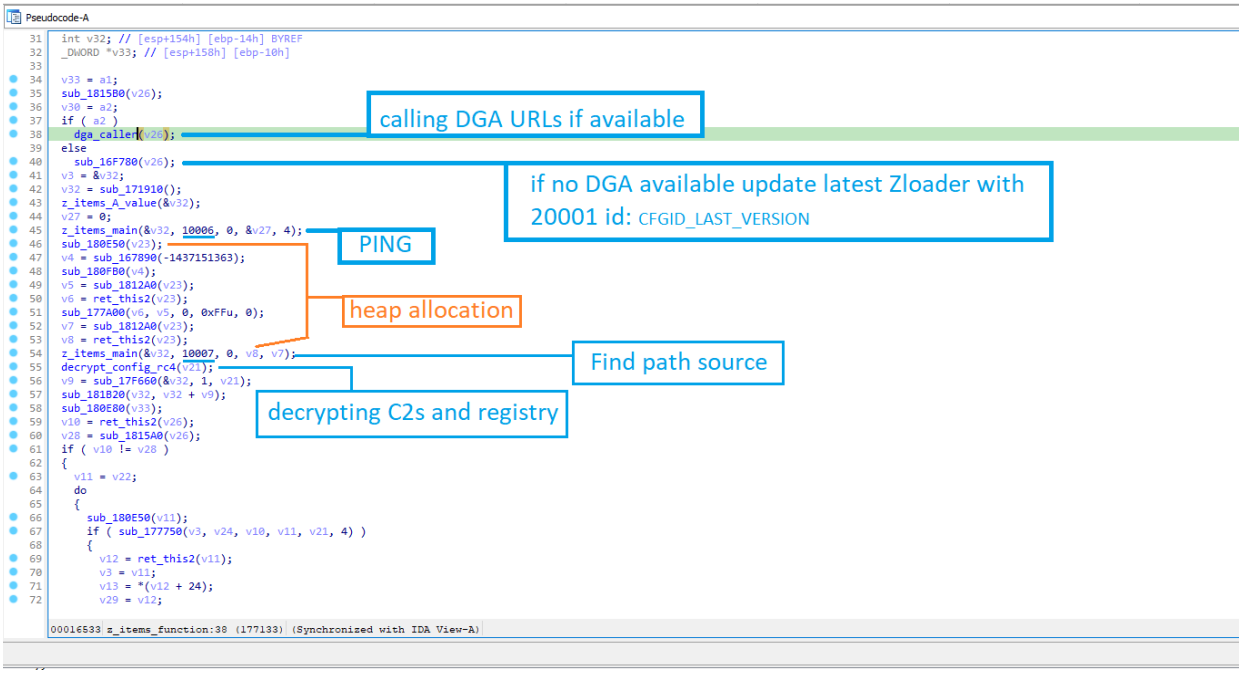
Just to give an example of the level of obfuscation on every stage of Zloader. Not all the items ID values are retrieved in decimal passed to the function. Some values passed into another function and require to calculate separately like below in v29 value return from **sub_167890** .

- 
```
if ( z_items_A_value(v41) )
{
  v29 = sub_167890(-1437145989);
  v30 = sub_167890(-1437151383);
  z_items_main(v41, v29, 0, &a1, v30);
  z_items_main(v41, 11031, 0, &a2, 4);
  z_items_main(v41, 11032, 0, &a3, 4);
  v31 = a4;
  if ( sub_1812A0(a4) )
```

Pseudocode-A

```
1  unsigned int   cdecl sub_167890(int a1)
2  {
3    unsigned int result; // eax
4    unsigned int v2; // esi
5    char v3; // bl
6    unsigned int v4; // edx
7    int v5; // edi
8    void *v6; // edi
9
10   result = a1 ^ 025225547555;
11   v2 = a1 & (a1 ^ 025225547555);
12   v3 = v2 * (a1 ^ 0155) * (a1 & (a1 ^ 0155));
13   v4 = v3 + (a1 ^ 025225547555) * v2;
14   if ( v3 != a1 )
15   {
16     v2 = v4 + 973;
17     v5 = (v4 + 973) * v3;
18     v3 = result & ((v4 - 51) * v3);
19     v4 = v5;
20   }
21   v6 = 0;
22   if ( a1 ^ v3 | a1 ^ v2 )
23     v6 = v4;
24   if ( v4 == a1 )
25     v6 = v4;
26   pvReserved = v6;
27   return result;
28 }
```

To give an example of how Zeus item works let take a look at this function **sub_177110**.



sub_16F780 has (*20001: CFGID_LAST_VERSION*) that updates Zloader version. Looking at it in the disassembler would show so much obfuscated function, but to have an idea of what possibly this update could do let's see the leaked source code having this similar Item ID in similar fashion [9].

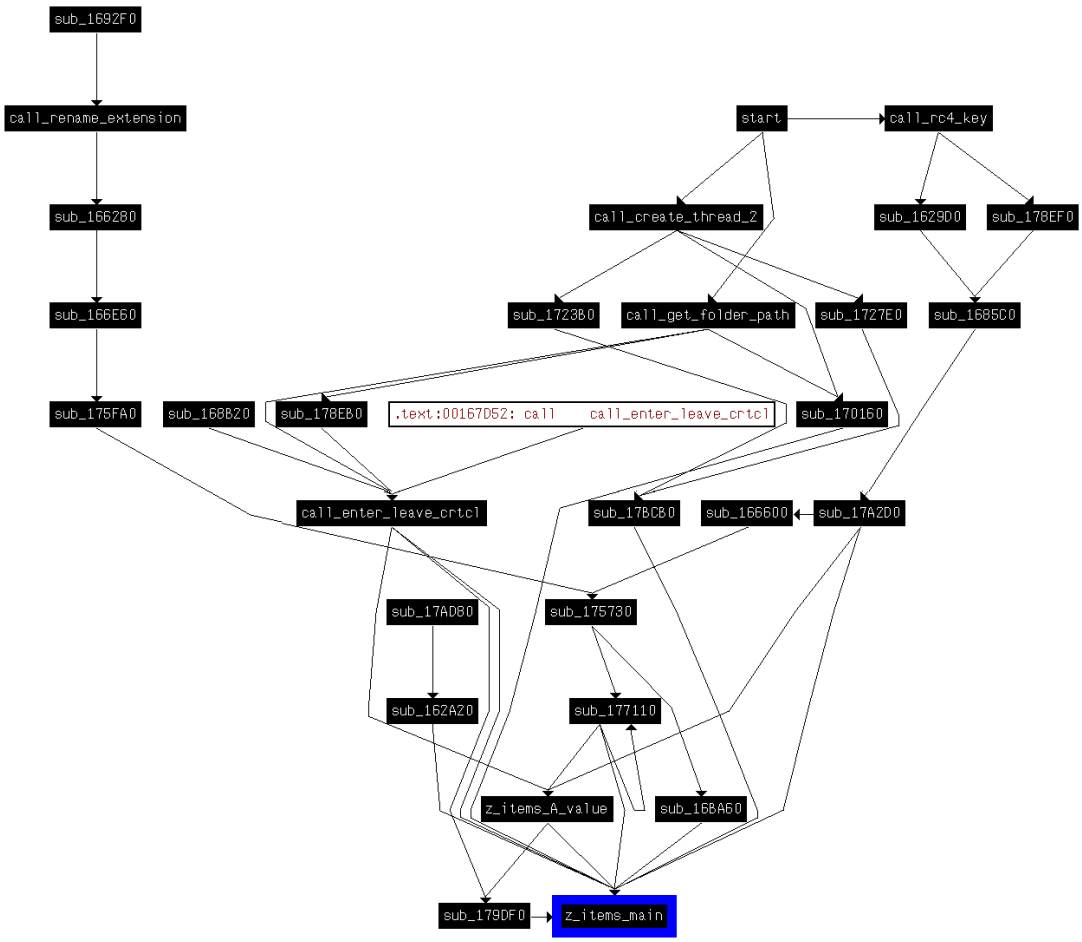
```
141   if(md->data != NULL &&
142      (md->size = BinStorage::_unpack(NULL, md->data, md->size, &baseConfig->baseKey)) != 0 &&
143      BinStorage::_getItemDataAsDword((BinStorage::STORAGE *)md->data, CFGID_LAST_VERSION, BinStorage::ITEMF_IS_OPTION, NULL))
144   {
145     WDEBUG0(WDDT_INFO, "Configuration unpacked.");
146
147     //Update the configuration.
148     {
149       PESETTINGS pes;
150       DWORD type;
151
152       initRegistry();
153       Core::getPeSettings(&pes);
154
155       if((md->size = BinStorage::_pack((BinStorage::STORAGE **)&md->data, BinStorage::PACKF_FINAL_MODE, &pes.rc4Key)) == 0 ||
156          Registry::_setValueAsBinary(HKEY_CURRENT_USER, registryKey, registryValue, REG_BINARY, md->data, md->size) == false)
157       {
158         WDEBUG0(WDDT_ERROR, "Failed to repack configuration.");
159       }
160       else
161       {
162         retVal = true;
163         tryToUpdateBot(flags & UCF_FORCEUPDATE);
164       }
165     }
166   }
```

id: 20001

If update successful it changes registry values

The successful update would lead to update registry values in HKCU\Software\Mircosoft*bbxk*\ which points to %AppData% directory of possible the new Zloader that has new C2 connections

Finally, to have an idea how Zeus function being called here's a mind map when Xref-to it.



Appendix A

Address	XORed string
rdata:00183DED	kernel32.dll
rdata:00183DFA	http
rdata:00183E26	post
rdata:00183E37	.63
rdata:00183E3B	Wininet.dll
rdata:00183DE0	Imagehlp.dll
rdata:00183DB0	C:\Windows\SystemApps
rdata:00183D9C	Local
rdata:00183D92	.exe
rdata:00183D50	NtQueryVirtualMemory
rdata:00183D43	Bcrypt.dll
rdata:00183D38	Ftllib.dll
rdata:00183D2A	Samlib.dll
rdata:00183D20	Post.php
rdata:00183E47	Ntdll.dll
rdata:00183E5C	CmpMem64
rdata:00183E70	INVALID_BOT_ID
rdata:00183E8E	\start
rdata:00183EA0	HideClass
rdata:00183EB4	advapi32.dll
rdata:00183ED0	ABCDEFGHIJKLMNOPQRSTUVWZabcdefghijklmnopqrstuvwz
rdata:00183F21	ws2_32.dll
rdata:00183F3B	Shlwapi.dll
rdata:00183F47	crypt32.dll
rdata:00183F60	NtProtectVirtualMemory
rdata:00183F77	GetMem64

Address	XORed string
rdata:00183F90	Get
rdata:00183FA0	Software\Microsoft\Windows\CurrentVersion\Run
rdata:00183FFC	Urlmon.dll
rdata:0018400A	wtsapi32.dll
rdata:00184040	Software\Microsoft
rdata:00184068	tmp
rdata:0018407C	Iphlpapi.dll
rdata:0018408C	Version.dll
rdata:0018409E	rpcrt4.dll
rdata:001840AA	Dll
rdata:00184111	wldap32.dll
rdata:00184165	ole32.dll
rdata:0018416F	psapi_dll
rdata:00184180	NtFreeVirtualMemory
rdata:001841A0	NtSetContextThread
rdata:001841B3	Winsta.dll
rdata:001841D0	user32.dll
rdata:001841E0	Software\Microsoft\WindowsNT\CurrentVersion
rdata:00184288	gdi32.dll
rdata:00184292	Gdiplus.dll
rdata:001842C0	regsvr32.exe
rdata:001842F0	RtlCreateUserProcess
rdata:00184310	NtWriteVirtualMemory
rdata:00184330	InstallDate
rdata:001843B0	NtReadVirtualMemory
rdata:001843E0	RtlCreateProcessParameters

Address	XORed string
rdata:00184780	Connection_close
rdata:00184794	Dnsapi.dll
rdata:001847BC	secur32.dll
rdata:001847D0	kernel32.dll
rdata:001847F0	NtGetContextThread
rdata:00184820	Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36.
rdata:00184892	NtResumeThread
rdata:001848B0	SeSecurityPrivilege
rdata:00184914	shell32.dll
rdata:00184920	Ntdll.dll
rdata:00184940	LdrGetProcedureAddress
rdata:00184957	netapi32.dll
rdata:00184964	Mpr.dll
rdata:0018496C	https:\\
rdata:00184975	X64Call
rdata:00184980	NtAllocateVirtualMemory
rdata:001849B4	.com
rdata:001849BA	Global
rdata:001849CA	Winscard.dll
rdata:001849D7	Cabinet.dll
rdata:001849E3	Userenv.dll
rdata:001849EF	Ncrypt.dll

References

[1] Zeus opensource, https://github.com/Visgean/Zeus

[2] Titans' revenge: detecting Zeus via its own flaws, https://www.honeynet.it/wp-content/uploads/Papers/04-Titans%20revenge.pdf

[3] Hide and Seek | New Zloader Infection Chain Comes With Improved Stealth and Evasion Mechanisms, https://www.sentinelone.com/labs/hide-and-seek-new-zloader-infection-chain-comes-with-improved-stealth-and-evasion-mechanisms/

[4] The Squirrel Strikes Back: Analysis of the newly emerged cobalt-strike loader "SquirrelWaffle" , https://elis531989.medium.com/the-squirrel-strikes-back-analysis-of-the-newly-emerged-cobalt-strike-loader-squirrelwaffle-937b73dbd9f9

[5] QakBot Quick analysis, https://twitter.com/aaqeel87/status/1443255927000424449?s=20

[6] HashDB project, https://hashdb.openanalysis.net/#section/Using-The-API/Hash-Format

[7] The "Silent Night" Zloader/Zbot , https://www.malwarebytes.com/resources/files/2020/06/the-silent-night-zloader-zbot_final.pdf

[8] The DGA of Zloader, https://bin.re/blog/the-dga-of-zloader/

[9] Zeus source code, https://github.com/Visgean/Zeus/blob/c55a9fa8c8564ec196604a59111708fa8415f020/source/client/dynamicconfig.cpp