References

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

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:

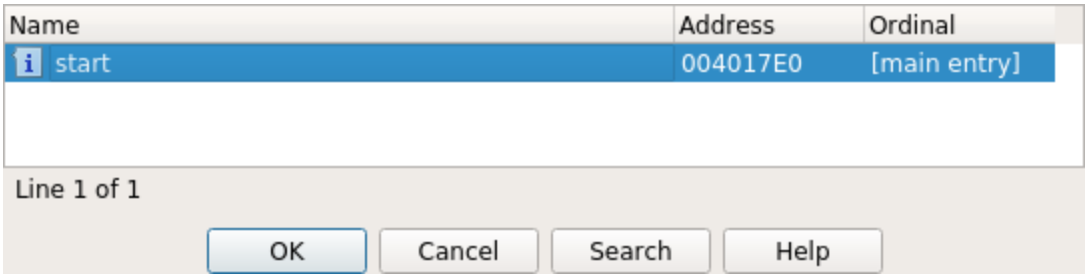
- The core of the program that performs VM checks, downloads the code, decrypts and runs it
- A small wrapper that hides the core by encrypting it with a simple xor algorithm

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.

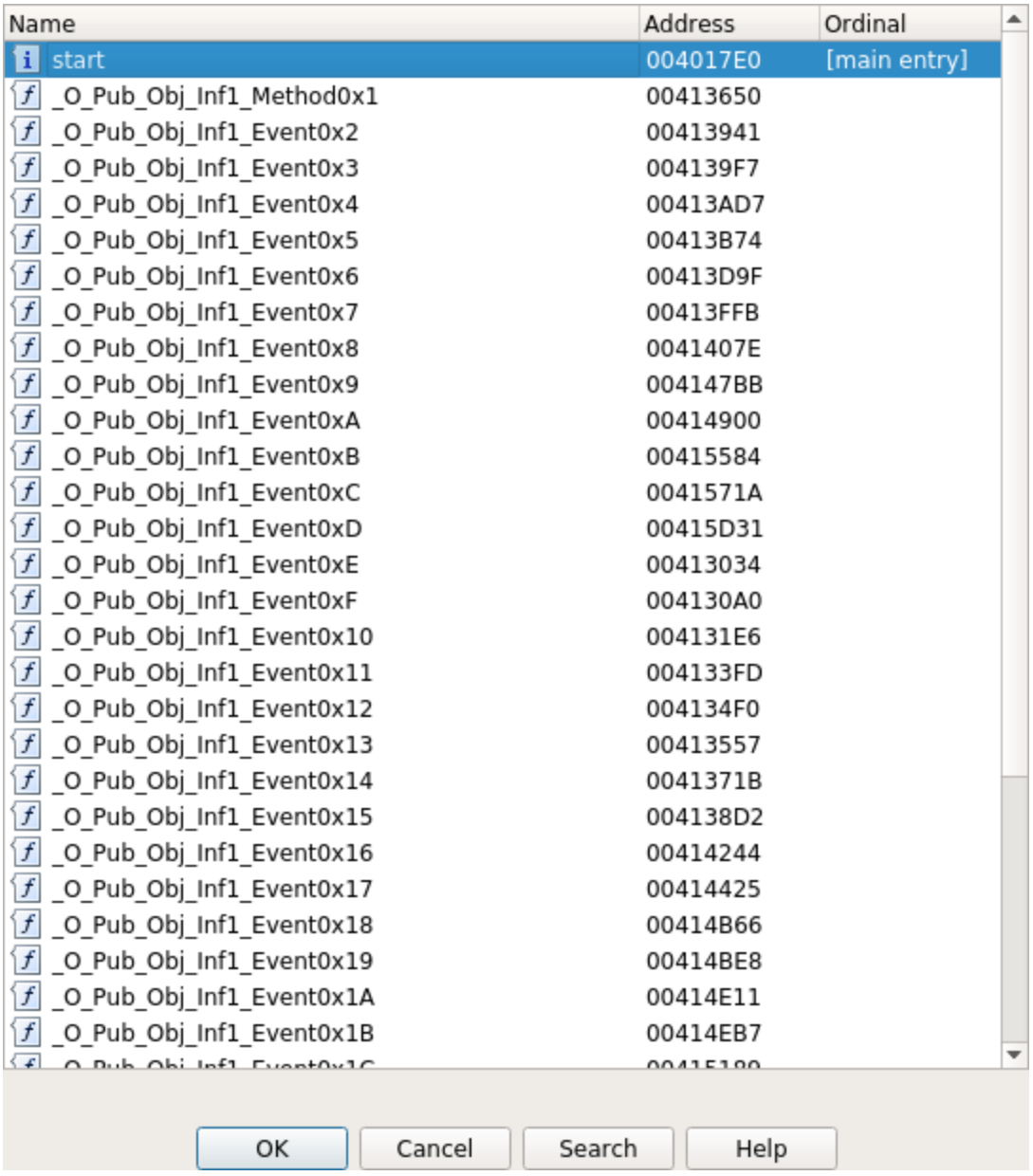
Sample analysed:

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:

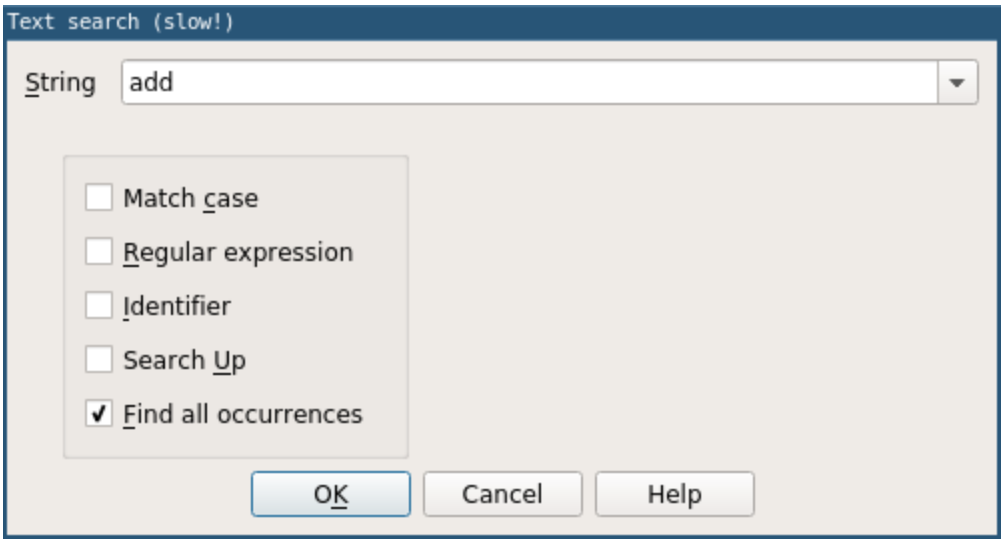


And after:

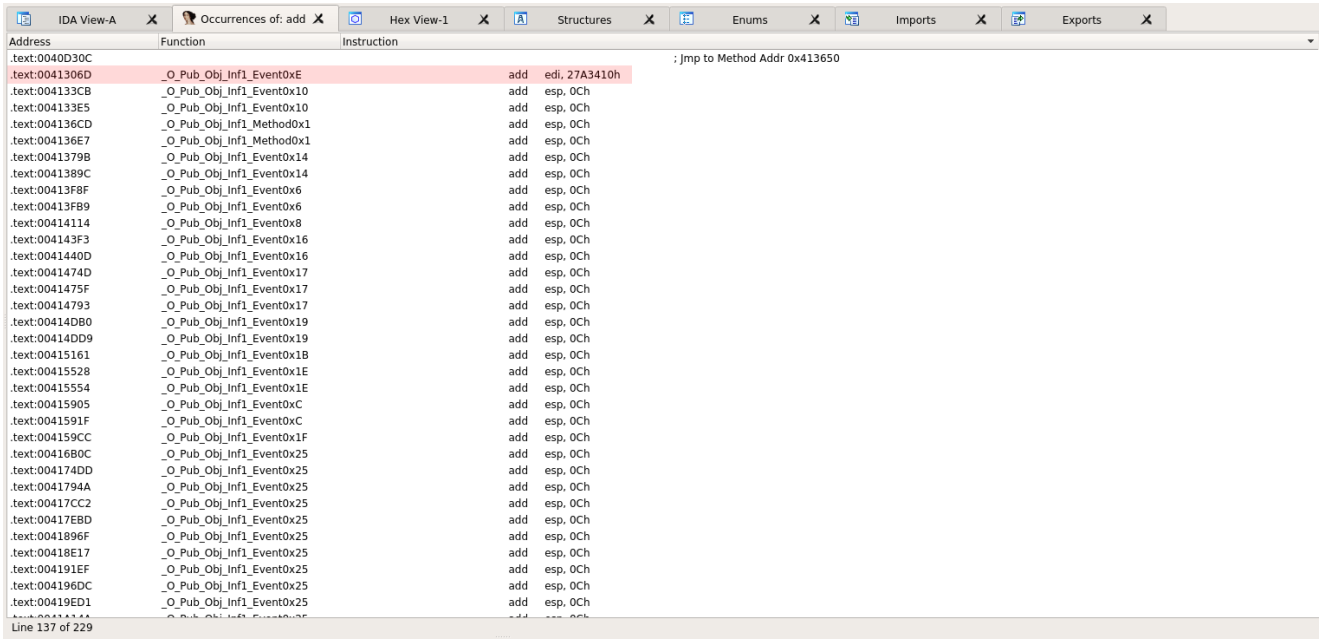


Locating the malware entry point is still not trivial, though. You can iterate over all discovered entry points and judge if there's anything suspicious or not, but that can become quite tedious, and you can still miss some better-hidden code.

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:

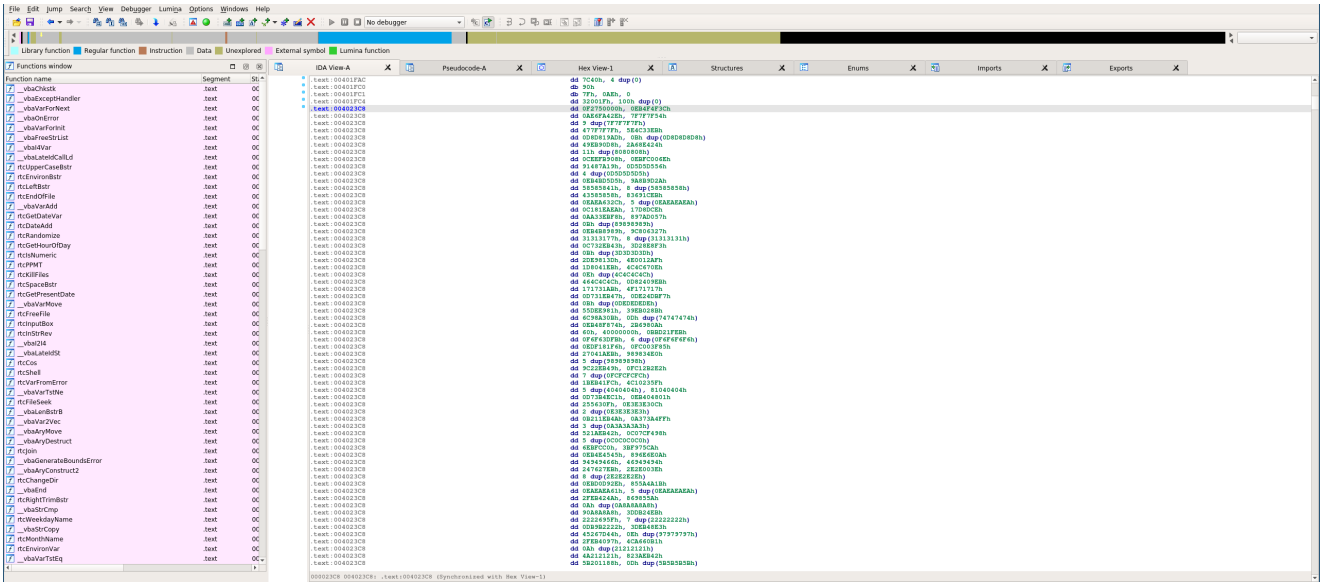
```
.text:00413034                          _O_Pub_Obj_Inf1_Event0xE                 public _O_Pub_Obj_Inf1_Event0xE
.text:00413034                                                                  proc near               ; CODE XREF: .text:0040D65C↑j
.text:00413034
.text:00413034                          var_14                                  = dword ptr -14h
.text:00413034                          var_8                                   = dword ptr -8
.text:00413034                          var_4                                   = dword ptr -4
.text:00413034
.text:00413034 55                                                               push    ebp             ; _O_Pri_Obj_Inf1_Event0xE
.text:00413035 8B EC                                                            mov     ebp, esp
.text:00413037 51                                                               push    ecx
.text:00413038 51                                                               push    ecx
.text:00413039 68 B6 15 40 00                                                   push    offset __vbaExceptHandler
.text:0041303E 64 A1 00 00 00 00                                                mov     eax, large fs:0
.text:00413044 50                                                               push    eax
.text:00413045 64 89 25 00 00 00 00                                             mov     large fs:0, esp
.text:0041304C 6A 10                                                            push    10h
.text:0041304E 58                                                               pop     eax
.text:0041304F E8 5C E5 FE FF                                                   call    __vbaChkstk
.text:00413054 53                                                               push    ebx
.text:00413055 56                                                               push    esi
.text:00413056 57                                                               push    edi
.text:00413057 89 65 F8                                                         mov     [ebp+var_8], esp
.text:0041305A C7 45 FC 78 11 40 00                                             mov     [ebp+var_4], offset dword_401178
.text:00413061 C7 45 EC 33 33 33 33                                             mov     [ebp+var_14], 33333333h
.text:00413068 BF BE EF C5 FD                                                   mov     edi, 0FDC5EFBEh
.text:0041306D 81 C7 10 34 7A 02                                                add     edi, 27A3410h
.text:00413073 57                                                               push    edi
.text:00413074 C3                                                               retn
.text:00413074                          _O_Pub_Obj_Inf1_Event0xE                endp ; sp-analysis failed
.text:00413074
```

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 }
```

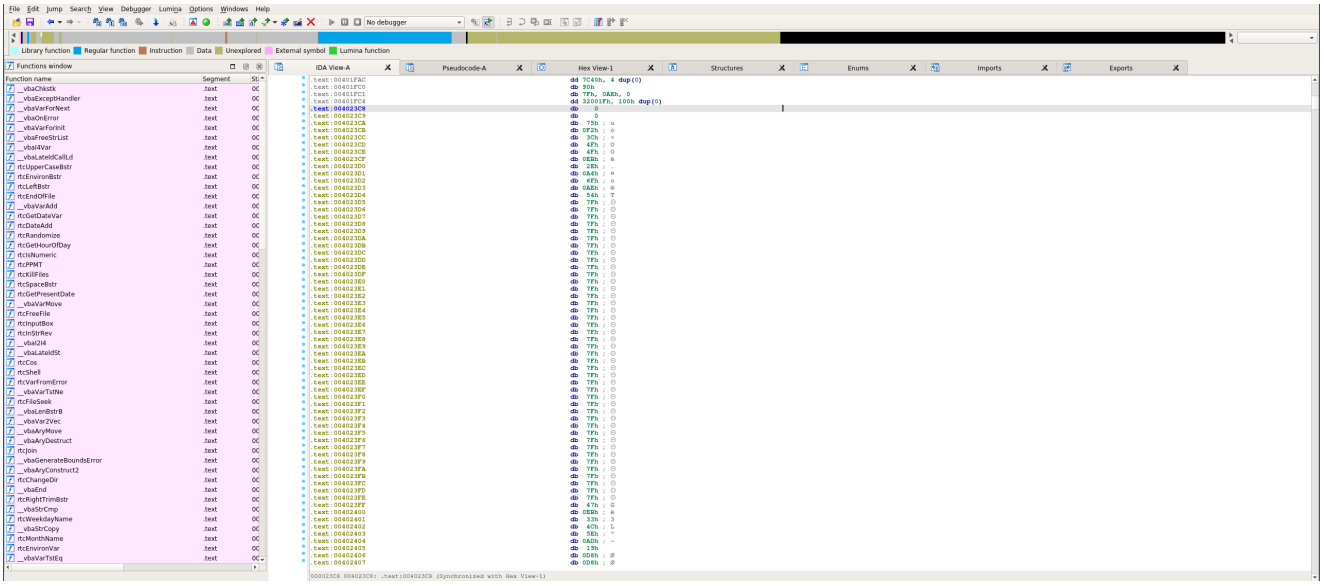
With the malware entry address `0x4023CE` we can now begin analyzing the real loader. Let's jump to the entry address by selecting `Jump` -> `Jump to address` (shortcut `g`)

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

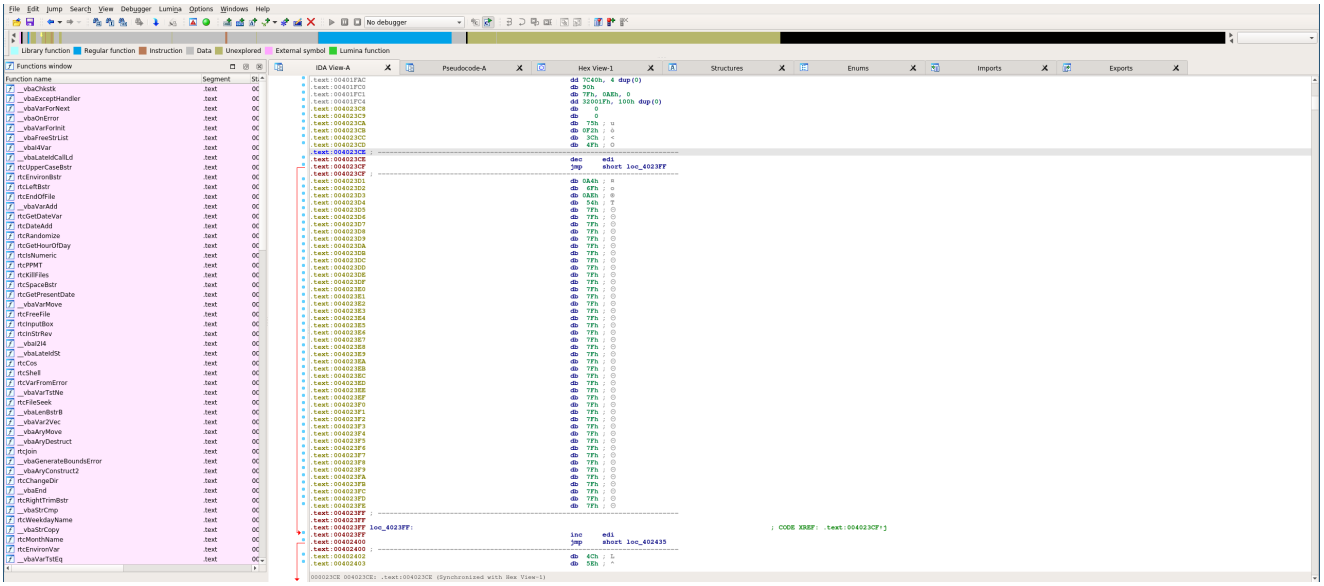


That's because IDA didn't know to follow the code reference; to fix that, we'll have to mark the data as code ourselves. Start by undefining the fragment `Edit` -> `Undefine` (shortcut `u`)

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

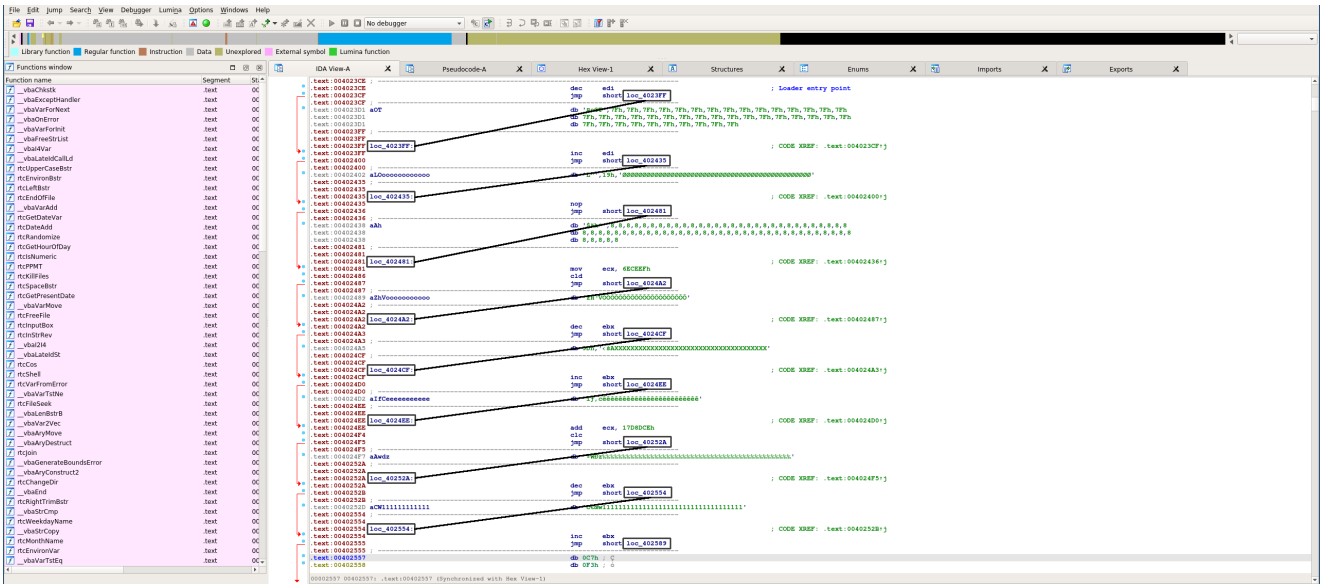


Now we can once again jump to `0x4023CE` (the original line contained many bytes, and IDA doesn't know which one to follow and decides to stay on the first byte) and mark the data as code `Edit` -> `Code` (shortcut `c`).



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.

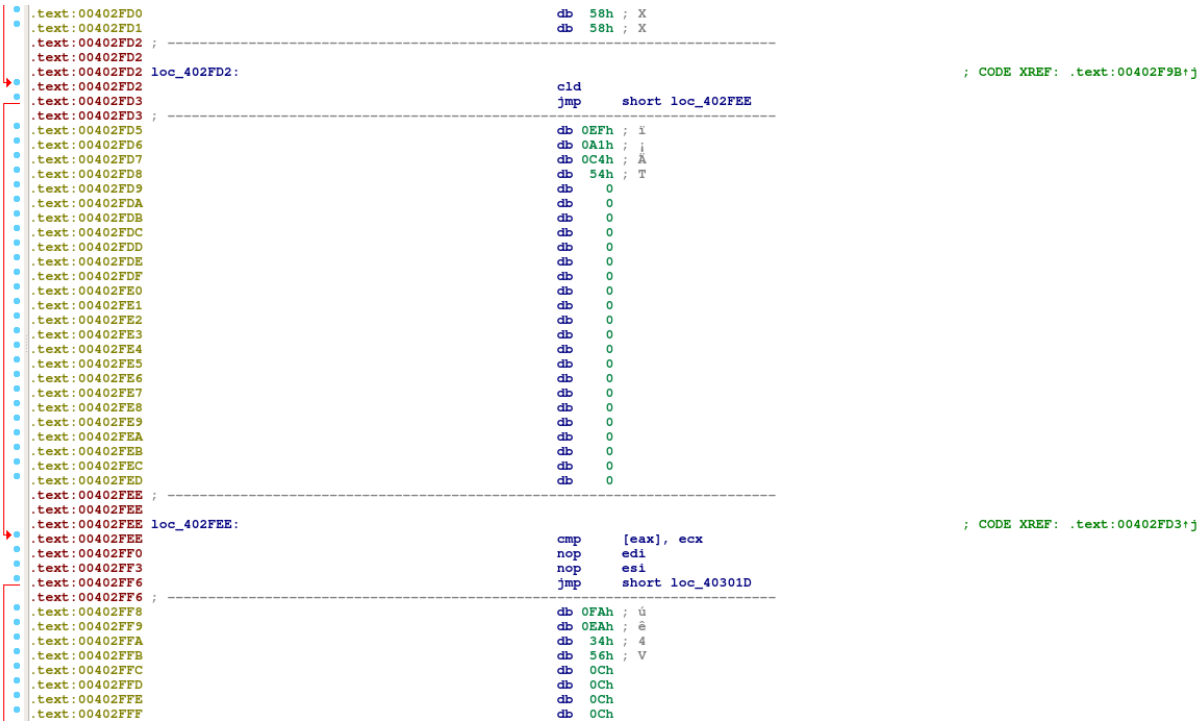
But if we try to create a new function at the start address (`Edit` -> `Functions` -> `Create Function` shortcut `p`), all we get is this annoying error message:

```
.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 ; í
.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.



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
```

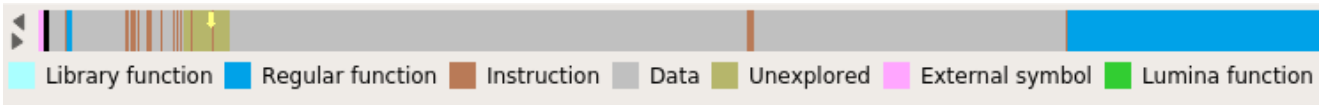
That's the code that gets decrypted in previous code blocks; naturally, IDA won't decompile invalid instructions. We can get around that using (at least) 2 methods:

- 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:

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.