Extracting Hancitor’s Configuration with Ghidra part 1

medium.com/@crovax/extracting-hancitors-configuration-with-ghidra-7963900494b5

Crovax January 21, 2022

Crovax

Dec 27, 2021

·

7 min read

Topics covered

- Locate the decryption function
- Determine the decryption algorithm
- Build Ghidra script to decrypt the configuration file
- Build yara detection rule (In part 2)

Hancitor Overview

Hancitor is an older malware loader that is known to drop additional malware on to the system once infected (cobalt strike and some ransomware variants). The entry point is from your typical malspam campaign, that contains a link to an office doc or already has one embedded in the email.

The original dll is usually packed, and once unpacked, will execute the hancitor payload and reach out to one of its hard coded c2's.

> **Original packed samples:**

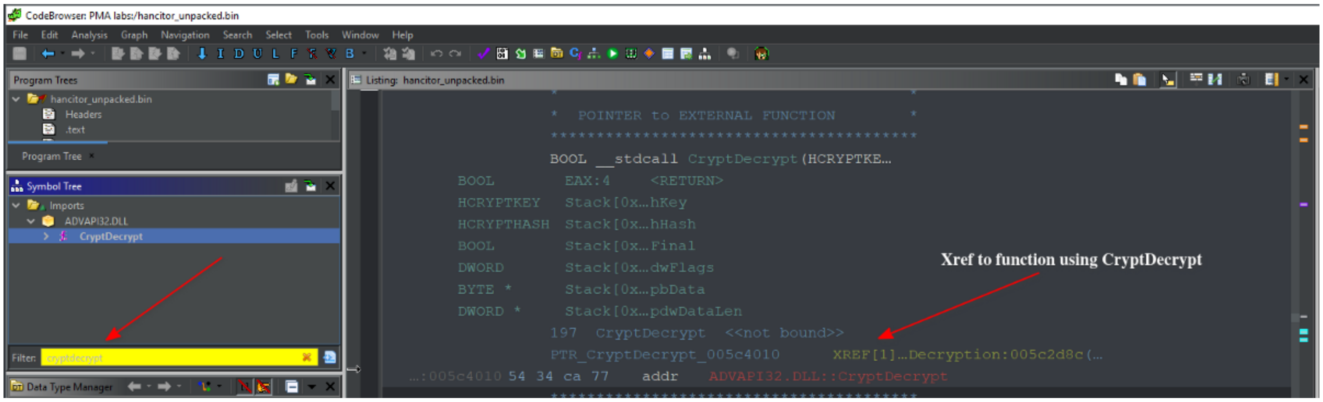
https://www.malware-traffic-analysis.net/2021/06/17/index.html

> **Unpacked Sample**:
>
> SHA-256
>
> 882b30e147fe5fa6c79b7c7411ce9d8035086ad2f58650f5d79aadfb2ffd34f4

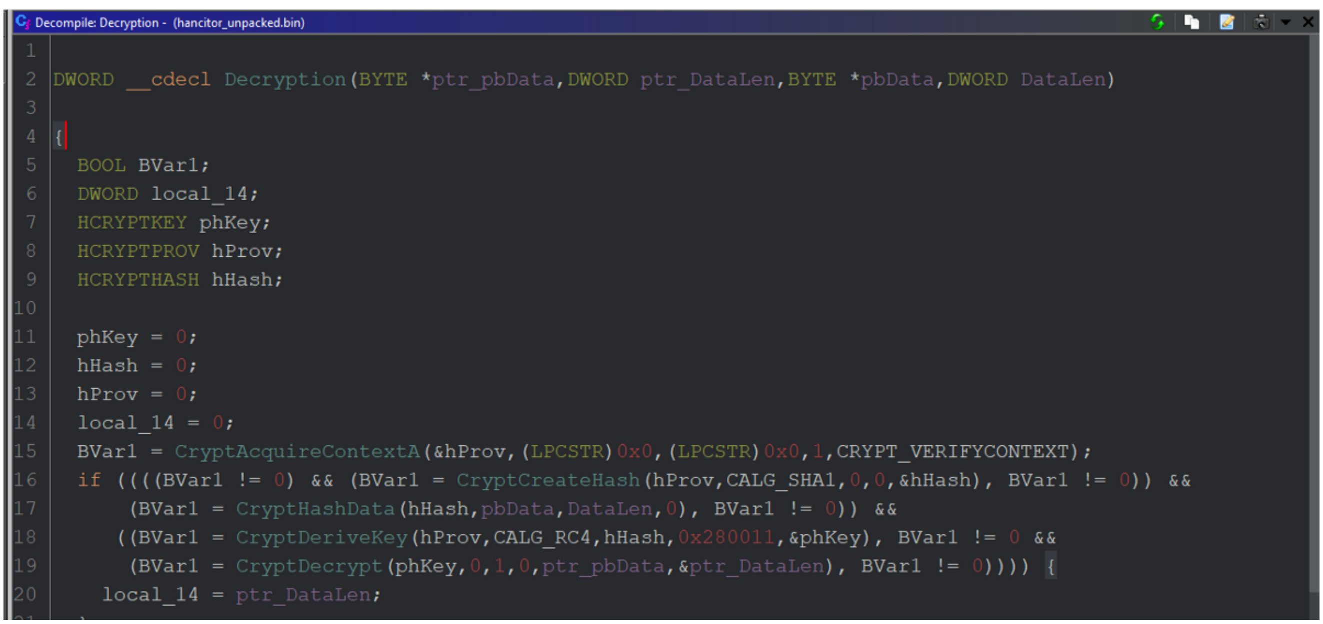
Locating the decryption function

Hancitor leverages the Windows native Cryptographic Service provider (CSP) context to perform its decryption routine. Knowing that, locating the decryption function will be relatively easy.

One way, is to load the binary into Ghidra and look at import table list for one of the cryptographic functions (CryptDecrypt, CryptAcquireContextA etc) then follow the referenced function from there.



Now that we have located the function leveraging the CryptDecrypt routine, we can get a better picture of how the decryption process works. Below is the decompiled view of the function performing the decryption routine. We'll step through each function to determine how the decryption logic works then start programming it out.



CryptAcquireContextA

The CryptAcquireContextA function initiates the call to the Cryptographic Service Provider (CSP) to determine what kind of CSP to use for the CryptoAPI functions. This function is straightforward as its role is just to initiate the CSP context for use. We can determine which CSP is going to be used by the first 'PUSH 0x0" instruction. Since its a null value being used, the default selection is made (native windows cryptographic service provider).

```
:005c2cd0 55              PUSH    EBP
:005c2cd1 8b ec           MOV     EBP, ESP
:005c2cd3 83 ec 14        SUB     ESP, 0x14
:005c2cd6 c7 45 f4 00…    MOV     dword ptr [EBP + phKey], 0x0
:005c2cdd c7 45 fc 00…    MOV     dword ptr [EBP + hHash], 0x0
:005c2ce4 c7 45 f8 00…    MOV     dword ptr [EBP + hProv], 0x0
:005c2ceb c7 45 f0 00…    MOV     dword ptr [EBP + local_14], 0x0
:005c2cf2 c7 45 ec 11…    MOV     dword ptr [EBP + local_18], 0x2…
:005c2cf9 68 00 00 00…    PUSH    CRYPT_VERIFYCONTEXT              ; DWORD dwFlags for CryptAcquireContextA
:005c2cfe 6a 01           PUSH    0x1                              ; DWORD dwProvType for CryptAcquireConte…
:005c2d00 6a 00           PUSH    0x0                              ; LPCSTR szProvider for CryptAcquireCont…
:005c2d02 6a 00           PUSH    0x0                              ; LPCSTR szContainer for CryptAcquireCon…
:005c2d04 8d 45 f8        LEA     EAX=>hProv, [EBP + -0x8]
:005c2d07 50              PUSH    EAX                              ; HCRYPTPROV * phProv for CryptAcquireCo…
:005c2d08 ff 15 20 40…    CALL    dword ptr [->ADVAPI32.DLL::Cryp… ; = 77c69143
:005c2d0e 85 c0           TEST    EAX, EAX
:005c2d10 75 0a           JNZ     LAB_005c2d1c
:005c2d12 e9 89 00 00…    JMP     LAB_005c2da0
```

CryptCreateHash

When the CryptCreateHash function is called, it creates the hashing object to be used in the cryptographic routine, and determines the type of hashing algorithm to be used. Once successful, it returns a handle to the object, for subsequent calls to the other cryptographic functions.

To determine what hashing algorithm is being used, we can look at the 'Algid' value that will be pushed on to the stack. In this case, we see the 'PUSH CALG_SHA1' instruction is being used. So, we know that the hashing algorithm is SHA1.

```
                    LAB_005c2d1c                            XREF[1]…005c2d10(j)
…:005c2d1c 8d 4d fc        LEA     ECX=>hHash, [EBP + -0x4]
…:005c2d1f 51              PUSH    ECX                              ; HCRYPTHASH * phHash for CryptCreateHash
…:005c2d20 6a 00           PUSH    0x0                              ; DWORD dwFlags for CryptCreateHash
…:005c2d22 6a 00           PUSH    0x0                              ; HCRYPTKEY hKey for CryptCreateHash
…:005c2d24 68 04 80 00…    PUSH    CALG_SHA1                        ; ALG_ID Algid for CryptCreateHash
…:005c2d29 8b 55 f8        MOV     EDX, dword ptr [EBP + hProv]
…:005c2d2c 52              PUSH    EDX                              ; HCRYPTPROV hProv for CryptCreateHash
…:005c2d2d ff 15 0c 40…    CALL    dword ptr [->ADVAPI32.DLL::C… ; = 77c6deb6
…:005c2d33 85 c0           TEST    EAX, EAX
…:005c2d35 75 04           JNZ     LAB_005c2d3b
…:005c2d37 eb 67           JMP     LAB_005c2da0
…:005c2d39 eb              ??      EBh
…:005c2d3a 65              ??      65h    e
```

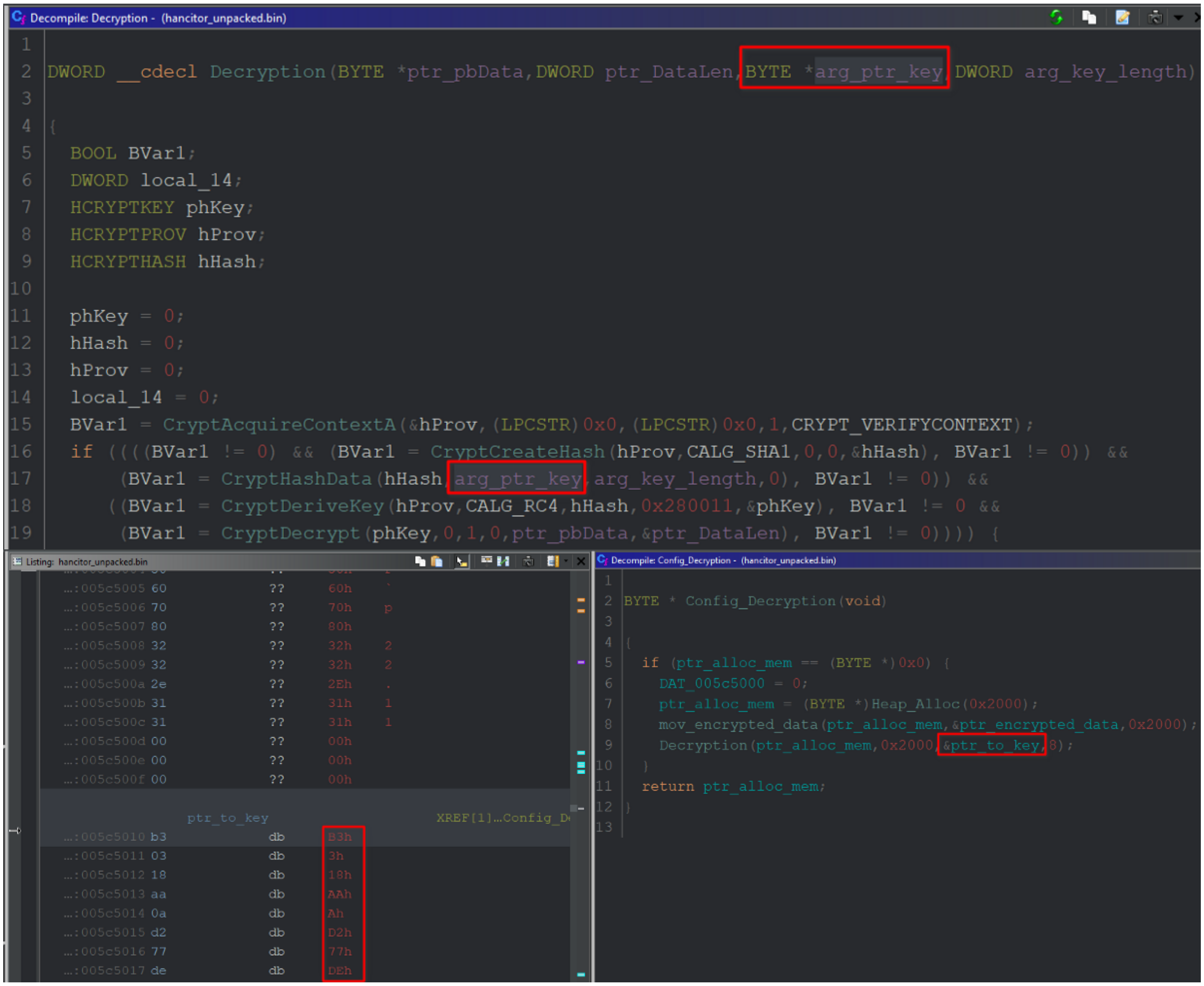
CryptHashData

This function is going to hash the data passed to it, using the previously specified algorithm(SHA1). But what is being hashed and how do we figure it out?

First we need to look back at the arguments being passed into the function (I have labeled it "Decryption') to see what data is being passed and where its located.

by looking at the arguments and their position in which they're being passed in, we need to find out what the data is expecting and where is it located. In order to find this information we can follow the data that is being passed into the function.

Hint: I have already labeled it 😃



As you can see, the third argument being passed is a pointer to the 'key' thats going to be hashed, and the argument being passed to the right of it (8) is the key length.

so we know the following:

Key = b'\xb3\x03\x18\xaa\x0a\xd2\x77\xde'

Key length = 8 bytes

CryptDeriveKey

This next part is going to be a bit tricky. The CryptDeriveKey is going to accept a few parameters:

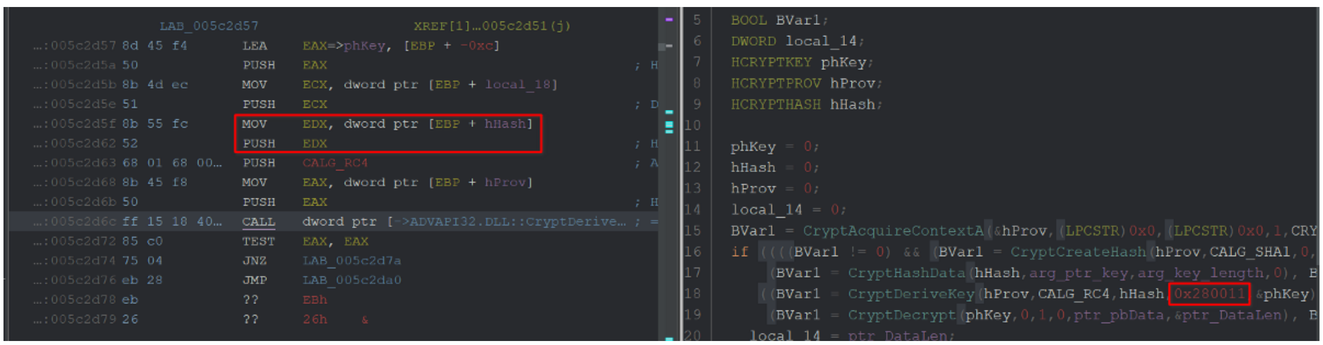
hProv = handle to the cryptographic service provided being used.

Algid = algorithm for which the key is to be generated. In this case, its going to be RC4.

hHash or hBaseData = handle to the hash object that points to the data.

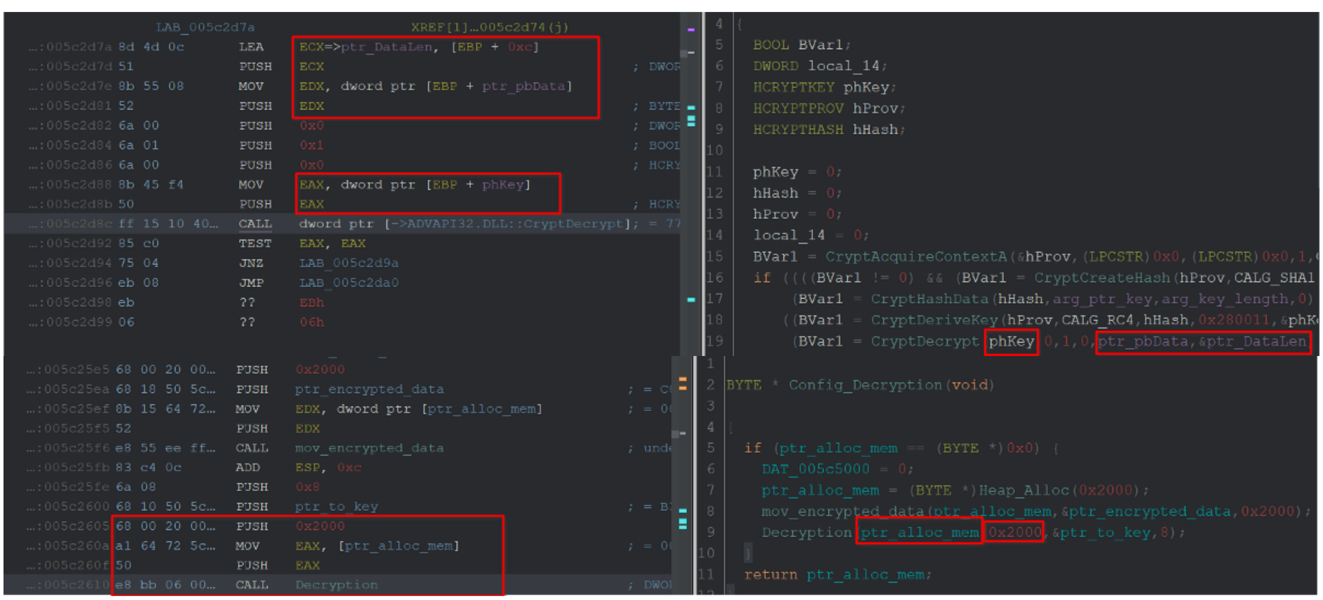
dwFlags = this is going to be key length that is going to be used. Which is the lower 16 bits of the value being passed. In our case that value is 0x00280011, so we only want the lower values or 0x0028 (we can truncate the leading zeros). So by dividing 0x28 / key length(8) that was being passed to the CryptHashData function, we know how many bytes of the RC4 key is going to be used to decrypt the configuration data.

RC4 key: 0x28 / 8 = (5 bytes)



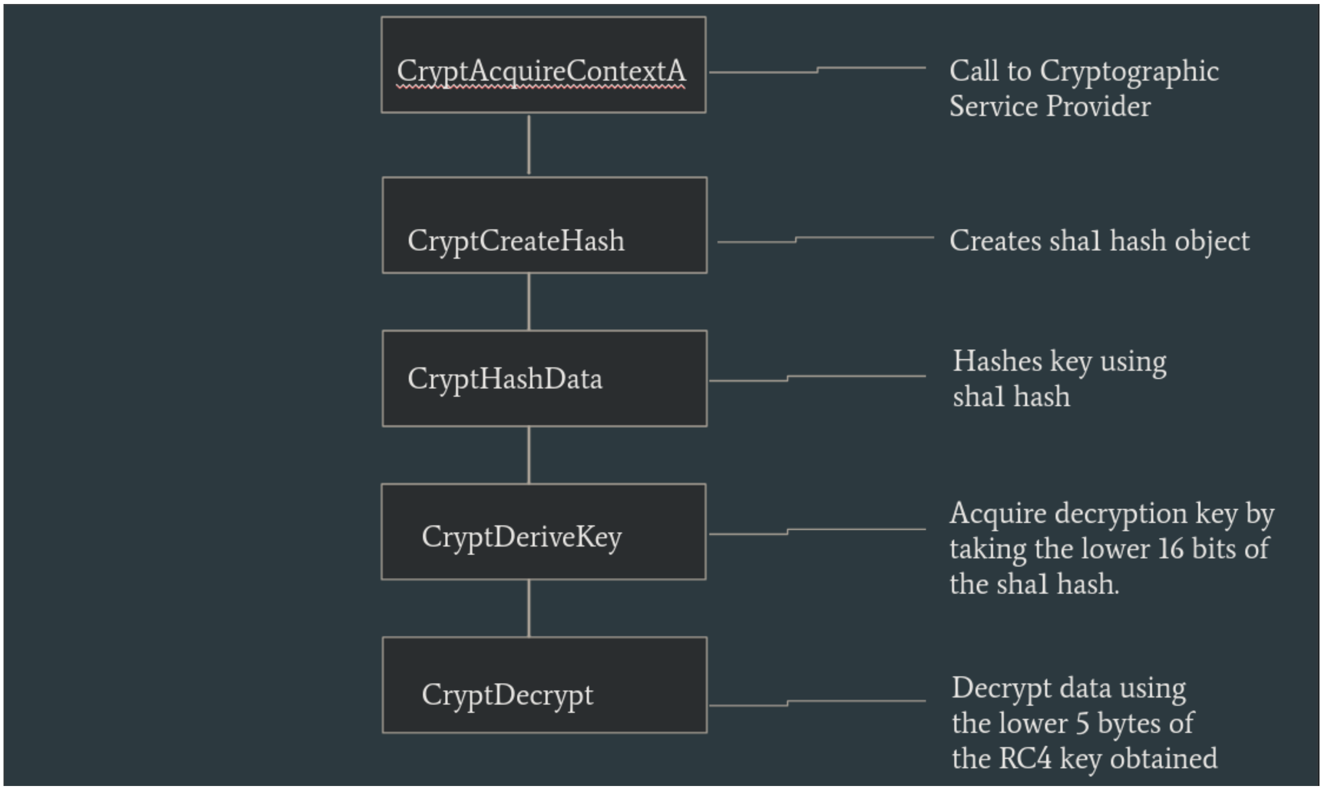
CryptDecrypt

This function is straightforward, it accepts the data to be decrypted, the key used, and the length of the data as parameters. To verify, we can once again look at the arguments that are being passed to the function.



based on the value being passed in, we can see the encrypted data length is equal to 0x2000 bytes.

Below is a graphical representation of the actions performed thus far.



Creating the Ghidra script

We now understand how the decryption process works. The next step is to create a script in Ghidra to automate this whole process, so we can extract the build/campaign id and the c2 domains from this sample.

Note: the python script can be downloaded from my github (link here). Your cursor needs to be pointing to the first address of the encrypted data before running the script. This is due to the 'currentAddress' method.

We know we need to create a sha1 hash of the 8 byte key that's being passed into the Decryption function. So we can copy those bytes out of Ghidra and store them into a variable. The next step is to get a hash (sha1) of the key and extract the first 5 bytes (Key Hash: a956a1e6ff).

Next we need to get the encrypted data using ghidra's getBytes function, and then perform any necessary conversions on the hex values

```
def get_encrypted_bytes():
    get_addr = currentAddress
    get_bytes = list(getBytes(get_addr, 2000))
    converted_bytes = ''
    cByte = ''
    for byte in get_bytes:
        if byte < 0:
            cByte = (0xff - abs(byte) + 1)
        else:
            cByte = byte
        converted_bytes += chr(cByte)

            return converted_bytes
```

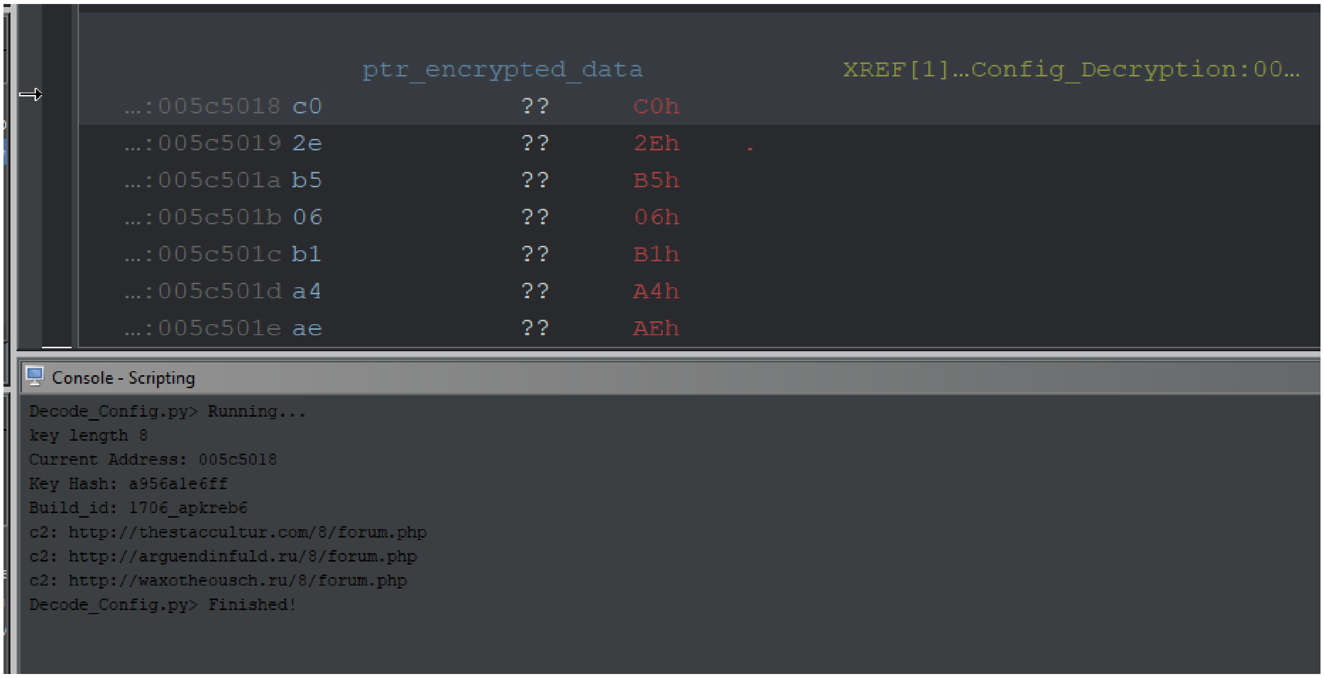
We now have our key and the encrypted data we need to decrypt. The last step is to pass these parameters to a our rc4 decryption function to perform the remaining steps.

```
def rc4_decrypt(key, data):    x = 0    box = range(256)    for i in range(256):
x = (x + box[i] + ord(key[i % len(key)])) % 256        box[i], box[x] = box[x],
box[i]    x = 0    y = 0    out = []    for char in data:        x = (x + 1) % 256
y = (y + box[x]) % 256        box[x], box[y] = box[y], box[x]
out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))    return ''.join(out)
```

Now that we have our decrypted data, we just need to format the data we want and discard any null bytes.

```
print 'Current Address:', currentAddress print 'Key
Hash:',binascii.hexlify(key_hash)get_data = get_encrypted_bytes()config =
rc4_decrypt(key_hash, get_data)build_id = config.split('\x00')[0]print  'Build_id:',
build_idfor string in config.split('\x00')[1:]:    if string != '':        c2 =
string        breakc2_List = c2.split('|')for c2 in c2_List:    if c2 != '':
print 'c2:', c2
```

The output should look something like this 😃



Decode_Config.py> Running…key length 8Current Address: 005c5018Key Hash: a956a1e6ffBuild_id: 1706_apkreb6c2: http://thestaccultur.com/8/forum[.]phpc2: http://arguendinfuld.ru/8/forum[.]phpc2: http://waxotheousch.ru/8/forum[.]php

Conclusion

By breaking down the individual functions of the decryption routine we were able to determine how hancitor was decrypting its c2 domain configuration. We then applied the same process in creating a Ghidra script to automatically perform the same steps statically, to reveal the encrypted data.

In part 2, we well take a similar approach and build a yara rule to test our theory and see if we can detect multiple hancitor variants.

As always, Don’t expect much, as I have no clue what I’m doing. 😃