Reversing a NSIS dropper using quick and dirty shellcode emulation

malcat.fr/blog/reversing-a-nsis-dropper-using-quick-and-dirty-shellcode-emulation/

Sample:

e850f3849ea82980cf23844ad3caadf73856b2d5b0c4179847d82ce4016e80ee (Bazaar, VT)

Infection chain:

Excel stylesheet -> Office equation -> Shellcode (downloader) -> NSIS installer -> Shellcode (stage 1) -> Shellcode (stage 2) -> Lokibot

Tools used:

Malcat, Speakeasy emulator

Difficulty:

Easy

The Excel document

The sample we are about to dissect today is an OpenXML Excel document which came as email attachment. The malicious document is very similar to the one we did analyze in our previous blog post: an encrypted OpenXML Excel document embedding an Equation object exploiting CVE-2018-0798. The same author is most likely behind this document as well, they just updated the bait picture:

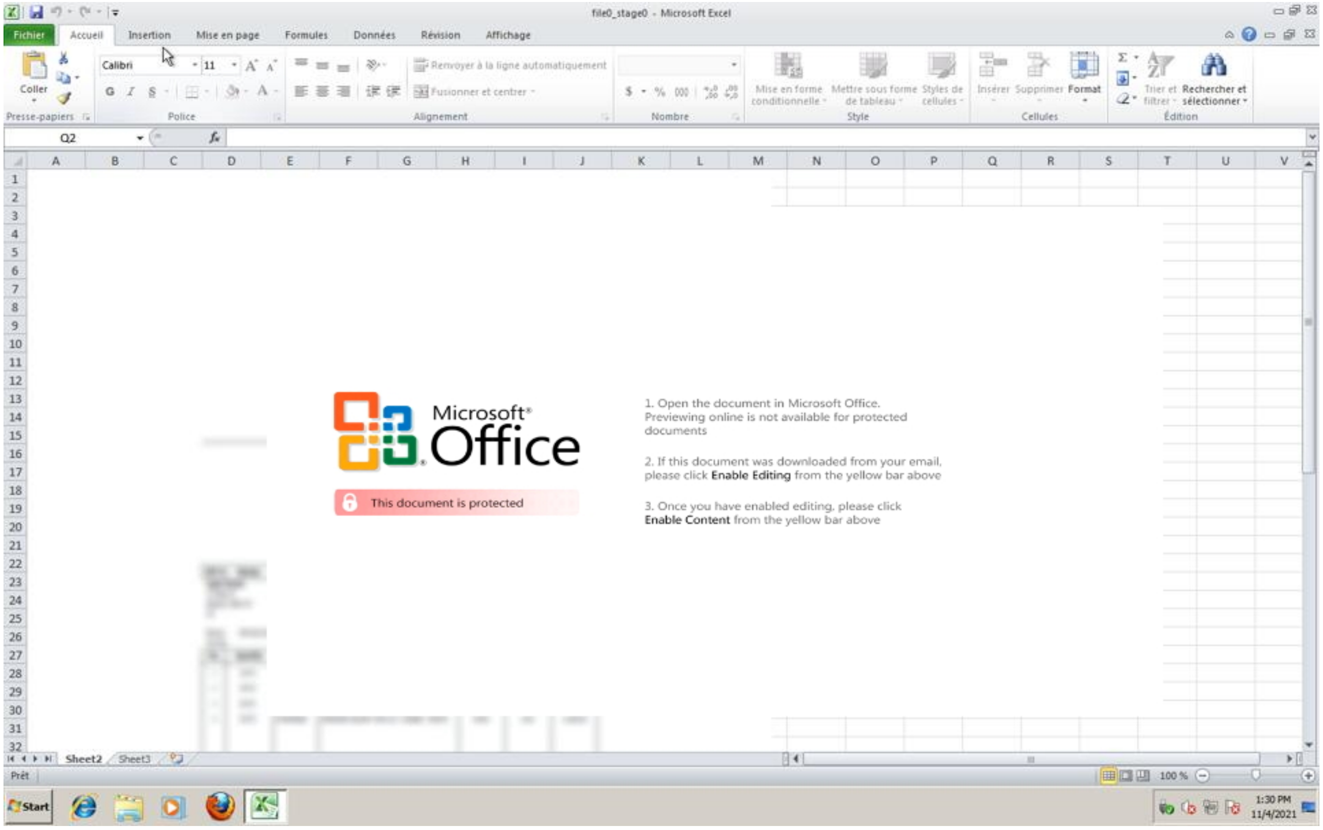

Figure 1: Excel sheet baiting the user to deactivate safe mode

We won't go through the exploit shellcode extraction and decryption process again since the procedure is exactly the same (see here, shellcode offset is also `0x50`). The exploit is again a downloader, downloading from the following url:

```
hxxp://103.153.79.104/windows10/csrss.exe
```

At the time of the analysis, the file is still online. But this time, we don't get a DBatLoader instance, but a NSIS installer instead. So let us fetch the file and have a look at the installer.

NSIS installer

The file `csrss.exe` is a 418KB PE file of sha256 `291df8186e62df74b8fcf2c361c6913b9b73e3e864dde58eb63d5c3159a4c32d` (Bazaar, VT). A NSIS installer is nothing more than a NSIS archive appended to the NSIS PE installer. The file format of the archive, while not very documented, is relatively simple as we will see.

NSIS archive

A NSIS archive is composed of a small NSIS header followed by the archive content. The header does not contain a lot of information:

```
FirstHeader:
    Flags:                                          // some installation
flags
    Signature:                      0xdeadbeef      // NSIS archive start
magic
    Magic:                          "NullsoftInst"  // also magic
    InstallerSize:                  0x6244          // unpacked size of
the setup script
    ArchiveSize:                    0x5e12e         // size of the
archive
```

Like you can see, it does not tell us a lot. Directly following the headers come the "files". I say "files" because they don't really have names, it is more like a list of data bytes or buffers. The files are compressed, and can be stored using two modes:

- the *solid* mode: archive content is a single compressed stream. The unpacked stream is a sequence of N buffers, where each buffer is prefixed by a DWORD telling the size of the buffer.
- the *non-solid* mode: archive content is a sequence of N compressed streams, one for each file. Each compressed stream is prefixed by a DWORD telling the size of the stream.

There is sadly no flag in the header telling us which mode is used, this information is hardcoded inside the NSIS installer executable. The only solution there is trial and error: if the start of the archive starts with a DWORD which could be a size, then it's most likely the *non-solid* mode. If it looks like a compression header, then it's most likely the solid mode. And regarding compression, NSIS supports three compression algorithms:

- LZMA (without headers)
- Zlib
- A custom bzip2 compression algorithm

Malcat supports NSIS files using both solid and non-solid mode for the Zlib and LZMA compression methods, but lacks support for bzip2, since the compression algorithm is custom. But since it's also the least used one, it's not really a big deal. The NSIS archive we are looking at is a solid LZMA archive, so unpacking it is no issue. Like for most archive formats, Malcat lists found files in the upper left corner of the screen, under the *Virtual File System* tree. Double-clicking on a file opens it inside Malcat.

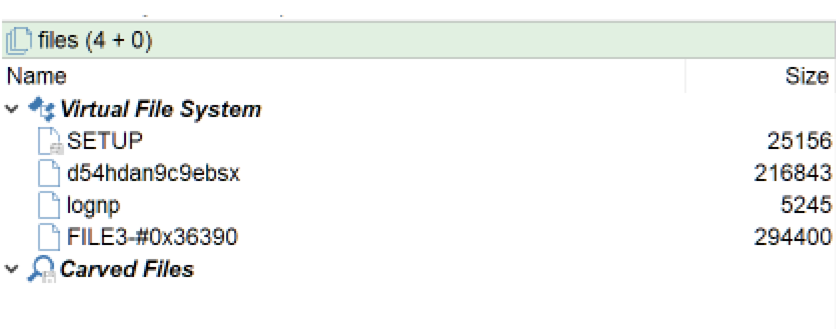

Figure 2: Content of the archive

The first file is always the installer setup script, followed by user-provided files and/or installer plugins. As you can see, Malcat did give name to some of the files (all but the last one) which somehow contradicts what I said before. But these names have been recovered by reversing the SETUP script, and there is no guarantee that it is the real name for these files. Even worse, a buffer in the archive can be extracted under different names on the local filesystem, so don't trust these names 100%.

The SETUP script

The first thing to look at when reversing a NSIS installer is the setup script. NSIS scripts are a bunch of sections and assembly code written for the NSIS virtual machine. The NSIS VM architecture is relatively simple:

- Every instruction is encoded on 7 DWORDs: first DWORD is for the opcode (about 70 different opcodes) and the other 6 DWORD encode arguments
- Depending on the opcode, arguments can be either:
  - a register (up to 31 registers): `$0` .. `$9` , `$R0` .. `$R9` or one of 11 specific registers like `$EXEPATH` or `$CMDLINE` (some are read-only, so more like constants)
  - a global variable: `$var0` .. `$varN`
  - an integer, signed or unsigned. It can also be an offset into the code section for jump-like opcodes
  - a string, more precisely an index into the `Strings` section of the setup script
- Strings themselves can be somewhat complex to parse/interpret:
  - there are 3 NSIS versions: ansi, park (a mix between ansi and unicode) and unicode. Each version encodes strings differently. There is sadly no flag telling you which version is used.
  - strings can contains any of 4 special opcodes: `skip` , `shell` , `var` or `lang`
  - strings can include reference to system paths, variables or other strings, e.g. `"open {$INSTDIR}\rampage\goodie\noticeably.tif"`

Luckily for us, the full edition of Malcat features a NSIS disassembler / decompiler, so let us jump directly to the entry point of the script (**Ctrl+E**) and have a look at the `OnInit` method:

```
                  ;================================================================ ONINIT ================================================================
                  OnInit() {
Entries||000003ea0:    0500000021020000000000000000    Call          sub_4974()    ↓1                    ; →
Entries||000003ebc:    0D0000000D0000000000000000100    SetFlag       0xd, ""
Entries||000003ed8:    0B000000B10F0000010000000000    CreateDirectory "{$INSTDIR}", 0x1
Entries||000003ef4:    1400000090000005B50F00000000    ExtractFile   0x5000090, "d54hdan9c9ebsx", 0x0, 0xfb612500, 0x1d8068e, 0xffffffd9
Entries||000003f10:    1400000090000005C40F00000F4F    ExtractFile   0x5000090, "lognp", 0x34f0f, 0xfb612500, 0x1d8068e, 0xffffffd9
Entries||000003f2c:    0500000021020000000000000000    Call          sub_4974()    ↓1                    ; →
Entries||000003f48:    1400000091000005CA0F00009063    ExtractFile   0x5000091, "{$PLUGINDIR}\dwksh.dll", 0x36390, 0xffffffff, 0xffffffff, 0xffffffd9
Entries||000003f64:    0D0000000D0000000000000000100    SetFlag       0xd, ""
Entries||000003f80:    2C000000CA0F0000D80F00000000    RegisterDLL   "{$PLUGINDIR}\dwksh.dll", "sdvffryp", "", 0x0
Entries||000003f9c:    0D000000020000007F0400000000    SetFlag       0x2, 0x4f7b
Entries||000003fb8:    0D0000000B0000002A0700000000    SetFlag       0xb, 0x5226
Entries||000003fd4:    1F0000000C000000010000000000    Pop           $R2
Entries||000003ff0:    0F000000090000000B0000000000    GetFlag       $9, 0xb
Entries||00000400c:    1F000000DE0E0000000000000000    Push          "{$R3}"
Entries||000004028:    1F0000000000000000000000100    Swap          $0
Entries||000004044:    1F0000000D000000010000000000    Pop           $R3
Entries||000004060:    2E000000E10F0000011000001007    CopyFiles     "{$INSTDIR}\doohickey\guardian\slut.bmp", "{$INSTDIR}\divorces\extract.docx", 0x710
Entries||00000407c:    240000008E060000000000000000    SetCtlColors  "{$1}", 0x0
Entries||000004098:    1F000000381000000000000000000    Push          "{$R5}"
Entries||0000040b4:    0B0000003C100000000000000000    CreateDirectory "{$INSTDIR}\flannel\haggling\procrastinating", 0x0
Entries||0000040d0:    300000006110000006F1000000000    WriteINI      "ghtcbslisjklc", "ydilvvilstoe", "", "{$INSTDIR}\manic\tidings\dunk.zip"
Entries||0000040ec:    1F00000002000000010000000000    Pop           $2
Entries||000004108:    0600000097100000000000000000    UpdateText    "nnxkkxwxomg"
Entries||000004124:    19000000001000000A31000000000    Assign        $1, "gvrzioegyz", 0x0, 0x0
Entries||000004140:    2B0000000E00000000000000AE10    GetDLLVersion $R4, $0, 0x10ae
Entries||00000415c:    3500000007000000002000080CC10    RegEnumKey    $7, 0x80000002, "dytsrudxmci", "4432"
Entries||000004178:    0B000000DD100000000000000000    CreateDirectory "{$INSTDIR}\marmalade\implausible\admonish", 0x0
Entries||000004194:    0200000000000000000000000000    Jmp           loc_0
                  }
```

Figure 3: NSIS setup script entry point

We can see that the script does the following:

- extract the first buffer (offset `header+0` in archive) to a file named `d54hdan9c9ebsx`
- extract the second buffer (offset `header+0x34f0f` in archive) to a file named `lognp`
- extract the third buffer (offset `header+0x36390` in archive) to `${PLUGINDIR}\dwksh.dll`, wherever that could be
- call dwksh.dll's exported method `sdvffryp` without any argument

The rest of the method seems like junk code, judging by the strings which are either random letters or picked out of dictionary. Quickly inspecting the first two files tells us that both are encrypted and/or compressed, so no quick-win there. We have to dig into the dll.

Stage 1: dwksh.dll

dwksh.dll is a 294KB 32-bits DLL of sha256 `be00a655cdf4e4706bf1bbf3659d698f967cad66acdf7cda0d12f16dc0cfda3e` (VT). It contains several obfuscated methods. But we reversed the setup script and know what to look for: the method `sdvffryp`. This methods starts by reading a local file named `lognp`:

```
sdvffryp() {
    55              push    ebp
    8BEC            mov     ebp, esp
    B8A4180000      mov     eax, 0x18A4
    E8266E0200      call    sub_10047e60()     ↓1                ; →
    56              push    esi
    6A6C            push    0x6C = 'l'
    58              pop     eax
    6A6F            push    0x6F = 'o'
    668945F4        mov     [ebp-0x0C], ax
    58              pop     eax
    6A67            push    0x67 = 'g'
    668945F6        mov     [ebp-0x0A], ax
    58              pop     eax
    6A6E            push    0x6E = 'n'
    668945F8        mov     [ebp-0x08], ax
    58              pop     eax
    6A70            push    0x70 = 'p'
    668945FA        mov     [ebp-0x06], ax
    58              pop     eax
    668945FC        mov     [ebp-0x04], ax
    33C0            xor     eax, eax
    668945FE        mov     [ebp-0x02], ax
    8D85DCFBFFFF    lea     eax, [ebp-0x424]
    50              push    eax
    6803010000      push    0x103
    FF1518800410    call    [kernel32.GetTempPathW]
```

Figure 4: lognp file is accessed

It then seems to decrypt it in memory into an executable buffer before jumping at the beginning of the buffer (see the `call eax` below?). The file `lognp` is relatively small (5KB), it definitely looks like a shellcode.

```
push    esi
lea     ecx, [ebp-0x10]
mov     esi, 0x147D
push    ecx
push    esi
lea     ecx, [ebp-0x18A4]
push    ecx
push    eax
call    [kernel32.ReadFile]
mov     ecx, 0x00
```

```
mov     al, [ebp+ecx*1-0x18A4]
cmp     ecx, 0x147D
jz      .2
```

```
sub     al, 0x4E
inc     al
sub     al, 0xD5
inc     al
inc     al
dec     al
add     al, 0x1E
inc     al
dec     al
xor     al, 0xF2
sub     al, 0x6A
inc     al
add     al, 0x02
sub     al, 0x7E
add     al, 0xF5
inc     al
xor     al, 0x28
dec     al
mov     [ebp+ecx*1-0x18A4], al
add     ecx, 0x01
jmp     .1
```

```
mov     al, 0x00
mov     ecx, 0x00
push    "VirtualProtect"
push    "kernel32.dll"
call    [kernel32.LoadLibraryW]
push    eax
call    [kernel32.GetProcAddress]
lea     ecx, [ebp-0x14]
push    ecx
push    0x40 = '@'
push    esi
lea     ecx, [ebp-0x18A4]
push    ecx
call    eax
lea     eax, [ebp-0x18A4]
call    eax
xor     eax, eax
pop     esi
mov     esp, ebp
pop     ebp
ret
```

Figure 5: decryption loop in method sdvffryp

The decryption is pretty straightforward according to the sleigh decompiler. Every byte seem to be decrypted using the following formula:

```
byte[i] = ((((byte[i] - 3) ^ 0xf2) + 0x11) ^ 0x28) - 1
```

Decrypting the `lognp` file should be a piece of cake. Just open the file inside Malcat, select every byte (**Ctrl+A**) and open the transform dialog (**Ctrl+T**). There you can chose the `custom` arithmetic transform which allows you to transform sequence of bytes/words/dwords using a custom python expression. Just paste the equation above, replacing `byte[i]` by `value` and *voila*, you've just decrypted the second stage.

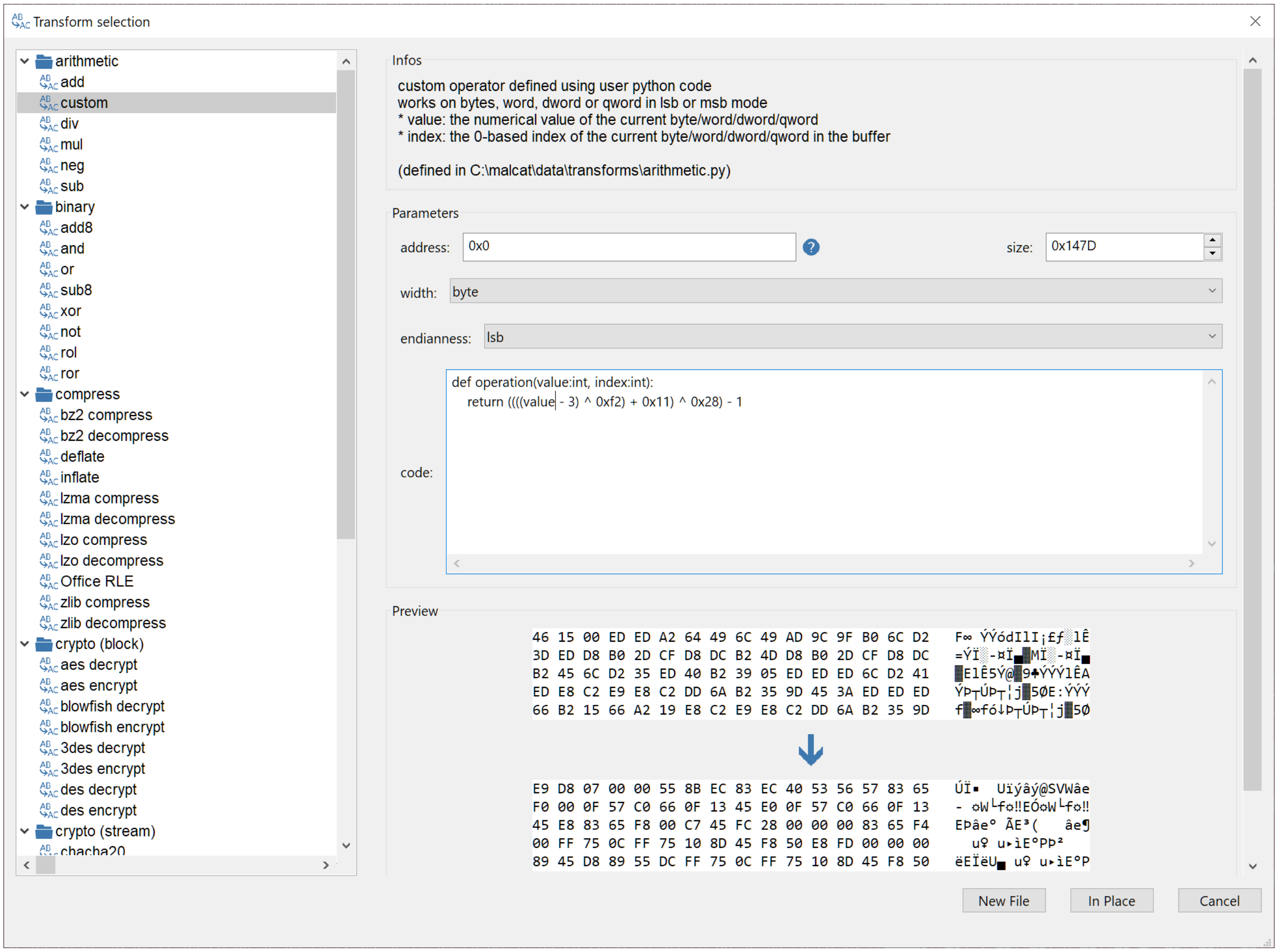

Figure 6: decrypting the shellcode

For the lazy readers, you can download the decrypted `lognp` file here (password: infected).

Stage2: obfuscated shellcode

Analyzing the shellcode

The `lognp` file, once decrypted, does not appear to be in any known file format. But the first byte is `E9`, which is a jump in the x86 architecture and is very typical for shellcode prologs. So before starting the analysis, we will have to tell Malcat two things:

- the architecture used: x86 in our case. This can be set using the dropdown menu in the status bar
- the entry point of the shellcode, which is at address 0 in our case. We just have to define a new function start at this address using the context menu in disassembly mode (**F3**)

After this, Malcat is smart enough to recover most of the shellcode's CFG using its usual set of analyses. Following the control flow, we quickly arrive in the function `sub_7dd` which contains interesting patterns:

- the string `d54hdan9c9ebsx` (one of the NSIS archive's file names) is pushed on the stack at address `0x8eb`
- something like a `CreateFileA` API call is performed at address `0x989` (the constant `0x80000000` is most likely for `GENERIC_READ`). If we wanted to be sure, we would have to emulate the API lookup function at address `0x776`, but it looks like safe assumption.
- soon after, the function `sub_a01` gets called. Decompiling this functions reveals something similar to a decryption loop

The whole process can be retraced in the animated GIF below:

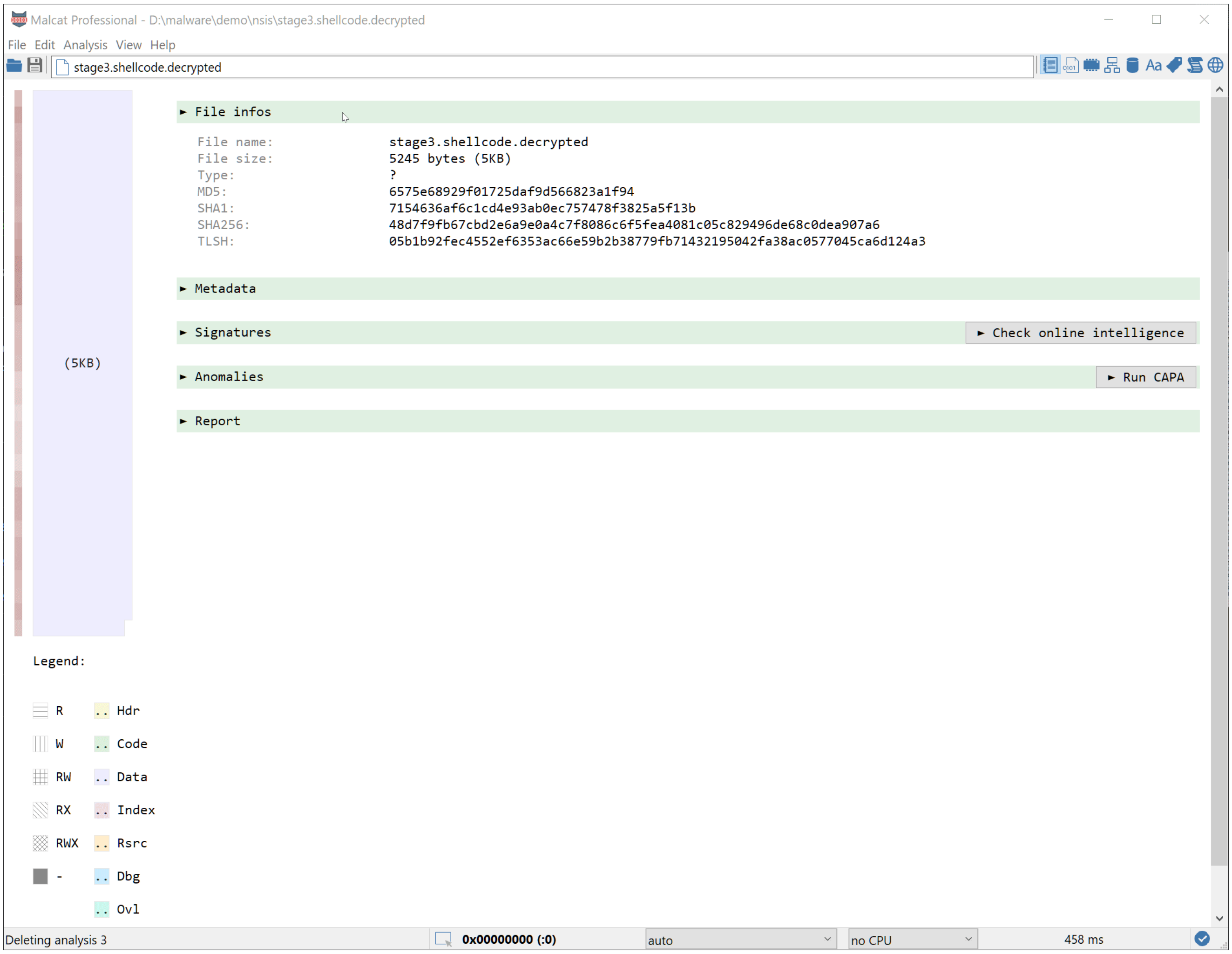

Figure 7: locating the decryption function inside the shellcode

The code of the decryption function is given below. It is obviously obfuscated, and sadly it would not be immediate to reimplement it in python in Malcat. So we will have to find an alternative. Since the decryption function prototype is very simple (it just needs a pointer to the buffer and the buffer size) and is without side effects, why not give emulation a go?

```
BYTE* sub_a01(BYTE* buffer, uint4 size) {
    uint1 uVar1;
    char cVar2;
    uint1 uVar3;
    uint4 i;

    i = sub_0;
    while (i < size) {
        uVar3 = i;
        uVar1 = -uVar3 - ((*(buffer + i) >> 1 | buffer[i] << 7) - 0x40 ^ 0xf2);
        uVar1 = -uVar3 - (uVar1 >> 7 | uVar1 * '\x02');
        uVar1 = -uVar3 - (((uVar1 >> 3 | uVar1 * ' ') ^ uVar3) - uVar3 ^ 0x9c) ^
0xd6;
        cVar2 = ~((uVar1 >> 7 | uVar1 << 1) + 0x34 ^ 0x87) - 0x10;
        uVar1 = ~(((-cVar2 >> sub_5 | cVar2 * -8) ^ 0x1d) + 0xac) ^ 0x5e;
        uVar1 = ~-(((0x99 - ((uVar1 >> 2 | uVar1 << 6) + 0x49) ^ 0xa0) + 0x30 ^ 0x34)
+ uVar3);
        uVar1 = (-uVar1 >> 6 | uVar1 * -4) - uVar3 ^ uVar3;
        uVar1 = (-uVar1 >> 2 | uVar1 * -0x40) + 0x93;
        uVar1 = (-((((uVar1 >> sub_5 | uVar1 * '\b') - 0x2e ^ 7) + 0xd ^ 0x96) +
0x31) ^ 0x73) + uVar3;
        uVar1 = -uVar3 - ((uVar1 >> 2 | uVar1 * '@') + 0x61) ^ uVar3;
        uVar1 = ~((uVar1 >> 3 | uVar1 << sub_5) ^ uVar3);
        uVar1 = (uVar1 >> 7 | uVar1 << 1) + uVar3 ^ 0x2e;
        uVar1 = ~(~((uVar3 - (~(~(-(0xbc - ((uVar1 >> 6 | uVar1 << 2) - uVar3) ^
0x1e) ^ 0xc5) ^ 0x46) ^ 0xc1) ^ 0x4c) +
                   uVar3) ^ 0x4d) + 0x4c ^ uVar3;
        uVar3 = 0x2d - (-((uVar1 >> 3 | uVar1 << sub_5) + uVar3) ^ 0x43);
        buffer[i] = (uVar3 >> 7 | uVar3 * '\x02') + 0x15;
        i = i + 1;
    }
    return buffer;
}
```

Emulating the decryption function

To emulate shellcodes, Malcat comes bundled with a script named `speakeay_shellcode.py` which emulates shellcodes using the Speakeasy emulator. Note that Speakeasy is not bundled with Malcat, you will have to install the python package yourself (and if you are running Malcat under Windows, be sure to check `Use system python interpreter` in the options).

Patching lognp

Before emulating anything, we need to solve a problem: the data to decrypt (`d54hdan9c9ebsx`) is not embedded in the `lognp` shellcode, it is read from the filesystem using `CreateFileA`. So emulation is likely to fail. How are we going to solve this issue?

There is the clean way: we could hook the CreateFileA/ReadFile APIs in speakeasy and intercept the call to give back the content of the file `d54hdan9c9ebsx`.

But there is also the dirty way: we could patch the decrypted `lognp` shellcode in order to embed the content of `d54hdan9c9ebsx` in the shellcode space and patch the shellcode entry point to perform a call to the decryption function with the right parameters. Of course we will chose the *dirty* way. It is not only way faster, it is also more fun.

Here is how to proceed:

1. First open a copy of the decrypted `lognp` shellcode in Malcat with extra space at the end of the file (**File > Open Copy of File**). The file `d54hdan9c9ebsx` is 216843 bytes big, we'll append 300KB just to be sure.
2. Copy the content of the file `d54hdan9c9ebsx` in the clipboard: in a second Malcat instance, open `d54hdan9c9ebsx` and then hit *Ctrl+A* followed by *Ctrl+C*
3. Paste the copied data after the shellcode in the first Malcat instance, let's say at address `0x2000` to make it easy to remember
4. Enter disassembly view (*F3*) and go to the shellcode's entry point at address 0

Malcat does not (yet) support assembling your own instruction, so we will need to manually edit the machine code. Click on any hexa byte in disasm mode and enter edit mode (**Insert** key). We need to assemble the following code:

1. Push the second parameter which is the size of the buffer to decrypt (216843 = 0x34F0B). `push uint32` is assembled using `0x68` + `LSB-encoded uint32` in x86: `68 0B 4F 03 00`
2. Push the first parameter which is the address of the buffer to decrypt (`0x2000`): `68 00 20 00 00`
3. Call to the decryption function. The call opcode is `0xE8` + signed displacement starting from the end of the call opcode. The end of our call opcode is at address `0x000F` , we want to jump to `0x0A01` , so `0x0A01` - `0x000F` = `0x09f2` . We need to assemble `E8 F2 09 00 00` .

> You can use Malcat's calculator to perform quick computation while analysing a binary, just hit **Ctrl+Space**. Internally, it uses the python interpreter, so use python syntax.

At the end, the patched shellcode should look like in the picture below. For the lazy readers, you can download the patched `lognp` file here (password: infected).

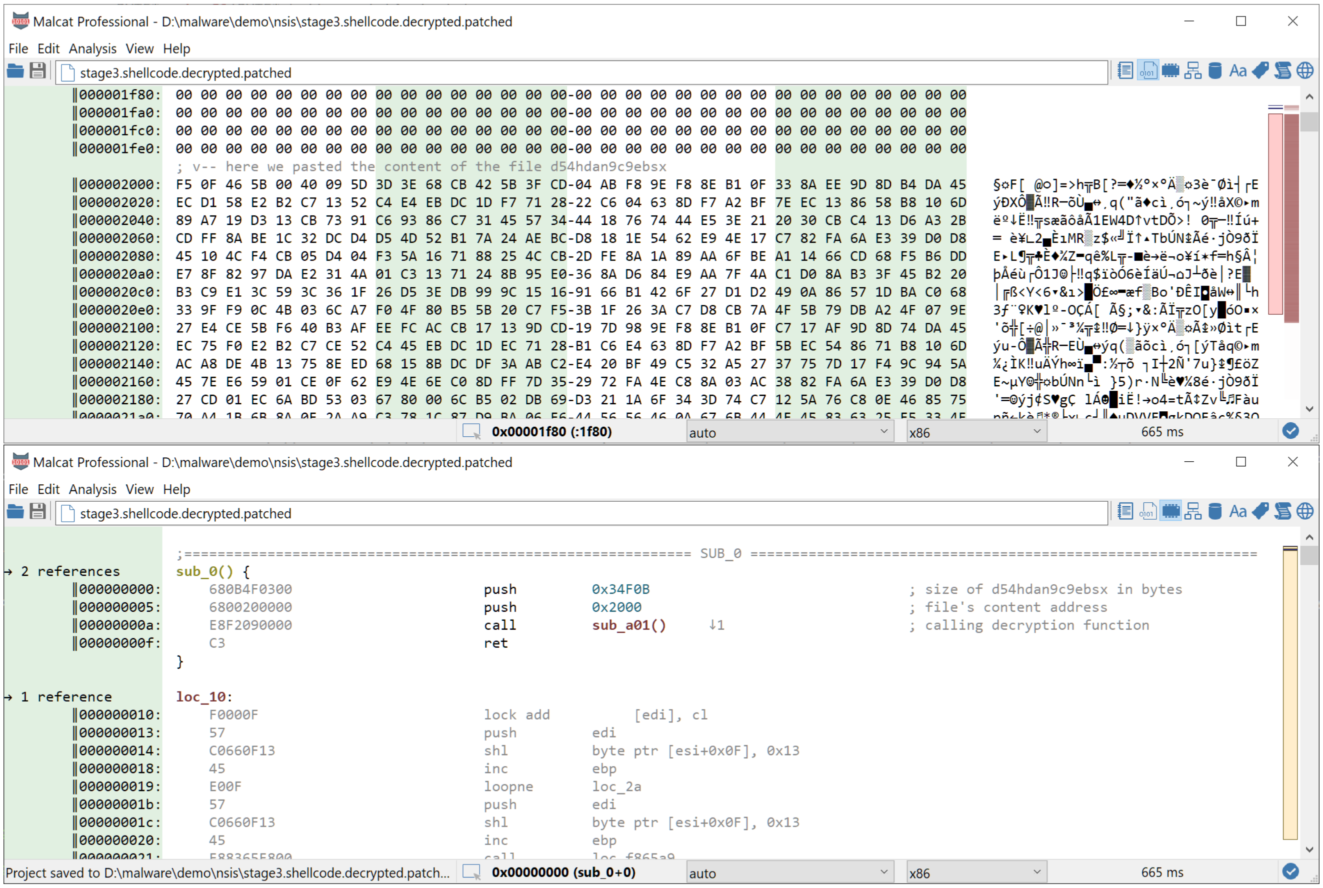

Figure 8: patching the shellcode

Running speakeasy

Now the only thing we have to do is to let speakeasy do its magic:

- let us define the entry point: right-click at address 0 and chose **Force function start** in the context menu
- run the script `speakeasy_shellcode.py` (**Ctrl+U** to run user scripts)

... and voila, Malcat should open the result in a new file. A PE file has been detected by Malcat's file format parser at address 0x2000, perfect! Just double-click the PE file under "Carved Files" to open it.

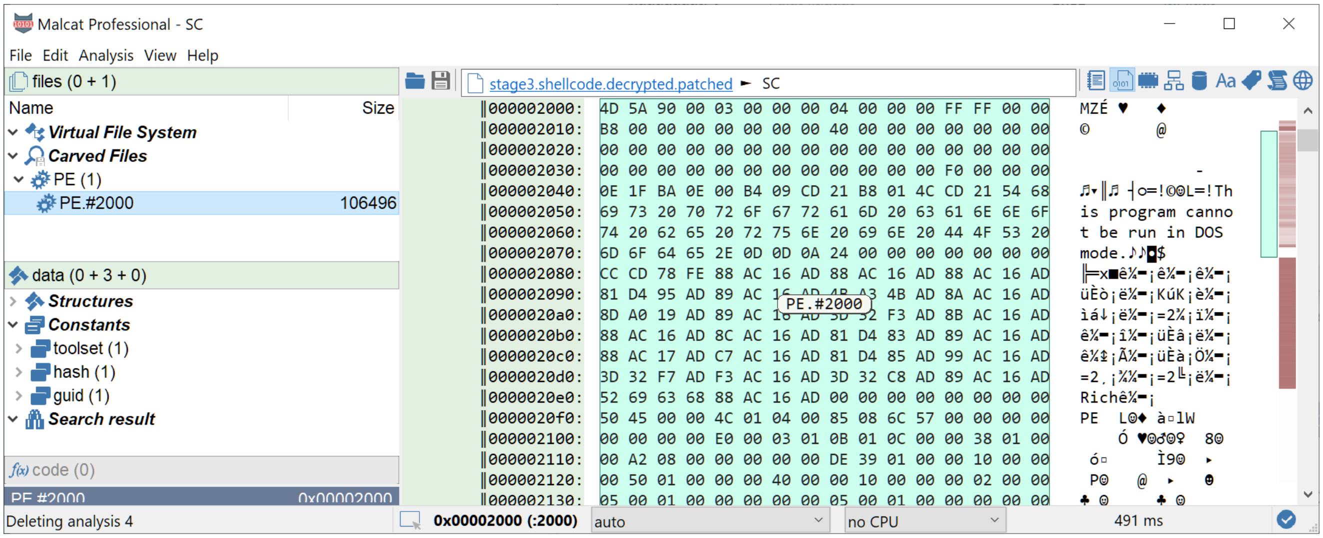

Figure 9: decrypted d54hdan9c9ebsx

Stage 3: Lokibot and config extraction

The last (and final) stage we get is a PE file of sha `02dee91a81652e5234414e452622307fd61e7988e82bec43665f699f805c3151` (VT). Judging by the low entropy and the visible strings, the file does not seem to be obfuscated, good news. So which kind of malware do we face? Malcat's Yara rules already spotted one of main malware intent: stealing credentials, as we can see in the screenshot below:

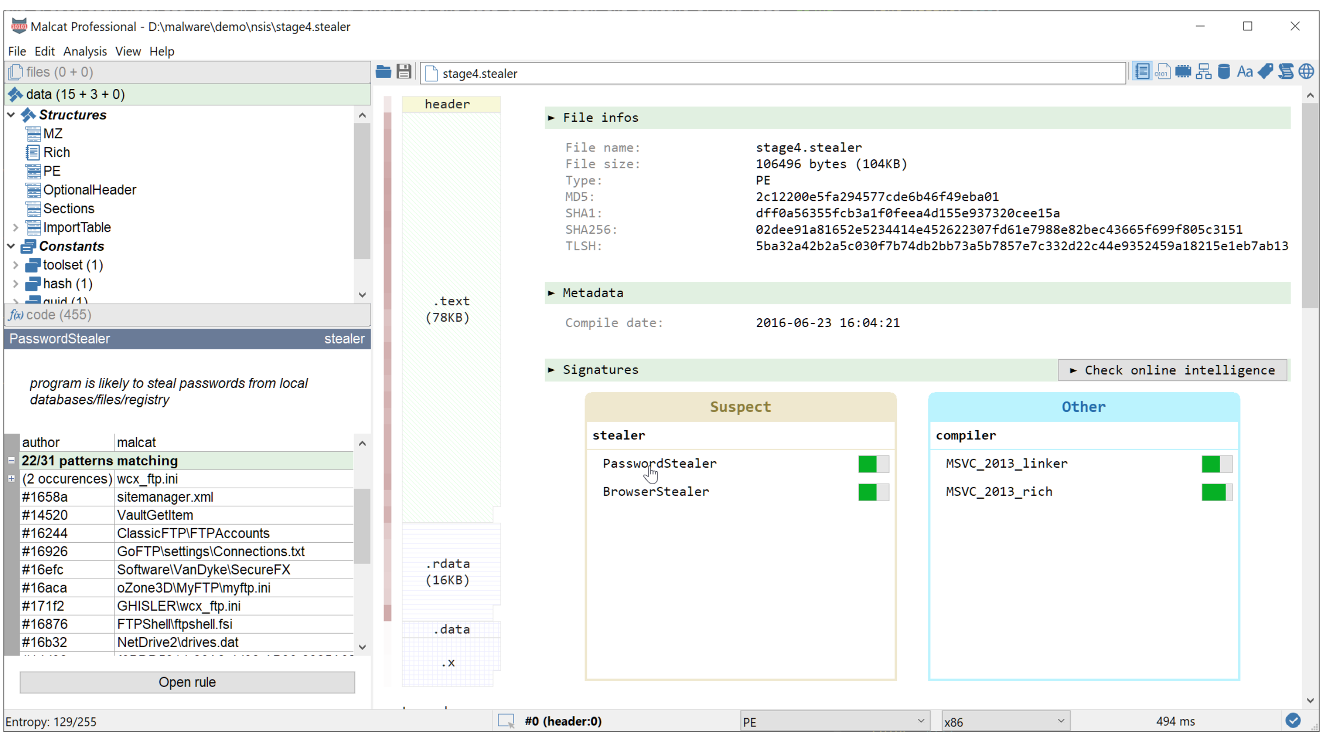

Figure 10: file summary, displaying matching Yara rules

If we want to be more precise, we can use Malcat's online intelligence view (**Ctrl+I**, only for paid versions). Normally I would avoid using Virustotal to identify a malware family (because of packer reuse among threat actors). But here we are dealing with the plain text final malware, so we should get at least *some* valid labels. In our case, it seems to be Lokibot, a simple password stealer:

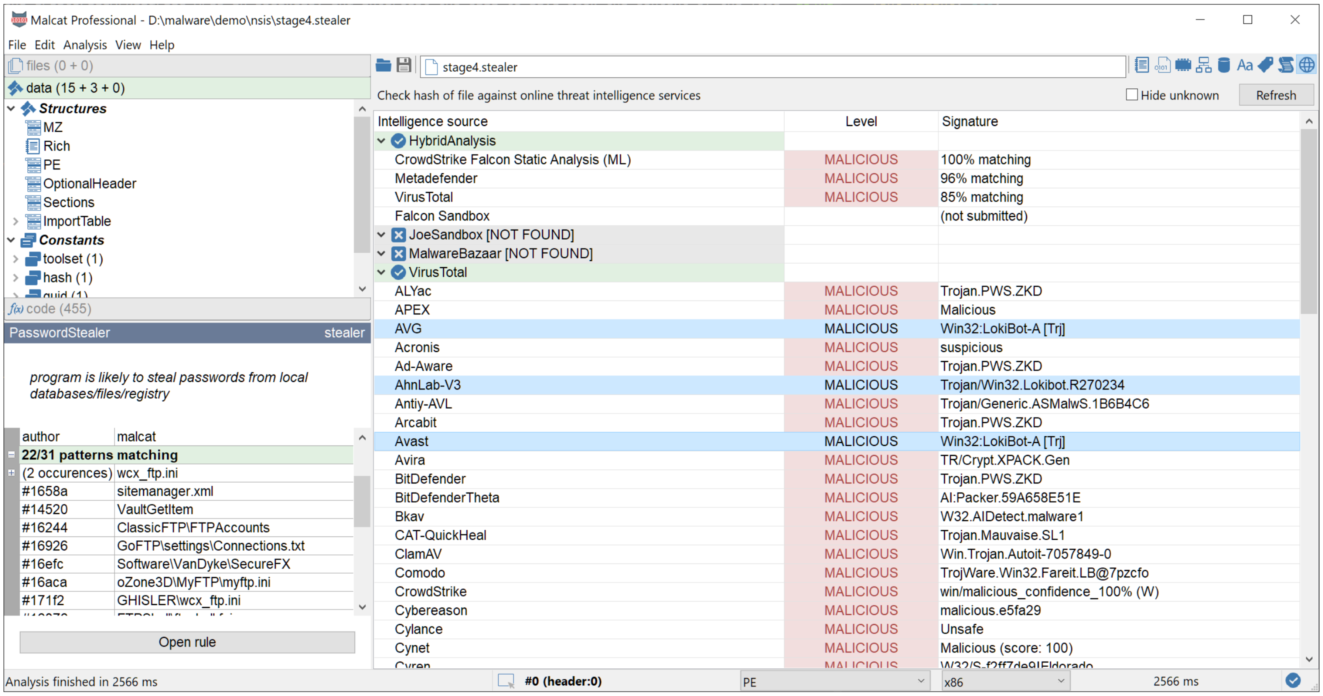

Figure 11: querying online intelligence

Can we go further? The last section of the PE file is weirdly named `".x"`. It contains a single method at address `0x4a0000` and a few bytes of referenced data at address `0x4a0074`. Looking at the function, it seems to decode the data using a `XOR` opcode, with the key `0xDDDDFFFF`. But actually, only the first byte of the key is used (`0xFF`), so it is strictly equivalent to performing a simple `NOT` on the data. Great, let us decrypt these few bytes using Malcat's transform:

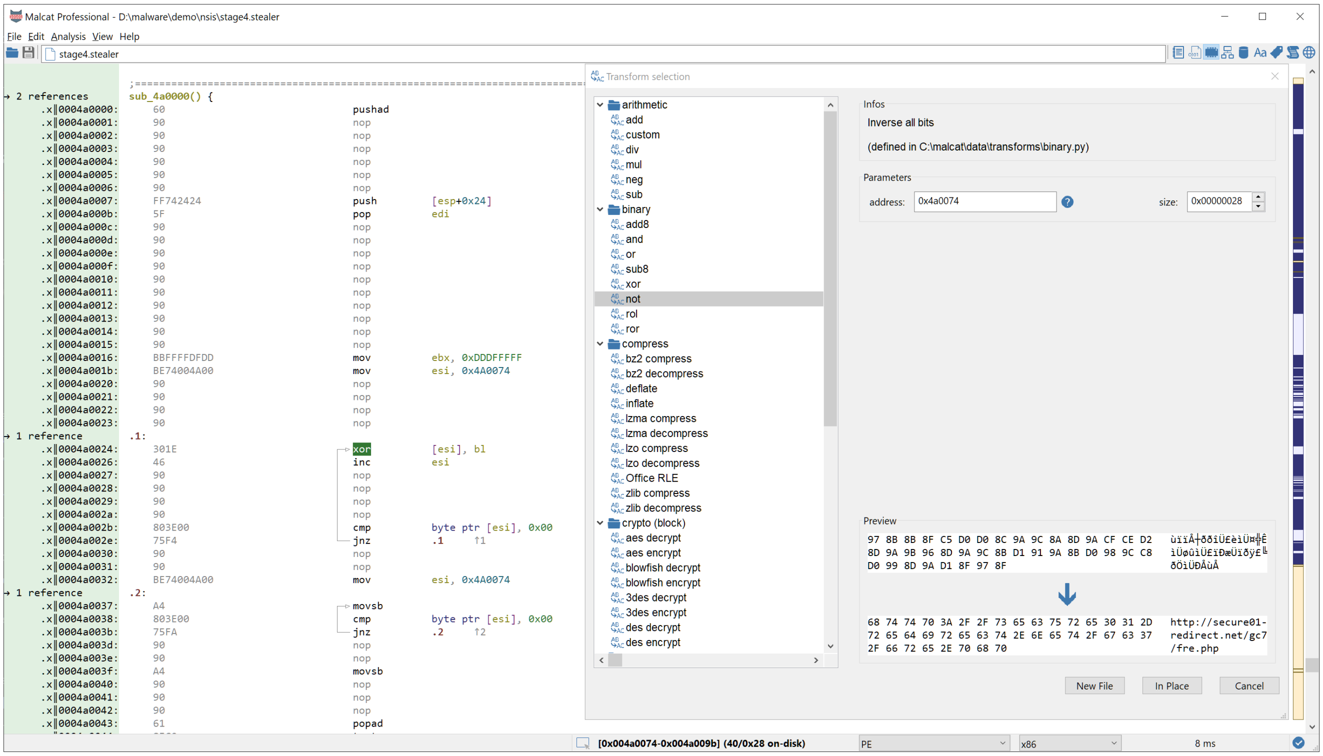

Figure 12: decrypting the data buffer in the .x section

Great, we got the address of the command and control server for this sample. This was a pretty easy catch ^^

Conclusion

NSIS installers have been abused by malware authors for some years now. While the NSIS VM instruction set is relatively limited, DLL plugins allow malicious actors to extend installer capabilities and obfuscate malware. In this example, two layers of shellcodes were used by the NSIS installer in order to deliver its final payload: a LokiBot password stealer.

Instead of running everything in a VM, we made great use of Malcat's NSIS disassembler, Malcat's transforms and speakeasy emulator in order to quickly unpack these two layers statically.

We hope you enjoyed this new quick-and-dirty malware unpacking sessions. Future blog posts will be more focused toward beginners as we will introduce a few of Malcat's features as in-depth tutorials.