Keeping an eye on CloudEyE (GuLoader) - Reverse engineering the loader

Archived: 2026-04-05 23:47:06 UTC

CloudEye (originally GuLoader) is a small malware downloader written in Visual Basic that's used in delivering all sorts of malicious payloads to victim machines. Its primary function is to download, decrypt and run an executable binary off a server (commonly a legitimate one like Google Drive or Microsoft OneDrive).

At the time of writing this article, the malicious code can be split into two parts:

- The core of the program that performs VM checks, downloads the code, decrypts and runs it
- A small wrapper that hides the core by encrypting it with a simple xor algorithm

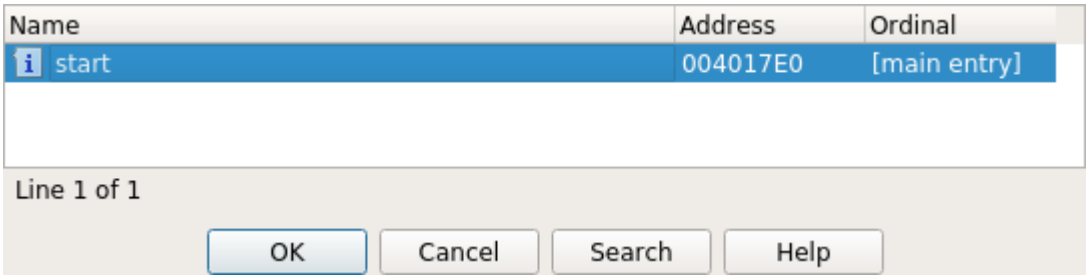
While the outer layer is pretty tiny and straightforward, mimicking it and manually unpacking the core can be a bit of a headache. In this article, we'll explain how one can leverage IDA Pro functionalities to simplify this process.

Sample analysed:

- ZAPYTANIE_OFERTOWE_KPM_03-022021.exe - `41b83b01b84f95653e2cc9467a6d509c291ab6e834bdf786e75616d51ad45164` (VirusTotal, MWDB)

The first thing you want to do while reverse-engineering Visual Basic binaries in IDA is grab a copy of vb.idc. It's a super useful IDA script that parses the embedded VB metadata and provides you with much more information about the binary than the original analysis.

Compare the number of detected event entry points before running the script:



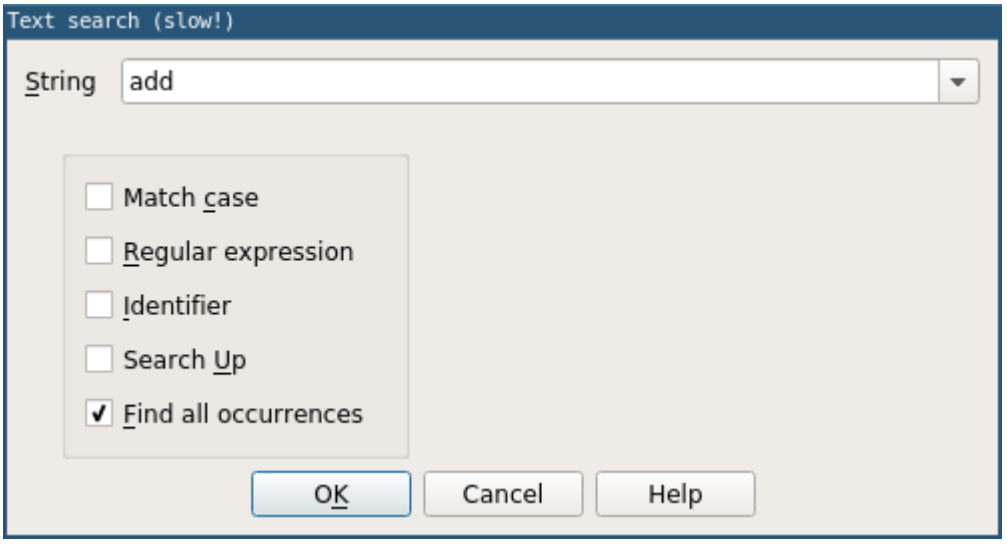
And after:

Name	Address	Ordinal
start	004017E0	[main entry]
_O_Pub_Obj_Inf1_Method0x1	00413650	
_O_Pub_Obj_Inf1_Event0x2	00413941	
_O_Pub_Obj_Inf1_Event0x3	004139F7	
_O_Pub_Obj_Inf1_Event0x4	00413AD7	
_O_Pub_Obj_Inf1_Event0x5	00413B74	
_O_Pub_Obj_Inf1_Event0x6	00413D9F	
_O_Pub_Obj_Inf1_Event0x7	00413FFB	
_O_Pub_Obj_Inf1_Event0x8	0041407E	
_O_Pub_Obj_Inf1_Event0x9	004147BB	
_O_Pub_Obj_Inf1_Event0xA	00414900	
_O_Pub_Obj_Inf1_Event0xB	00415584	
_O_Pub_Obj_Inf1_Event0xC	0041571A	
_O_Pub_Obj_Inf1_Event0xD	00415D31	
_O_Pub_Obj_Inf1_Event0xE	00413034	
_O_Pub_Obj_Inf1_Event0xF	004130A0	
_O_Pub_Obj_Inf1_Event0x10	004131E6	
_O_Pub_Obj_Inf1_Event0x11	004133FD	
_O_Pub_Obj_Inf1_Event0x12	004134F0	
_O_Pub_Obj_Inf1_Event0x13	00413557	
_O_Pub_Obj_Inf1_Event0x14	0041371B	
_O_Pub_Obj_Inf1_Event0x15	004138D2	
_O_Pub_Obj_Inf1_Event0x16	00414244	
_O_Pub_Obj_Inf1_Event0x17	00414425	
_O_Pub_Obj_Inf1_Event0x18	00414B66	
_O_Pub_Obj_Inf1_Event0x19	00414BE8	
_O_Pub_Obj_Inf1_Event0x1A	00414E11	
_O_Pub_Obj_Inf1_Event0x1B	00414EB7	

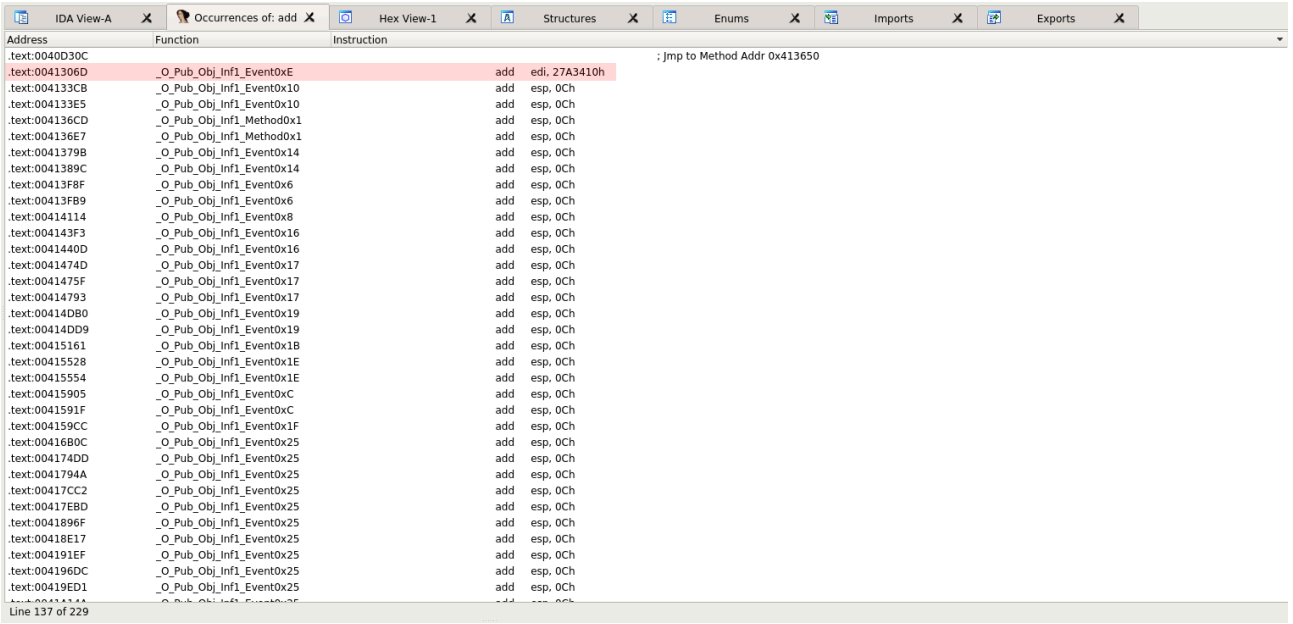
OK Cancel Search Help

Locating the malware entry point is still not trivial, though. You can iterate over all discovered entry points and judge if there's anything suspicious or not, but that can become quite tedious, and you can still miss some better-hidden code.

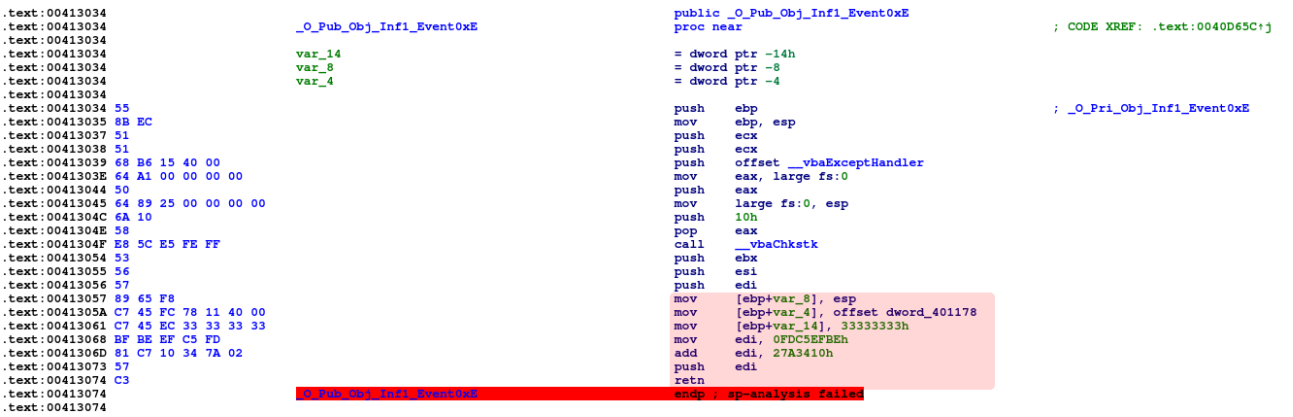
Sometimes, a good method is to search for all `add` instructions and find the "odd one" with a large immediate value sticking out. You can do that in IDA either by selecting `Search` -> `Text` or if you're aspiring to be a power-user: by quickly tapping `Alt` + `t`.



Make sure you check the `Find all occurrences` box, this will take IDA a bit longer, but it will allow you to inspect all matches at once.



Now, navigate to the function in question:

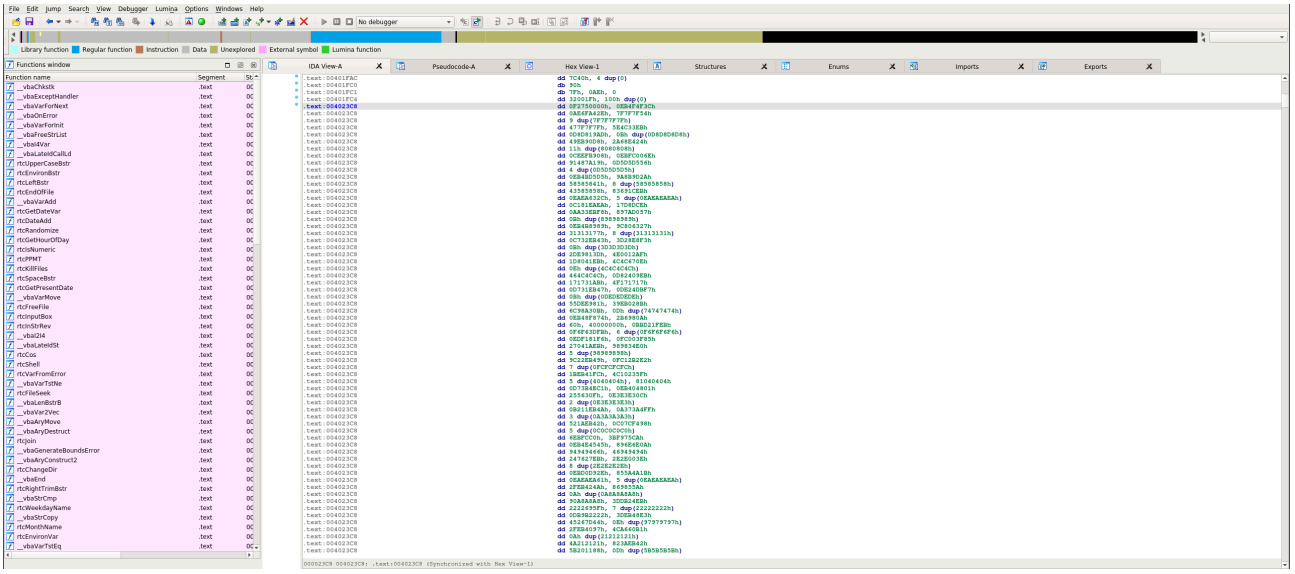


And press `tab` to see the matching decompilation (as usual, the decompiler does most of the job for us):

```
1  void O_Pub_Obj_Inf1_Event0xE()
2  {
3    void *v0; // esp
4    _DWORD v1[11]; // [esp-18h] [ebp-2Ch] BYREF
5
6    v0 = alloca(16);
7    v1[9] = v1;
8    v1[10] = dword_401178;
9    v1[6] = 858993459;
10   JUMPOUT(0x4023CE);
11 }
```

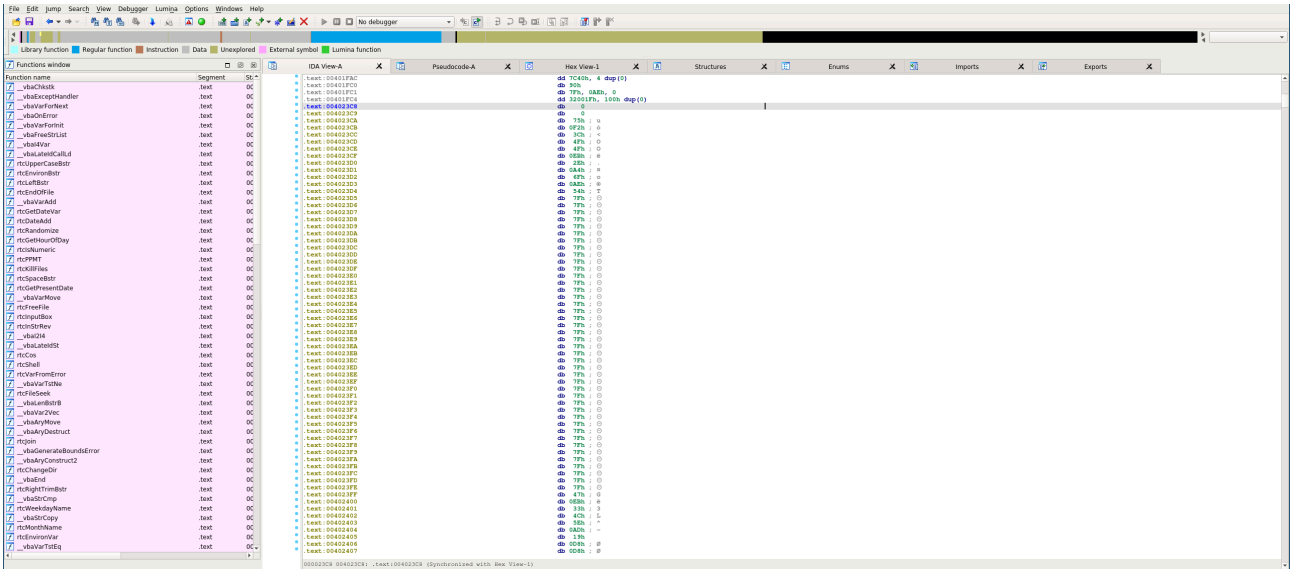
With the malware entry address `0x4023CE` we can now begin analyzing the real loader. Let's jump to the entry address by selecting `Jump` -> `Jump to address` (shortcut `g`)

Surprisingly, there's no code there, just a bunch of data.

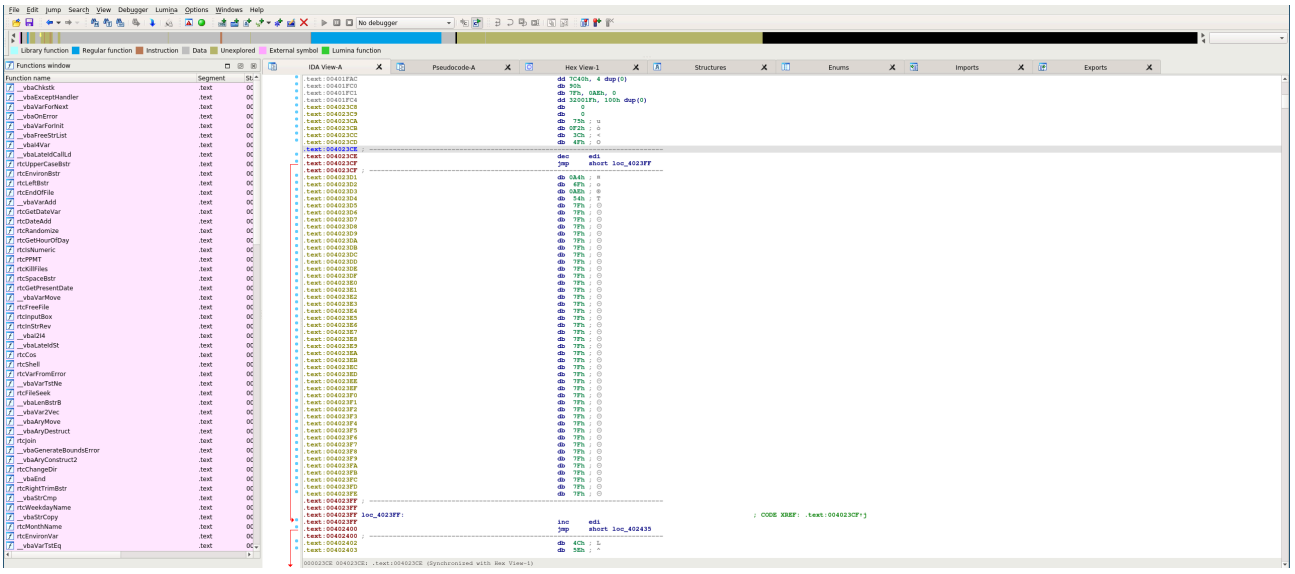


That's because IDA didn't know to follow the code reference; to fix that, we'll have to mark the data as code ourselves. Start by undefining the fragment `Edit` -> `Undefine` (shortcut `u`)

This will split the large chunk of data into single-byte lines:

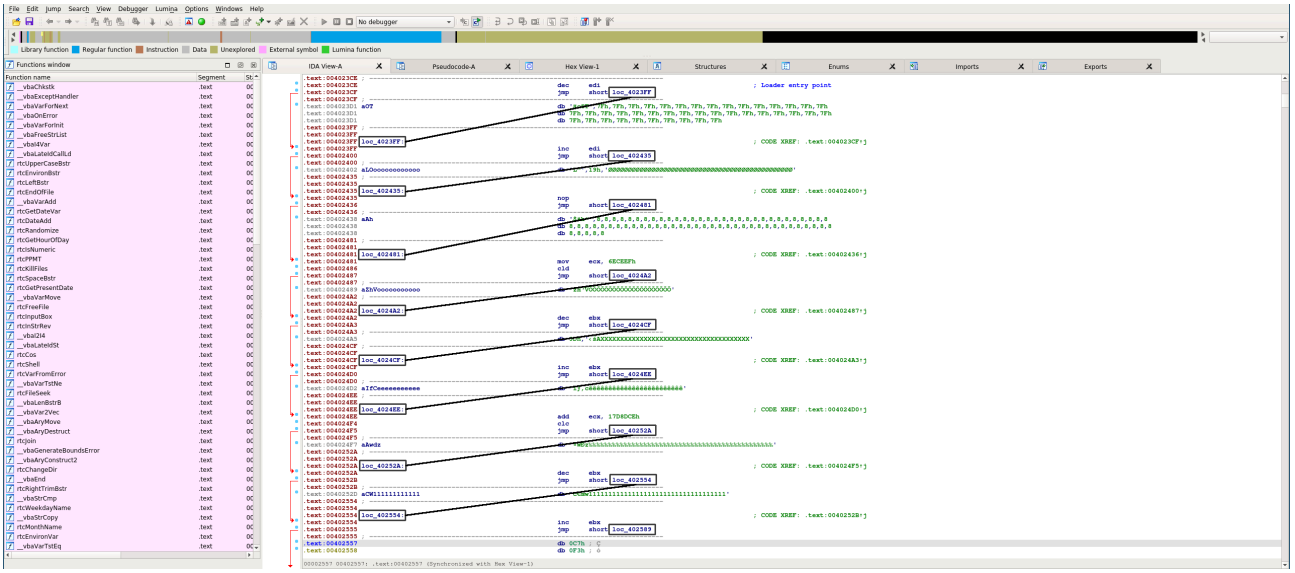


Now we can once again jump to `0x4023CE` (the original line contained many bytes, and IDA doesn't know which one to follow and decides to stay on the first byte) and mark the data as code `Edit` -> `Code` (shortcut `c`).



This will automatically disassemble all reachable code blocks and functions.

We can almost immediately notice that this isn't an ordinary function, but something rather weird is going on: there are many jumps with random data between them. We can clear it up a bit by grouping the data between code blocks together and adding a few arrows:



Some of our avid readers will surely recognize this pattern from our [Dissecting Smoke Loader](#) article. The main takeaway was that while manually reconstructing the code flow and creating a new disassembly is possible; usually, the best method is to let IDA decompiler deal with such obfuscations.

But if we try to create a new function at the start address (`Edit` -> `Functions` -> `Create Function` shortcut `p`), all we get is this annoying error message:

```
.text:00402FEE: The function has undefined instruction/data at the specified address.
Your request has been put in the autoanalysis queue.
```

That's because IDA wasn't able to disassemble the code at the given address; let's see what the fuzz is about.

```
.text:00402FD0                                          db  58h ; X
.text:00402FD1                                          db  58h ; X
.text:00402FD2 ; ---------------------------------------------------------------------------
.text:00402FD2
.text:00402FD2 loc_402FD2:                                                          ; CODE XREF: .text:00402F9B↑j
.text:00402FD2                                          cld
.text:00402FD3                                          jmp     short near ptr dword_402FEC+2
.text:00402FD3 ; ---------------------------------------------------------------------------
.text:00402FD5                                          db 0EFh ; ï
.text:00402FD6                                          db 0A1h ; ¡
.text:00402FD7                                          db 0C4h ; Ä
.text:00402FD8                                          db  54h ; T
.text:00402FD9                                          db    0
.text:00402FDA                                          db    0
.text:00402FDB                                          db    0
.text:00402FDC                                          db    0
.text:00402FDD                                          db    0
.text:00402FDE                                          db    0
.text:00402FDF                                          db    0
.text:00402FE0                                          db    0
.text:00402FE1                                          db    0
.text:00402FE2                                          db    0
.text:00402FE3                                          db    0
.text:00402FE4                                          db    0
.text:00402FE5                                          db    0
.text:00402FE6                                          db    0
.text:00402FE7                                          db    0
.text:00402FE8                                          db    0
.text:00402FE9                                          db    0
.text:00402FEA                                          db    0
.text:00402FEB                                          db    0
.text:00402FEC dword_402FEC                             dd 8390000h, 0FC71F0Fh                  ; CODE XREF: .text:00402FD3↑j
.text:00402FEC                                          dd 25EBC61Fh, 5634EAFAh
.text:00402FEC                                          dd 8 dup(0C0C0C0Ch)
.text:00402FEC                                          dd 0EF850F0Ch, 0EBFFFFFEh
.text:00402FEC                                          dd 0BE57B247h, 5F5F5F29h
.text:00402FEC                                          dd 10h dup(5F5F5F5Fh)
.text:00402FEC                                          dd 45EB474Fh, 468E7316h
.text:00402FEC                                          dd 10h dup(0E6E6E6E6h)
.text:00402FEC                                          dd 2EEBFCE6h, 2A73EE35h
.text:00402FEC                                          dd 0Ah dup(0ADADADADh)
.text:00402FEC                                          dd 0E8B9ADADh, 0EBFFFFEEh
.text:00402FEC                                          dd 0BC5B4F3Bh, 43434361h
.text:00402FEC                                          dd 0Dh dup(43434343h)
.text:00402FEC                                          dd 3FEB434Bh, 89425DCh
.text:00402FEC                                          dd 0Eh dup(0CFCFCFCFh)
.text:00402FEC                                          dd 49CFCFCFh, 5418EB41h
.text:00402FEC                                          dd 0C50BDB86h, 4 dup(0C5C5C5C5h)
.text:00402FEC                                          dd 81C5C5C5h, 0E02C1h
```

Well, yes, it doesn't look too correct. Instructions in the form of `jmp short near ptr <addr>+<number>` should almost always raise a red flag for you. It very often means that the jmp (or any other code-flow-altering instruction) tries to jump into the middle of already defined code/data. In this case, though, it looks like IDA just made an error, and we have to mark the data as code manually, similarly as we had done previously.

```
.text:00402FD0                                          db  58h ; X
.text:00402FD1                                          db  58h ; X
.text:00402FD2 ; ---------------------------------------------------------------------------
.text:00402FD2
.text:00402FD2 loc_402FD2:                                                          ; CODE XREF: .text:00402F9B↑j
.text:00402FD2                                          cld
.text:00402FD3                                          jmp     short loc_402FEE
.text:00402FD3 ; ---------------------------------------------------------------------------
.text:00402FD5                                          db 0EFh ; ï
.text:00402FD6                                          db 0A1h ; ¡
.text:00402FD7                                          db 0C4h ; Ä
.text:00402FD8                                          db  54h ; T
.text:00402FD9                                          db    0
.text:00402FDA                                          db    0
.text:00402FDB                                          db    0
.text:00402FDC                                          db    0
.text:00402FDD                                          db    0
.text:00402FDE                                          db    0
.text:00402FDF                                          db    0
.text:00402FE0                                          db    0
.text:00402FE1                                          db    0
.text:00402FE2                                          db    0
.text:00402FE3                                          db    0
.text:00402FE4                                          db    0
.text:00402FE5                                          db    0
.text:00402FE6                                          db    0
.text:00402FE7                                          db    0
.text:00402FE8                                          db    0
.text:00402FE9                                          db    0
.text:00402FEA                                          db    0
.text:00402FEB                                          db    0
.text:00402FEC                                          db    0
.text:00402FED                                          db    0
.text:00402FEE ; ---------------------------------------------------------------------------
.text:00402FEE
.text:00402FEE loc_402FEE:                                                          ; CODE XREF: .text:00402FD3↑j
.text:00402FEE                                          cmp     [eax], ecx
.text:00402FF0                                          nop     edi
.text:00402FF3                                          nop     esi
.text:00402FF6                                          jmp     short loc_40301D
.text:00402FF6 ; ---------------------------------------------------------------------------
.text:00402FF8                                          db 0FAh ; ú
.text:00402FF9                                          db 0EAh ; ê
.text:00402FFA                                          db  34h ; 4
.text:00402FFB                                          db  56h ; V
.text:00402FFC                                          db  0Ch
.text:00402FFD                                          db  0Ch
.text:00402FFE                                          db  0Ch
.text:00402FFF                                          db  0Ch
```

Good as new! We may have to repeat this several times before we get all parts correct.

At some point, though, we'll come across a fragment that no longer looks like correct x86 code:

```
.text:00403F30                                          db 0EEh ; î
.text:00403F31                                          db 0EEh ; î
.text:00403F32                                          db 0EEh ; î
.text:00403F33 ; ---------------------------------------------------------------------------
.text:00403F33
.text:00403F33 loc_403F33:                                                          ; CODE XREF: .text:00403F07↑j
.text:00403F33                                          inc     edi
.text:00403F34                                          jmp     short loc_403F3F
.text:00403F34 ; ---------------------------------------------------------------------------
.text:00403F36                                          db  92h ; '
.text:00403F37                                          db  85h ; …
.text:00403F38                                          db 0B0h ; °
.text:00403F39                                          db  29h ; )
.text:00403F3A                                          db 0BFh ; ¿
.text:00403F3B                                          db 0BFh ; ¿
.text:00403F3C                                          db 0BFh ; ¿
.text:00403F3D                                          db 0BFh ; ¿
.text:00403F3E                                          db 0BFh ; ¿
.text:00403F3F ; ---------------------------------------------------------------------------
.text:00403F3F
.text:00403F3F loc_403F3F:                                                          ; CODE XREF: .text:00403F34↑j
.text:00403F3F                                                                      ; .text:00403F66↓j
.text:00403F3F                                                                      ; DATA XREF: ...
.text:00403F3F                                          mov     gs, esp
.text:00403F41                                          ficomp  dword ptr [ecx+67h]
.text:00403F44                                          db      65h
.text:00403F44                                          wait
.text:00403F46                                          fstp9   st(6)
.text:00403F48                                          outsb
.text:00403F49                                          mov     eax, 0E6C877Ah
.text:00403F4E                                          dec     ebx
.text:00403F4F                                          aas
.text:00403F50                                          test    dword ptr [ecx-4254BAACh], 5D9C5B03h
.text:00403F5A                                          mov     eax, ds:0BF23D604h
.text:00403F5F                                          inc     edi
.text:00403F60                                          and     eax, 1D8EBBA2h
.text:00403F65                                          xchg    eax, edx
.text:00403F66                                          ja      short near ptr loc_403F3F+1
.text:00403F68                                          push    es
.text:00403F69                                          mov     bl, 95h ; '•'
.text:00403F6B                                          dec     ebp
.text:00403F6C                                          add     al, 0
.text:00403F6E                                          das
.text:00403F6F                                          les     ebp, [edi+eax*2-48BFF411h]
.text:00403F76                                          pop     ebx
.text:00403F77                                          and     eax, 0C9F95435h
.text:00403F77 ; ---------------------------------------------------------------------------
.text:00403F7C                                          db  8Fh
.text:00403F7D                                          db  5Fh ; _
.text:00403F7E                                          db  12h
.text:00403F7F                                          db  41h ; A
```

That's the code that gets decrypted in previous code blocks; naturally, IDA won't decompile invalid instructions. We can get around that using (at least) 2 methods:

- by selecting the code segments, we want to include in our newly-created function
- by patching the last `jmp` instruction to `ret`, which will cut off the last invalid block from our function

We'll go with the first method as it's a bit more elegant and simple; for any adventurous readers the `Edit` -> `Patch program` menu and a good x86 opcode reference (like http://ref.x86asm.net/coder32.html) should be more than enough to try out the other method.

Selecting the whole memory range by dragging the mouse is a bit boring and can sometimes deselect the selected code on its own. We'll use the `Edit` -> `Begin selection` (shortcut `Alt` + `l`) command. Position the cursor just before the final `jmp` instruction, begin the selection, go to the loaders entry point (`0x4023CE`) and create a new function.

If everything goes correctly, the relevant fragment in the sidebar should change its color to blue:

`-->`

And you should be able to tap `Tab` and view the simple decompilation pseudocode:

IDA View-A | Pseudocode-A | Occurrences of: add | Hex View-1 | Structures

```
1 void sub_4023CE()
2 {
3   int v0; // ecx
4   char *v1; // eax
5   void (*v2)(void); // eax
6   int i; // ecx
7
8   v0 = 21564845;
9   do
10  {
11    __asm { finit }
12    --v0;
13  }
14  while ( v0 );
15  v1 = (char *)&rtcCos;
16  do
17    --v1;
18  while ( *(_DWORD *)v1 != 9460301 );
19  v2 = (void (*)(void))(*((int (__stdcall **)(_DWORD, int, int, int))v1 + 1075))(0, 40960, 4096, 64);
20  for ( i = 0; i != 22396; i = i - 40 + 44 )
21    *(_DWORD *)((char *)v2 + i) = _mm_cvtsi64_si32(
22                                    _m_pxor(
23                                      _mm_cvtsi32_si64(*(_DWORD *)((char *)&loc_403F3F + i)),
24                                      _mm_cvtsi32_si64(0x59DA0A67u)));
25  v2();
26  JUMPOUT(0x403F34);
27 }
```

The logic is actually quite simple, but the code can get much more bloated and confusing in other samples.

```
 void sub_4023CE()
{
  int v0; // ecx
  char *v1; // eax
  void (*v2)(void); // eax
  int i; // ecx

  v0 = 21564845;
  do
  {
    __asm { finit }
    --v0;
  }
  while ( v0 );
  v1 = &rtcCos;
  do
    --v1;
  while ( *v1 != "\x90ZM" );
  v2 = (*(v1 + 1075))(0, 40960, 4096, 64);
  for ( i = 0; i != 22396; i = i - 40 + 44 )
    *(v2 + i) = _mm_cvtsi64_si32(_m_pxor(_mm_cvtsi32_si64(*(&loc_403F3F + i)), _mm_cvtsi32_si64(0x59DA0A67u)));
  v2();
  JUMPOUT(0x403F34);
}
```

Going step by step:

```
 v0 = 21564845;
do
{
  __asm { finit }
  --v0;
}
```

This is a simple sleep snippet, nothing super interesting there.

```
 v1 = &rtcCos;
do
  --v1;
while ( *v1 != "\x90ZM" );
v2 = (*(v1 + 1075))(4300, 0, 0, 40960, 4096, 64);
```

This is a bit more interesting, it fetches the pointer to `rtcCos` from `MSVBVM60.DLL` and then iterates downrange to find the images base address. It then uses that address to calculate a function address by adding `1075` to the

pointer.

If we load the dll in IDA and navigate to the fetched address (`0x732A0000 + 1075 * 4`) we can learn that it's `VirtualAlloc` .

```
.idata:732A10CC ; LPVOID __stdcall VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)
.idata:732A10CC                 extrn VirtualAlloc:dword
.idata:732A10CC                                         ; CODE XREF: sub_732A6702+2A↓p
.idata:732A10CC                                         ; sub_732A6702+43↓p ...
```

So this is all just a sneaky a way of calling it without clearly indicating it in imports. Let's move on.

```
for ( i = 0; i != 22396; i = i - 40 + 44 )
 *(v2 + i) = _mm_cvtsi64_si32(_m_pxor(_mm_cvtsi32_si64(*(&loc_403F3F + i)), _mm_cvtsi32_si64(0x59DA0A67u)));
```

This part copies `22396` bytes from `0x403F3F` into the newly allocated buffer dexoring it with the constant `0x59DA0A67` in the process.

We can get the decrypted core without debugging the binary using a short Python script:

```
 import struct
from malduck import xor

data = xor(key=struct.pack("<I", 0x59DA0A67), data=get_bytes(0x403F3F, 22396))
with open("decrypted.bin", "wb") as f:
    f.write(data)
```

And finally, the program jumps into the newly copied buffer.

```
 v2();
```

Tune in next time to the second part, where we'll describe some of the CloudEyE's functions and discuss how we can extract the download URLs and the encryption key from unpacked samples automatically using Malduck.

References

- https://research.checkpoint.com/2020/guloader-cloudeye/
- https://research.checkpoint.com/2020/threat-actors-migrating-to-the-cloud/
- https://malpedia.caad.fkie.fraunhofer.de/details/win.cloudeye

---

Source: https://cert.pl/en/posts/2021/04/keeping-an-eye-on-guloader-reverse-engineering-the-loader/