Writing a Qakbot 5.0 config extractor with Malcat

By Malcat EI

Archived: 2026-04-05 13:22:27 UTC

Sample:
: 73472cfc52f2732b933e385ef80b4541191c45c995ce5c42844484c33c9867a3.msi (Bazaar, VT)

Infection chain:
: MSI installer -> Backdoored DLL -> PE loader -> Qakbot

Tools used:
: Malcat

Difficulty:
: Intermediate

Introduction

Qakbot has been studied a lot over the last 15 years, and plays a big role in the malware landscape. After a successful takedown that took place in August 2023, it got a bit of attention lately as a new variant has been spotted around December 2023.

But after raising from the dead, the RAT also switched to a new version: 5.0. Sadly the existing Qakbot configuration extractors stopped working (as far as I know), suggesting that the malware code underwent non-trivial changes. That is relatively annoying: configuration extractors are really useful for botnet tracking and incident response. But instead of complaining, let us fire up Malcat and see if we can write a configuration extractor ourselves!

First stage: MSI installer

Thanks to Malware Bazaar, it was rather easy to find a recent Qakbot sample. This one happens to be a MSI installer. MSI installers are often abused by malware authors to package their malicious programs. So let us load the file in Malcat and see what we got. A first look in the summary view tells us that we are facing an "Acrobat" installer. Sure.

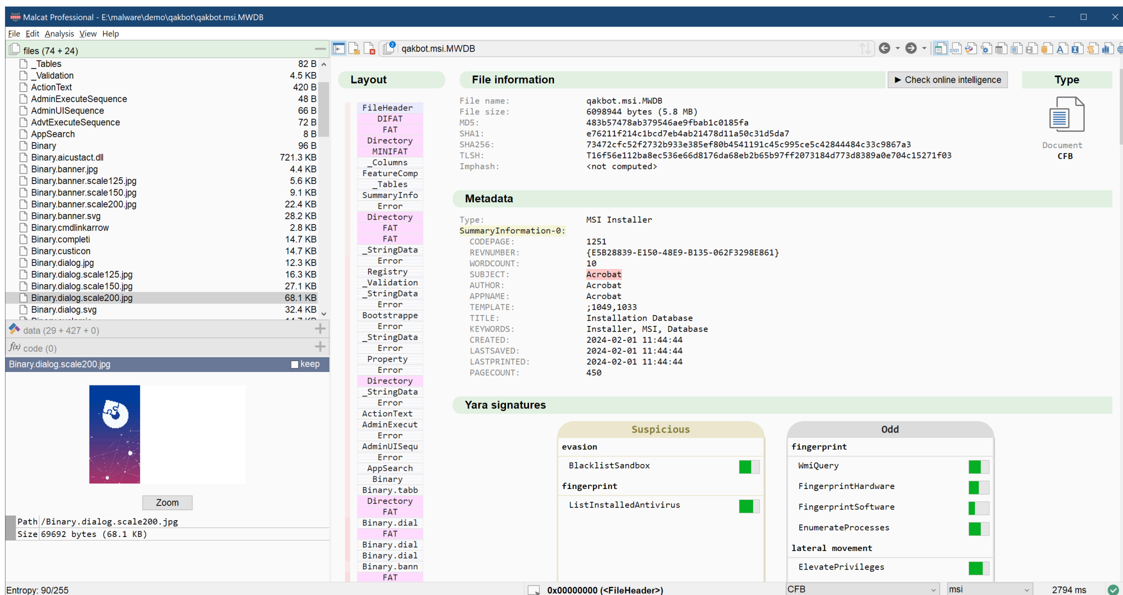

Figure 1: The installer

The first thing to look at when analysing a MSI installer is the `CustomAction` table, which somewhat drives the installation process. Luckily, Malcat full & pro can display the content of all MSI tables in the decompiler view. Just press **F4** and scroll down to the `CustomAction` table (tables are sorted alphabetically). Entries of type `LaunchFile` are particularly interesting, and there is indeed one running a program named `viewer.exe` with a pretty suspicious command line:

```
{
    "Action": "LaunchFile",
    "Type": 2,
    "Source": "viewer.exe",
    "Target": "/HideWindow rundll32 [APPDIR]\\MicrosoftOffice15\\ClientX64\\[ProductName].dll,CfGetPlatformInfo'
    "ExtendedType": null
}
```

We will first have a look at this `viewer.exe`. In my experience, there are two types of files in a MSI installer:

- Files that are just needed during the installation: pictures, plugins, tools etc. These files are stored inside the `Binary` database. Malcat will list them as `Binary.<filename>` in the **Virtual File System** tab.
- Files permanently installed to disk. These are stored inside a CAB archive, like the `disk1.cab` file in this installer.

Our file `viewer.exe` seems to be of the first type, and we just have to double-click `Binary.viewer.exe` in the **Virtual File System** tab to open it. A quick threat intelligence hash lookup (**Ctrl+I** or **Check intelligence** button from the summary view) suggests us that the file might be a simple third-party launcher:

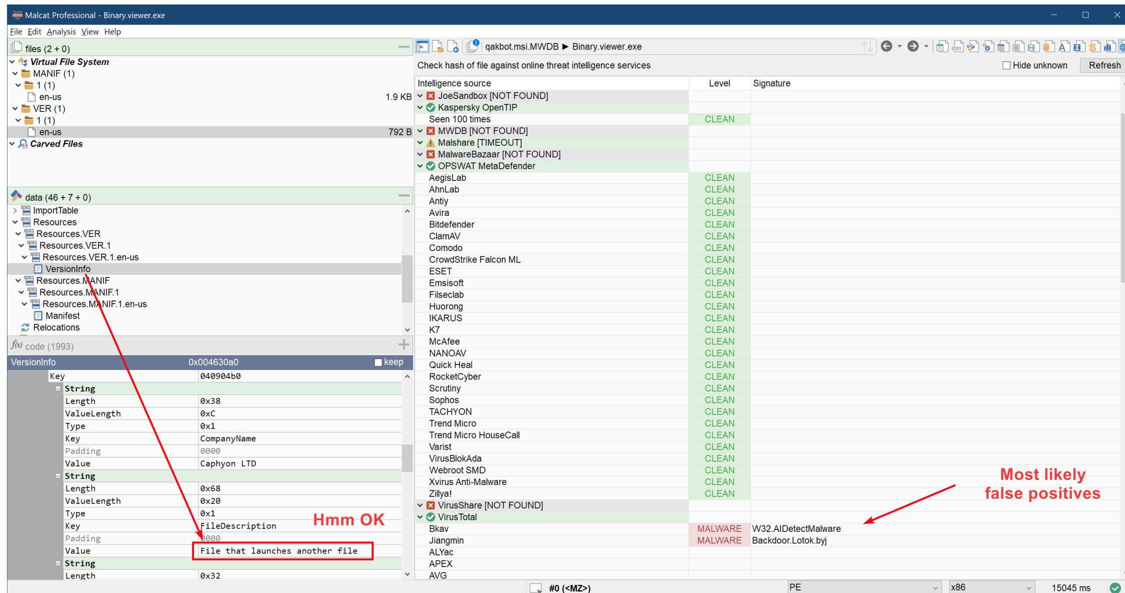

Figure 2: The file viewer.exe

Next in line of suspects is the DLL referenced in the `Target` property. We don't have the name of the DLL, but luckily for us there is a single DLL file named `dll_1` in the file `disk1.cab`. To open it, just double-click on `disk1.cab` and then on `dll_1`. We are now facing the second stage of the infection.

Second stage: Antimalw.dll

The file `dll_1` is a 922KB PE DLL of sha256 `a59707803f3d94ed9cb429929c832e9b74ce56071a1c2086949b389539788d8a` (Virusshare, VT) named either `antimalw.dll` (version infos) or `antimalware_provider64.dll` (export name). The file immediately strikes us as suspicious:

- It claims to be Bitdefender's AMSI provider, that is the script scannning component of the Bitdefender antivirus. `antimalw.dll` contains parts of Bitdefender's original DLL, but clearly isn't.
- Its data directory suggests that it is signed with a certificate, but the location of the certificate has been overwritten by the `.rsrc` section
- It has one large high-entropy resource named `ЬГнЦИРИ`
- Its entry-point function is empty
- It has a single exported function `CfGetPlatformInfo` which seems obfuscated

It looks that the malware author took Bitdefender's `antimalware_provider64.dll` and backdoored/overwrote it with malicious code.

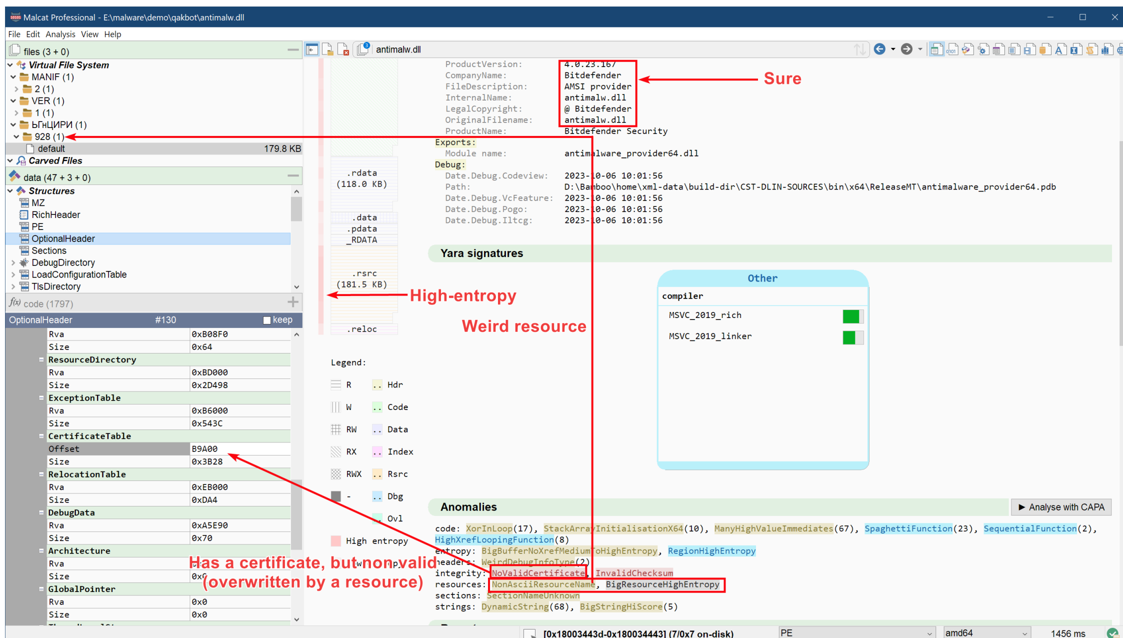

Figure 3: A suspicious DLL

Now that we have verified that the file is malicious, back to business. The first step I take when facing a packed malware is a process I call [Where is the poop, Robin](). See, there is no magic: malware *have to store their payload somewhere* (unless they're downloaders of course). So instead of diving blindly into the code or submitting the binary to a slow sandbox, it is often best to first *locate the encrypted payload*. Finding the hidden payload either allows you to decrypt it immediately or, worst case scenario, will give you useful pointers to start your reverse engineering.

The large high-entropy resource `ЬГнЦИРИ` seems like a good candidate to start our search. Scrolling through its bytes in the hexadecimal view, we can see a *repeating pattern* near the end of the file. This usually suggests some kind of rotating key encryption mechanism. Since there is a huge chance that the end of the file are zeroes, and since we know that malware authors *love* their XOR encryption, we will simply try to un-xor it with the key `"HU03!Mm!?qYHCTnaEX<\0"` (note the ending null byte). Incidentally, this string appears as a stack string in the exported function `CfGetPlatformInfo`, which is encouraging:

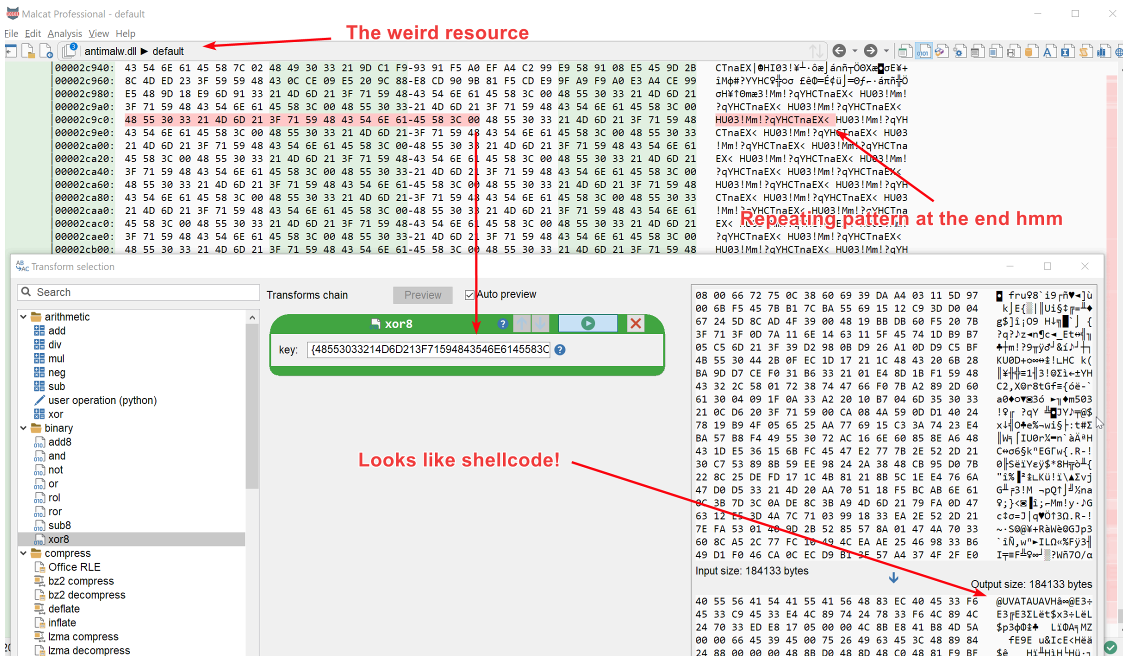

Figure 4: Decrypting the resource

And indeed, we have successfully decrypted the resource. Long live XOR encryption!

Stages 3: PE loader

We are now facing what looks like a 180KB x64 shellcode (sha256 `8c7401218e6da9533d4e97849ad6c528b231c1b9cdcf43d1788757c3862dc2d4`). Now there are two ways to go forth. The obvious one is to emulate the shellcode, which can be done following the steps below:

1. Force the architecture to x64
2. Select first byte of the shellcode and define a new function there
3. Try your luck with one of Malcat's emulator script, for instance running the script **emulation/Speakeasy (shellcode)**

On the other hand, Malcat did carve a 170KB plain text PE file out of the 180 KB shellcode. So let us take the easy way and just grab the next stage by double-clicking the carved PE file:

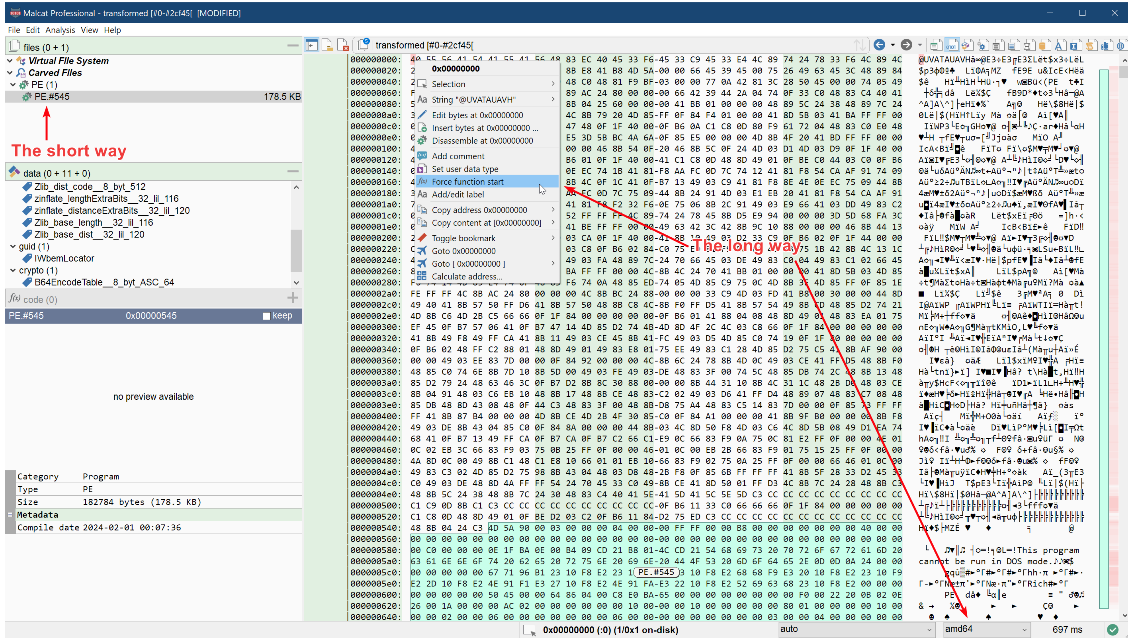

Figure 5: The shellcode and it embedded PE file

Stage 4: the Qakbot DLL

The next stage is a 170KB PE dll of sha256 `af6a9b7e7aefeb903c76417ed2b8399b73657440ad5f8b48a25cfe5e97ff868f` (Virusshare, VT) named `cldapi.dll`. We are facing the final stage of the infection chain: a *Qakbot* malware compiled the 2024-01-29, so most likely one of the new 5.0 version!

How can we be sure it's the final malware? Usually I tend to confirm with Malpedia's Yara rules, but sadly their Yara rule don't seem to cover the new Qakbot version. But if we compare our `cldapi.dll` sample against a Qakbot version from March 2023 (e.g. this one), we can see that even if some strings were changed or got encrypted, most are still there:

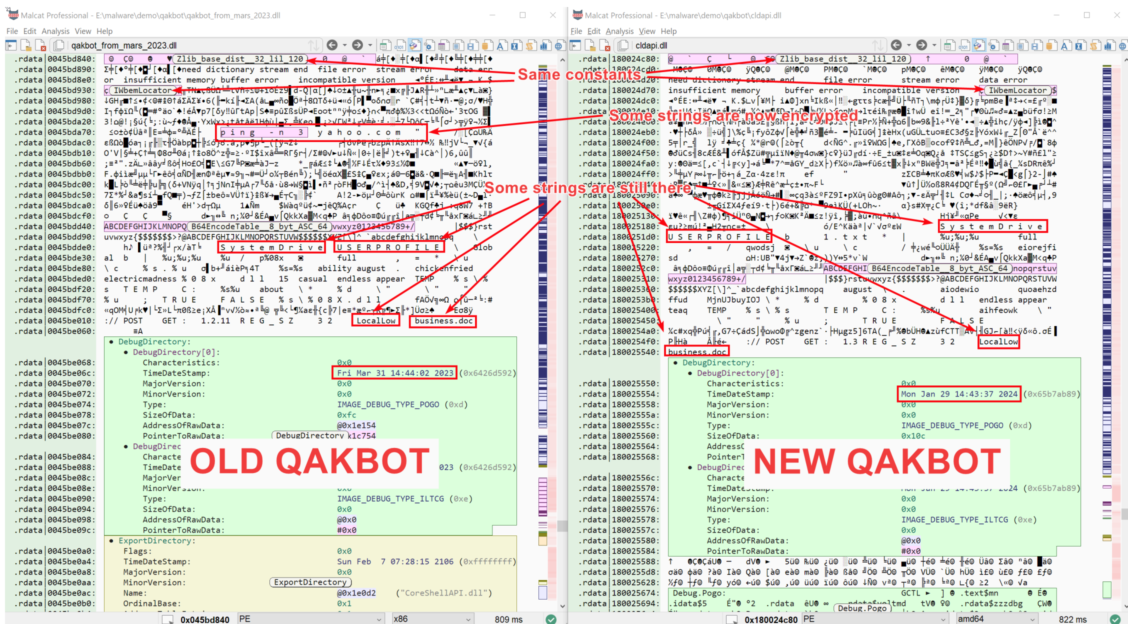

Figure 6: Strings comparison against a Qakbot sample from march 2023

Beside the Qakbot attribution, we can see that the DLL is slightly obfuscated:

- API addresses are resolved dynamically by hash at runtime (hashes are encrypted)
- Most strings are encrypted
- There are a few junk code islands here and there

While API obfuscation is not a big deal in our case, the string encryption might be problematic if we want to write a configuration extractor. This will be our first task: *locate and decrypt Qakbot's strings*.

Decrypting strings

Locating the first encrypted string array

While Qakbot is not a huge malware, reversing more than 120KB of code will always be tedious. And since we are looking for something rather precise, an encrypted data blob, we will again focus on the data instead instead of diving into the code. More precisely, we will try to find all data buffers in the any data section which are:

- relatively large, let's say more than 64 bytes
- have a high entropy
- have incoming code references

To ease your search, make sure that you have enabled the [incoming references highlighting](#).

By chance Malcat already identifies a few known constant arrays such as precomputed tables used by the embedded Zlib library, which saves us some time as these buffers are not interesting to us. Starting from address `0x180028150`, we can see a few candidates. The first three buffers look rather promising (we have [named and colored](#) them for the sake of clarity):

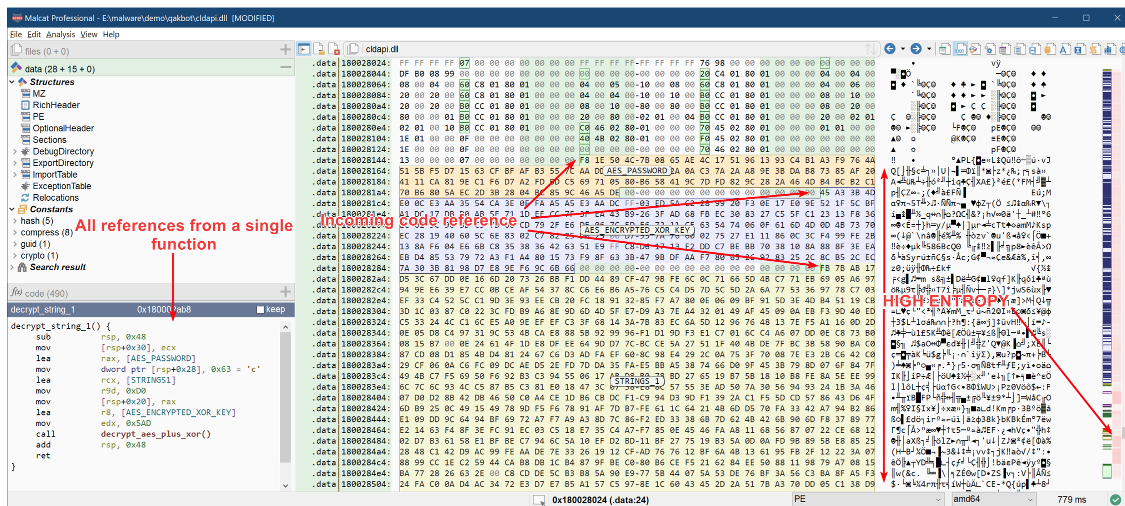

Figure 7: Candidates for the encrypted buffer award

These three buffers are all referenced by the same function `sub_180002ab8` that we have renamed to `decrypt_string_1`. This function looks like your typical string decryption function: it has numerous incoming references, as we can see below, each call with a different hardcoded parameter. There is a big chance that this parameter is a string index:

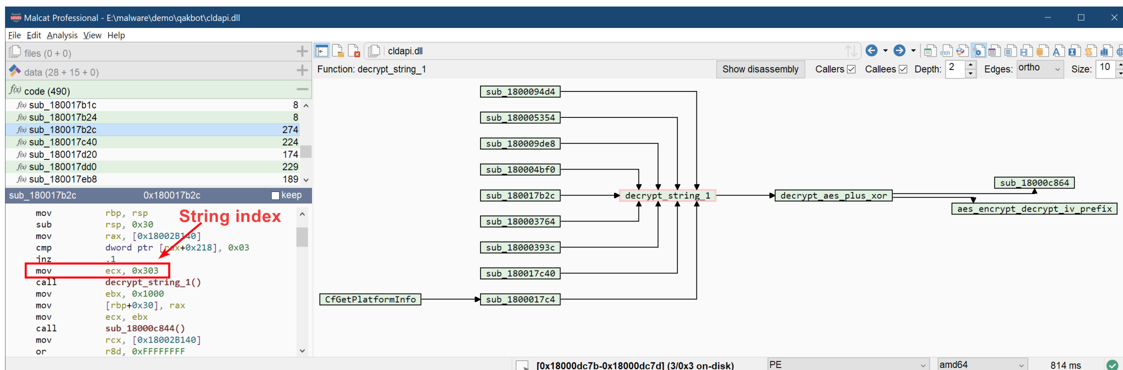

Figure 8: The first string decryption function in context

The function `decrypt_string_1` is rather simple: it calls an auxiliary function that we have named `decrypt_aes_plus_xor` with our encrypted three buffers as parameter. Its decompiled code (**F4**) is presented below:

```
 void decrypt_string_1(xunknown4 string_index)
{
    decrypt_aes_plus_xor(ENCRYPTED_STRINGS_1, 0x5ad, AES_ENCRYPTED_XOR_KEY, 0xd0, AES_PASSWORD, 0x63, string_in
    return;
}
```

The value of each variable is given below:

Name	Address	Size in bytes	Description
decrypt_strings_1	0x180002ab8	0x3f	Decryption *function* for the first encrypted strings array
STRINGS_1	0x1800282a0	0x5ad	First *encrypted strings array*
AES_ENCRYPTED_XOR_KEY	0x1800281c0	0xd0	The *XOR key* used to decrypt the string array, but AES256-CBC encrypted
AES_PASSWORD	0x180028150	0x63	The *password* used to derive the AES256 key for AES_ENCRYPTED_XOR_KEY
decrypt_aes_plus_xor	0x18000dc2c	0x1de	The *function* that decrypts the string array and selects the string
aes_encrypt_decrypt_iv_prefix	0x180011504	0x3f7	A *function* called by decrypt_aes_plus_xor that decrypts or encrypts an arbitrary data buffer using AES256 in CBC mode

Decrypting the strings

To get what the function `decrypt_aes_plus_xor` does, a little reverse engineering is needed. As the code is relatively short you can do it statically, although you will face some issues since APIs are resolved dynamically. Tracing the function using a debugger is the smarter choice there. Anyway, at the end it is relatively easy, and the string decryption routine looks something like that:

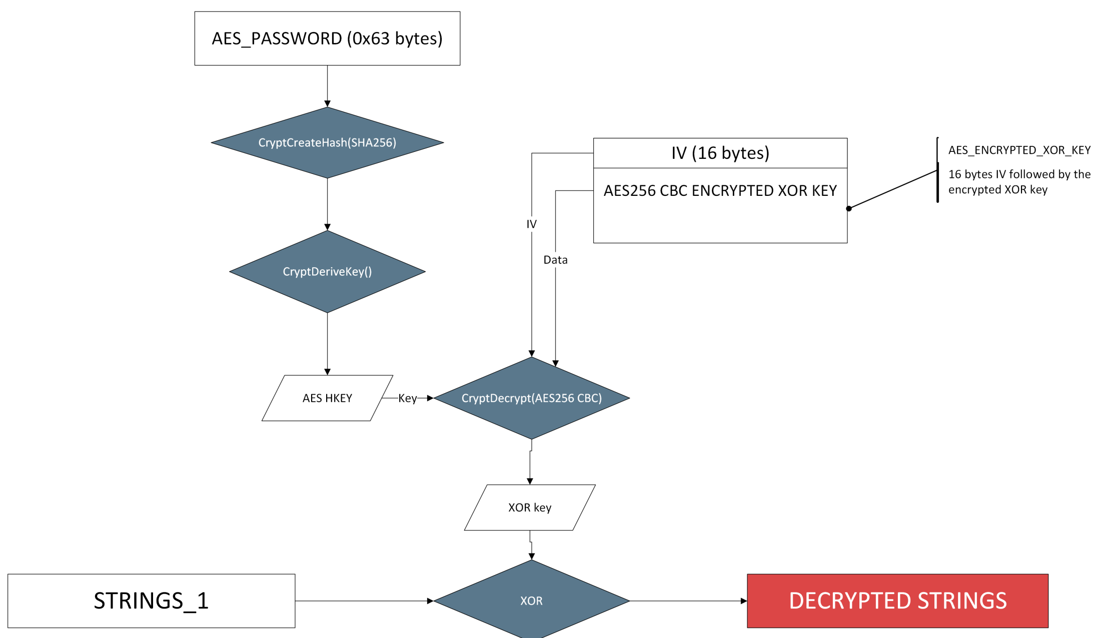

Figure 9: How the strings are decrypted

The good news is that we have all the material we need already in Malcat! Indeed, Malcat already has a [data transform](#) named `CryptDeriveKey`. And actually we don't even need it: what `CryptDeriveKey` does in this specific configuration is just compute the `SHA256` hash of the password and use it directly as key. As for `CryptDecrypt`: it is performing a simple AES 256 decryption in CBC mode, and we also have a transform for this.

> Note: Advapi32.dll crypto functions add/remove padding by default, so make sur to check "unpad" in the transform window

So using exclusively Malcat transforms, we can decrypt the strings manually in a few seconds, as demonstrated in the GIF below:

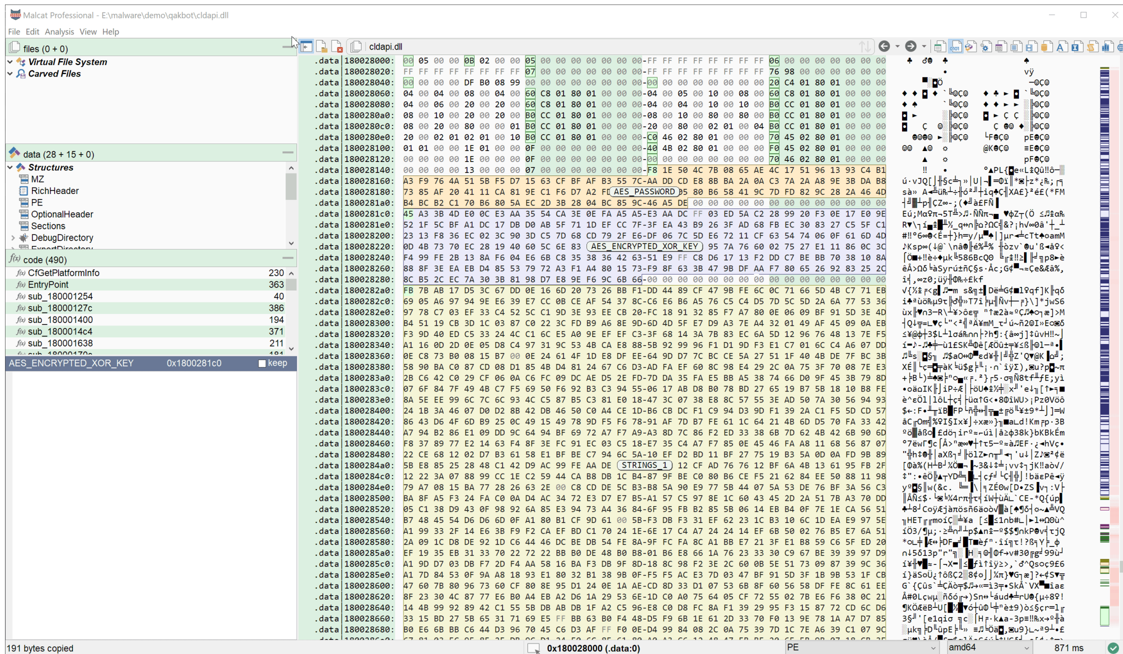

Figure 10: Decrypting th strings using Malcat transforms

The result is shown below:

```
 SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList
ProgramData
netstat -nao
%s "$%s = \"%s\"; & $%s"
net localgroup
powershell.exe
route print
"%s\system32\schtasks.exe" /Create /ST %02u:%02u /RU "NT AUTHORITY\SYSTEM" /SC ONCE /tr "%s" /Z /ET %02u:%02u /
Component_08
ERROR: GetModuleFileNameW() failed with error: ERROR_INSUFFICIENT_BUFFER
net view
ipconfig /all
Self check
T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9
4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIBV
Start screenshot
%s.%u
adrclient.dll
net share
qwinsta
\System32\WindowsPowerShell\v1.0\powershell.exe
at.exe %u:%u "%s" /I
Self test FAILED!!!
Component_07
whoami /all
 /c ping.exe -n 6 127.0.0.1 &  type "%s\System32\calc.exe" > "%s"
```

```
error res='%s' err=%d len=%u
nltest /domain_trusts /all_trusts
.lnk
cmd
schtasks.exe /Create /RU "NT AUTHORITY\SYSTEM" /SC ONSTART /TN %u /TR "%s" /NP /F
%s \"$%s = \\\"%s\\\\; & $%s\"
ERROR: GetModuleFileNameW() failed with error: %u
schtasks.exe /Delete /F /TN %u
arp -a
Self check ok!
cmd.exe /c set
%s %04x.%u %04x.%u res: %s seh_test: %u consts_test: %d vmdetected: %d createprocess: %d
Microsoft
powershell.exe -encodedCommand %S
SELF_TEST_1
microsoft.com,google.com,kernel.org,www.wikipedia.org,oracle.com,verisign.com,broadcom.com,yahoo.com,xfinity.cor
c:\ProgramData
nslookup -querytype=ALL -timeout=12 _ldap._tcp.dc._msdcs.%s
%u;%u;%u;
powershell.exe -encodedCommand
runas
/teorema505
Self test OK.
ProfileImagePath
p%08x
```

Sadly there is no CNC address list nor any valuable configuration data there, beside the CNC http endpoint (/teorema505). So we'll have to dig deeper.

Decrypting the second strings array

There is a second array of encrypted strings than can be found in this binary. This one is of lesser importance and can be decrypted exactly the same way as the first array. The only difference is that a different pair of XOR key and AES password are used. If you are interested, below are the location of the variables relevant to the second array in our Qakbot sample:

Name	**Address**	**Size in bytes**	**Description**
decrypt_strings_2	0x18000de90	0x3f	Decryption *function* for the second encrypted strings array
STRINGS_2	0x1800297a0	0x1836	Second *encrypted strings array*
AES_ENCRYPTED_XOR_KEY_2	0x18002afe0	0xa0	The *XOR key* used to decrypt the string array, but AES256-CBC encrypted

Name	Address	Size in bytes	Description
AES_PASSWORD_2	0x180029700	0x9f	The *password* used to derive the AES256 key for AES_ENCRYPTED_XOR_KEY_2

If you use the same process as for the first array, you should obtain the following strings list: [see on pastebin](#).

The configuration

Locating the configuration

Now having the strings decrypted is all well and good, but our goal is to get the configuration of Qakbot, or at least its list of command and control (CNC) servers. We will stay true to our process and start with data analysis. In the list of decrypted strings presented last chapter, two strings look kind of unusual:

- The 14th string at offset `0x182` in the string array: `T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9`
- The 15th string at offset `0x1a9` in the string array: `4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIBV`

We have seen in the [previous chapter](#) that the string decryption function `decrypt_strings_1` takes as first parameter the index of the string to decrypt, that is its position relative to the start of the encrypted string array. So if we want to know how these two strings are used, we can just look for references to their offset in the code. Let us focus on the *first string*:

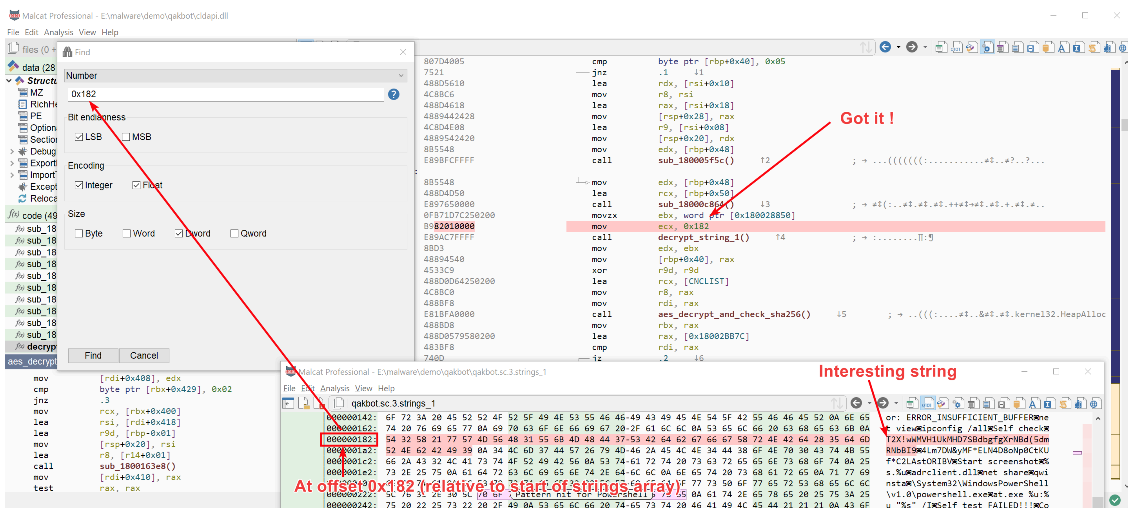

Figure 11: Looking for references to string 0x182

And we rapidely get two candidates: a function at offset `0x18000622c` that we will call `decrypt_CNC` and one at offset `0x18000345c` that we will call `decrypt_params`. And a second good news, both of these function reference a high-entropy buffer (named respectively `CNC_LIST` and `PARAMS`) in addition to our `0x182` string. The address of these functions and variables are given below:

Name	Address	Size in bytes	Description
decrypt_CNC	0x18000622c	0x2cc	Decryption *function* for Qakbot's CNC
CNC_LIST	0x180028852	0x51	Encrypted *CNC list*
decrypt_params	0x18000345c	0x76	Decryption *function* for Qakbot's campaign information
PARAMS	0x180029022	0x51	Encrypted *campaign informations*
aes_decrypt_and_check_sha256	0x180015d14	0x105	Function to decrypt both *encrypted blob*

And a last additional good news: both functions ultimately call our good old `aes_encrypt_decrypt_iv_prefix`. We have already identified this function while reversing the string decryption process: it decrypts an AES256-CBC encrypted buffer prefixed by a 16 bytes IV.

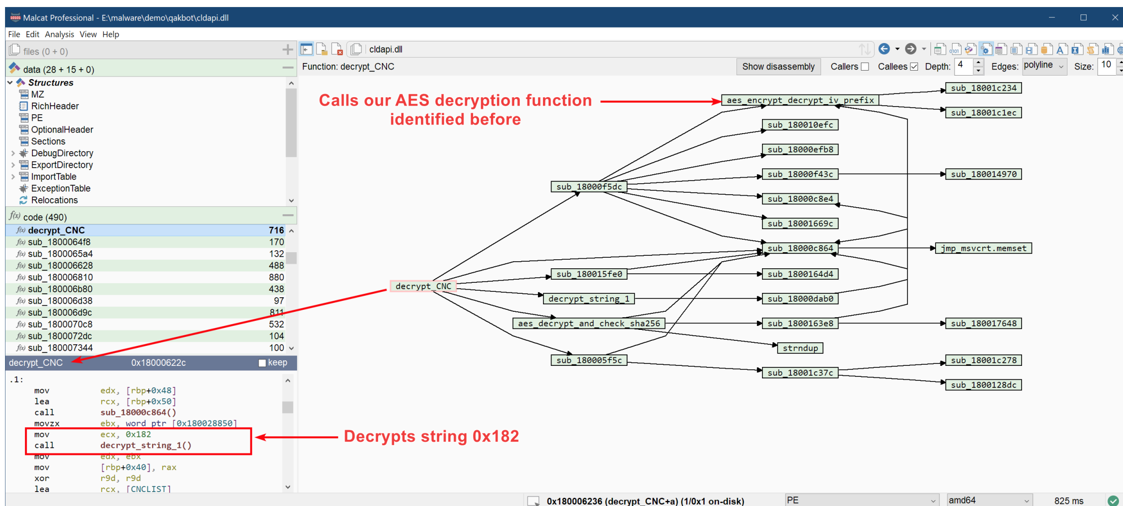

Figure 12: Cnc decryption function candidate

Decrypting the CNC list

If we dig a bit deeper in the function, in particular in `aes_decrypt_and_check_sha256`, we can see that the encrypted blobs `CNC_LIST` and `PARAMS` have a particular structure:

- They are prefixed with their size (16 bits int)
- Afterwards comes a blob identifier, on one byte
- Then we get our already-known encrypted AES blob:
    - A 16 bytes initialisation vector (IV)
    - The actual AES256-CBC encrypted content

The blob format is illustrated below:

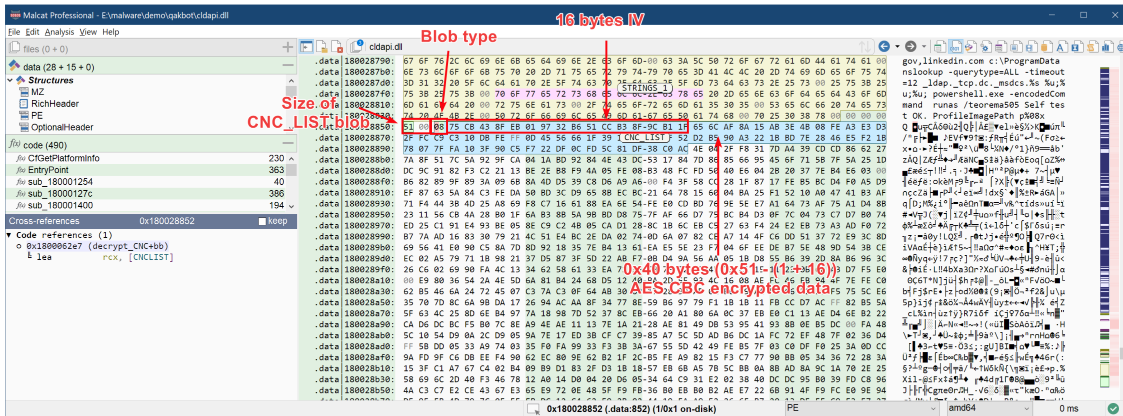

Figure 13: Cnc list encrypted blob

To decrypt the blob, we will use the same procedure as with the strings:

- Compute the SHA256 value of our password `"T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9"` (`7085d1138cbac863a9b4f1bf85a4d413804ef3a3ec52729fa15747a6ee320325`)
- Select the 0x40 bytes of AES encrypted data
- Use Malcat's transform **AES decrypt** in *CBC mode*, set the IV to the 16 bytes prefixing the encrypted data and the key to the sha256 hash
- Don't forget to check **unpad**

After decrypting the `CNC_LIST` blob, we are facing a relatively simple binary structure. A bit of reversing in the function `decrypt_CNC` rapidly tells us everything we need to know to interpret it. The decrypted blob starts with a sha256 checksum, followed by a list of (ip, port) pairs. The details are given below:

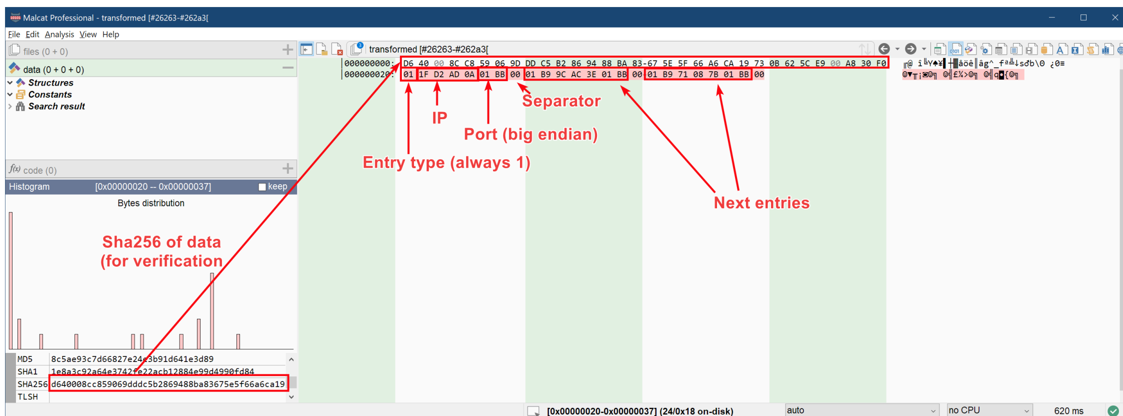

Figure 14: Cnc list decrypted

And that's it! We got our 3 CnC addresses:

- 31.210.173.10:443 (VT)
- 185.156.172.62:443 (VT)
- 185.113.8.123:443 (VT)

Now let us see which kind of information we get with the second buffer `PARAMS` .

Decrypting the campagn informations

The second referenced blob `PARAMS` is encrypted in the exact same way with the same password (the sha256 of `"T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9"`). If you reuse the same decryption process, you should get something like this at the end:

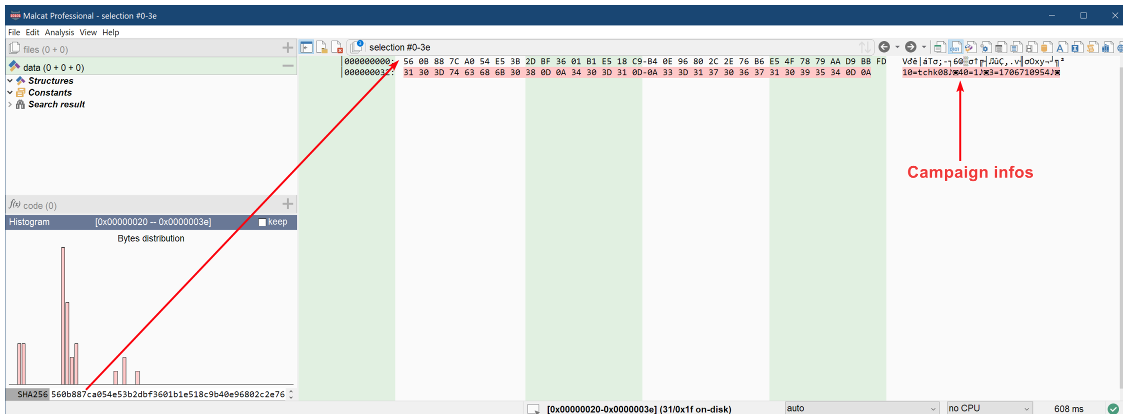

Figure 15: Campaign infos decrypted

We get three parameters:

- The parameter of id `10` seems to be the campaign ID (tchk08)
- The parameter of id `3` seems to be a timestamp, most likely compilation time
- No idea what parameter `40` is for

And with this last piece of information, we will stop our search for Qakbot configuration data.

Scripting everything

The idea

You may have noticed, but decrypting everything was a bit tedious at the end. In this chapter, we will automate the process by writing a python configuration extractor in Malcat. Indeed, Malcat features powerful python bindings which are [documented extensively](). In a script, you have access to the complete analysis object in a somewhat pythonic way.

> Note: if you own a full or pro version of Malcat, scripts can also be run from the command line in [headless mode]()

The idea behind the script is to redo all the steps that we have done manually:

- Collect all interesting referenced buffers in the .data section
- Look if these buffers are prefixed by their size. If not, try to infer the size by looking at constants used in referencing functions code
- Then decrypt everything:
    - For [strings arrays](): try *all possible triples permutations* (strings_array, xor_key_encrypted, aes_password) that we got to decrypt the strings and keep what works

- For [config extraction](): use *any high-entropy string found in the first string array* as AES password and try to decrypt the CNC IPs and the campaign information. Keep what works (we can double-check with the sha256)

This approach (trying all possible keys) may seem not very subtle, but I have found out that string array ordering changes from one sample to the next. I could have used code signatures in order to locate strings decryption functions more easily, but code may change and code signatures are not that robust against recompilation. Analysing data on the other hand is a bit more robust and I hope the script will work for a while.

Since this blog post is long enough, I will just leave you with the relatively well-documented code below. The only notion that may be a bit foreign to you is Malcat's address space and its a2p functions and alike. But beside this little detail, it should be rather easy to understand.

The script

```
 """
name: Qakbot 5.0
category: config extractors
author: malcat

Decrypt strings and extract CnC informations from a (plain-text) Qakbot 5.0 sample
"""

import malcat
import struct
import itertools
import hashlib
import json
import datetime
import re
import math
import collections

from transforms.binary import CircularXor
from transforms.block import AesDecrypt



############################ utility functions

def decrypt_aes_iv_prefix(data:bytes, aes_password: bytes):
    key = hashlib.sha256(aes_password).digest()
    iv = data[0:16]
    data = data[16:]
    return AesDecrypt().run(data, mode="cbc", iv=iv, key=key, unpad=True)
```

```
def get_all_referencing_functions(a:malcat.Analysis, address:int):
    res = []
    for incoming_ref_type, incoming_ref_address in a.xref[address]:
        fn = a.fns.find(incoming_ref_address)
        if fn is not None:
            res.append(fn)
    return set(res)



def entropy(data:str, base=2):
    if len(data) <= 1:
        return 0
    counts = collections.Counter()
    for d in data:
        counts[d] += 1
    ent = 0
    probs = [float(c) / len(data) for c in counts.values()]
    for p in probs:
        if p > 0.:
            ent -= p * math.log(p, base)
    return ent


############################ interesting buffer heuristics

def enumerate_interesting_buffers(a:malcat.Analysis, section_name:str, prefixed_buffer:bool = False):
    section = a.map[section_name]

    # get all incoming xref in the section: denotates the start of a buffer
    data_xrefs = [x.address for x in a.xref[section.start:section.end]]

    for i in range(1, len(data_xrefs) - 1): # let's assume the first and last xrefs will never be interesting
        prev, cur, next = data_xrefs[i-1:i+2]
        prev_off = a.a2p(prev)
        cur_off = a.a2p(cur)
        next_off = a.a2p(next)

        if prefixed_buffer and cur - prev == 2:
            # is it a size-prefixed buffer ? (i.e. there is a referenced word 2 bytes before)
            size, = struct.unpack("<H", a.file[prev_off:cur_off])
            yield cur, size
        elif not prefixed_buffer:
            # we'll look for all immediate constants in referencing functions and see which one could be a size
            for fn in get_all_referencing_functions(a, cur):
                for basic_block in fn:
```

```
                    if not basic_block.code:
                        continue
                    for instruction in basic_block:
                        for operand in instruction:
                            if operand.value and operand.value > 0x10 and cur + operand.value <= next and next -
                                yield cur, operand.value


############################ strings decryption


def get_potential_strings_triples(a:malcat.Analysis):
    # Here we will look for 3 buffers referenced from the same function:
    # one is the strings, one the xor key, one the aes password

    function_to_refs = {}
    done = set()

    # group all interesting buffers by referencing functions
    for address, size in enumerate_interesting_buffers(a, ".data", prefixed_buffer=False):
        if size < 0x20:
            continue
        # find all reference coming from functions
        for fn in get_all_referencing_functions(a, address):
            function_to_refs.setdefault(fn.address, []).append((address, size))

    # now try to find a function referencing 3 interesting buffers
    for fn_address, by_function in function_to_refs.items():
        if len(by_function) < 3:
            # there should be at least 3 references to candidate buffers inside one function
            continue
        # we don't know which is one is the data, xor key or aes password: try all permutations of triples
        for candidate_triple in itertools.permutations(by_function, r=3):
            if not candidate_triple in done:
                done.add(candidate_triple)
                yield candidate_triple

def get_strings_arrays(a:malcat.Analysis):
    res = []
    # tries to decrypt all string arrays candidates
    for strings, xor, aes_password in get_potential_strings_triples(a):

        print(f"Trying strings=({a.ppa(strings[0])}, {hex(strings[1])}), xor=({a.ppa(xor[0])}, {hex(xor[1])}), a

        try:
            # decrypt XOR key using AES
            xor_address, xor_size = xor
```

```
            xor_offset = a.a2p(xor_address)
            xor_buffer = a.file[xor_offset: xor_offset + xor_size]

            aes_address, aes_size = aes_password
            aes_offset = a.a2p(aes_address)
            aes_buffer = a.file[aes_offset: aes_offset + aes_size]

            xor_key = decrypt_aes_iv_prefix(xor_buffer, aes_buffer)

            # decrypt strings using XOR key
            strings_address, strings_size = strings
            strings_offset = a.a2p(strings_address)
            strings_buffer = a.file[strings_offset: strings_offset + strings_size]

            strings_decrypted = CircularXor().run(strings_buffer, key=xor_key).decode("utf8")
            all_strings = strings_decrypted.split("\x00")

            res.append(all_strings)
            print(f"Found {len(all_strings)} strings !")

        except BaseException as e:
            print(f"{e} :(")

    return res



############################ config extraction

def qakbot_config_extraction(a:malcat.Analysis):
    print("Running heuristic to find string arrays ...")
    config_password = None
    strings_1 = []

    # find string arrays
    for string_array in get_strings_arrays(a):
        print(f"\nFound one string array of {len(string_array)} strings:")
        print("\n".join(string_array))
        if "ipconfig /all" in string_array:
            strings_1 = string_array
        print()

    ips = []
    options = {}
    config_passwords = []

    # try to find endpoint
```

```
    for s in strings_1:
        if re.match(r"^/[a-zA-Z0-9_%?=&-]{2,16}$", s):
            options["http_endpoint"] = s
            break

    # try to find password candidates: high-entropy, good length, not a lot of space or backslaches
    for s in strings_1:
        if len(s) > 30 and len(s) < 60 and entropy(s) > 4 and s.count(" ") < 2 and s.count("\\") < 2:
            config_passwords.append(s)
    print(f"Found {len(config_passwords)} password candidates: {', '.join(config_passwords)}")

    # ok now try to look for prefixed buffers:
    for address, size in enumerate_interesting_buffers(a, ".data", prefixed_buffer=True):

        # and try to decrypt using our password candidates
        for config_password in config_passwords:
            print(f"Trying config decryption for {a.ppa(address)}, {hex(size)}) with password {config_password}
            try:
                offset = a.a2p(address)
                buffer = a.file[offset:offset+size]

                # AES decrypt the buffer (skip blob identifer)
                decrypted = decrypt_aes_iv_prefix(buffer[1:], config_password.encode("ascii"))

                # verify checksum
                checksum = decrypted[:32]
                data = decrypted[32:]
                if hashlib.sha256(data).digest() != checksum:
                    raise ValueError("Invalid blob checksum")

                # looks like campaign info?
                if data.count(b"=") >= 2:
                    data = data.decode("ascii").replace("\r", "")
                    d = dict([x.split("=") for x in data.split("\n") if x.strip()])
                    print(f"Found config dictionnary with  {len(d)} entries!")
                    for k, v in d.items():
                        if k == "10":
                            k = "campaign_id"
                        elif k == "3":
                            k = "date"
                            v = datetime.datetime.fromtimestamp(int(v)).isoformat()
                        options[k] = v

                # looks like campaign IPs list?
                elif data.startswith(b"\x01"):
                    for i in range(0, len(data), 8):
                        type, ip, port,_ = struct.unpack_from(">B4sHB", data, i)
```

```
                        if type != 1:
                            raise ValueError(f"Unknown CNC format {type}")
                        ip = ".".join(map(str, struct.unpack("BBBB", ip)))
                        ips.append((ip, port))
                    print ("Found IPs !")

                else:
                    print("Unknwon config data")

            except Exception as e:
                print(f"{e} :(")

    return {
        "cncs": ips,
        "options": options,
    }

################################ MAIN

if __name__ == "__main__":

    config = qakbot_config_extraction(analysis)

    print("\nQAKBOT_CONFIG = ", end="")
    print(json.dumps(config, indent=4))
```

Result

Against the analyzed sample

When run against the last stage `cldapi.dll`, the script will output something like this:

```
 Running heuristic to find string arrays ...
Trying strings=(0x180028150 (.data:150), 0x63), xor=(0x180028150 (.data:150), 0x63), aes_password=(0x180028150 (
Trying strings=(0x180028150 (.data:150), 0x63), xor=(0x180028150 (.data:150), 0x63), aes_password=(0x180028150 (
Trying strings=(0x1800297a0 (.data:17a0), 0x1836), xor=(0x18002afe0 (.data:2fe0), 0xa0), aes_password=(0x180029
Trying strings=(0x18002afe0 (.data:2fe0), 0xa0), xor=(0x18002afe0 (.data:2fe0), 0xa0), aes_password=(0x1800297a0
...
Trying strings=(0x18002afe0 (.data:2fe0), 0xa0), xor=(0x18002afe0 (.data:2fe0), 0xa0), aes_password=(0x18002afe0
Trying strings=(0x18002b190 (.data:3190), 0x9c0), xor=(0x18002b190 (.data:3190), 0x9c0), aes_password=(0x18002b1

Found one string array of 52 strings:
SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList
ProgramData
netstat -nao
%s "$%s = \"%s\"; & $%s"
```

```
...

Found one string array of 185 strings:
%SystemRoot%\SysWOW64\xwizard.exe
.dat
kernelbase.dll
WBJ_IGNORE
mpr.dll
...

Found 2 password candidates: T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9, 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIBV

Trying config decryption for 0x180028852 (.data:852), 0x51) with password T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI9
Trying config decryption for 0x180028852 (.data:852), 0x51) with password 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIBV
Trying config decryption for 0x180029022 (.data:1022), 0x51) with password T2X!wWMVH1UkMHD7SBdbgfgXrNBd(5dmRNbBI
Trying config decryption for 0x180029022 (.data:1022), 0x51) with password 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIB

QAKBOT_CONFIG = {
    "cncs": [
        [
            "31.210.173.10",
            443
        ],
        [
            "185.156.172.62",
            443
        ],
        [
            "185.113.8.123",
            443
        ]
    ],
    "options": {
        "http_endpoint": "/teorema505",
        "campaign_id": "tchk08",
        "40": "1",
        "date": "2024-01-31T15:22:34"
    }
}
```

It works!

Against another sample

But does the extractor script work with other samples too? Let us try with another unpacked Qakbot sample [found on Malpedia](#):

```
 Running heuristic to find string arrays ...
Trying strings=(0x140028150 (.data:150), 0x80), xor=(0x140028150 (.data:150), 0x80), aes_password=(0x1400281e0 (
Trying strings=(0x140028150 (.data:150), 0x80), xor=(0x140028150 (.data:150), 0x80), aes_password=(0x140028280 (
Trying strings=(0x140028150 (.data:150), 0x80), xor=(0x1400281e0 (.data:1e0), 0x94), aes_password=(0x140028150 (
Trying strings=(0x140028150 (.data:150), 0x80), xor=(0x1400281e0 (.data:1e0), 0x94), aes_password=(0x1400281e0 (
Trying strings=(0x140029620 (.data:1620), 0x1825), xor=(0x1400294c0 (.data:14c0), 0xc0), aes_password=(0x140029
...
Trying strings=(0x14002b220 (.data:3220), 0x9c0), xor=(0x14002b220 (.data:3220), 0x9c0), aes_password=(0x14002b

Found one string array of 52 strings:
Component_08
Self test FAILED!!!
route print
whoami /all
...

Found one string array of 185 strings:
kernelbase.dll
mcshield.exe
wmic process call create 'expand "%S" "%S"'
SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths
%ProgramFiles%\Internet Explorer\iexplore.exe
%SystemRoot%\SysWOW64\xwizard.exe
...

Found 2 password candidates: 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIBV, ArpBUw9Lb9ndqXhFTfBst9YHotv92LB7BKvK#ewZn@

Trying config decryption for 0x140028842 (.data:842), 0x61) with password 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORIB
Trying config decryption for 0x140028842 (.data:842), 0x61) with password ArpBUw9Lb9ndqXhFTfBst9YHotv92LB7BKvK#
Trying config decryption for 0x140029012 (.data:1012), 0x51) with password 4Lm7DW&yMF*ELN4D8oNp0CtKUf*C2LAstORI
Trying config decryption for 0x140029012 (.data:1012), 0x51) with password ArpBUw9Lb9ndqXhFTfBst9YHotv92LB7BKvK

QAKBOT_CONFIG = {
    "cncs": [
        [
            "146.70.158.28",
            6882
        ],
        [
            "116.202.110.87",
            443
        ],
        [
            "77.73.39.175",
            32103
        ],
```

```
        [
            "185.156.172.62",
            443
        ],
        [
            "185.117.90.142",
            6882
        ]
    ],
    "options": {
        "http_endpoint": "/teorema505",
        "campaign_id": "bmw01",
        "date": "2024-01-26T12:25:33"
    }
}
```


It works too! Note how the strings inside the two strings arrays are ordered differently from one sample to another.

Conclusion

In this blog post we have learnt how to leverage Malcat's file parsers and data transforms to *unpack a multilayered MSI installer* up to the *final Qakbot sample*. Sticking to *pure static analysis*, and with heavy emphasis on *data analysis*, we have seen how to *decrypt Qakbot's string arrays* and *decode its command and control configuration*. Finally, by making use of Malcat's python bindings, we have written a fully functional *static configuration extractor*. The extractor script does not use any code signature nor any hardcoded value, which should make it hopefully robust to future changes.

I hope that you enjoyed this unpacking/scripting session. Hopefully, you'll find the Qakbot configuration extractor useful for your future analyses. As usual, feel free to share with us your remarks or suggestions!

Source: https://malcat.fr/blog/writing-a-qakbot-50-config-extractor-with-malcat/