Malware Analysis Series (MAS): Article 1

by Alexandre Borges
date: DEC/03/2021 - revision: A.1

1. Introduction

Welcome to the ***MAS (Malware Analysis Series).*** Being very honest, in the last four years it was a quite difficult stopping my research job for writing an article as well as would have impossible writing a series of articles, but I think it's feasible now and let's try it. Just to give an example, last time I wrote a superficial article was in 2017 and, certainly, I didn't even remember until a colleague talked about it recently.

The goal is to produce a series of articles on malware analysis and explain since simple malware binaries up to most complex ones, covering a large list of topics such as **unpacking, API resolving, C2 extraction, C2 emulation** and, of course, **reverse engineering** in addition to some **dynamic analysis and**, maybe, use **few de-obfuscation techniques**. When it's necessary, I'll cover other topics such as **COM (Component Object Model)**, **cryptography, IDC/IDA Python** and everything it necessary to help readers to have a better comprehension of analysis.

Furthermore, I will also write short articles covering topics such as malicious documents (on this time I've already release one: https://exploitreversing.com/2021/11/02/malicious-document-analysis-example-1/), programming, de-obfuscation, operating system internals and so on. Nonetheless, the main focus will be ***MAS (Malware Analysis Series).***

As malware analysis produces extensive articles, so I'll break them up in parts 1, 2, and so on, when it will be necessary.

Every article will be published on my new blog (using a different font) and, at beginning of each post, **there will be a PDF version of that article**.

During this series of articles, I'm going to use several tools and try to point where you can get them to make things simpler for you.

I am not going to propose only hard samples because, in my humble opinion, this kind of approach wouldn't help anyone (mainly professionals that aim to learn to something) and, at end, it would be only a waste of time (and an useless show-off). Therefore, we'll analyze different samples, each one with a distinguished level of difficulty, and discuss some lines of code. As I mentioned previously, the strategy is to break up an article in different parts if it's necessary to avoid turning the reading so exhausting.

2. Lab Setup

Explaining about the lab setup, I usually analyze all samples using one or more of the following systems:

- **Windows 7, Windows 8.1 or Windows 10:** If you need a Windows 10 virtual machine, Microsoft continue offering one with expiration time on this website: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

- **REMnux (best distribution for reverse engineering):** https://docs.remnux.org/install-distro/get-virtual-appliance

- **Ubuntu 20.04.x:** https://ubuntu.com/download/desktop

I'll try to avoid using any non-sense techniques and focus on well-known tools. Unfortunately, few of them are not free (***like IDA Pro, which is my favorite one, by far and, in my opinion, the best reversing tool around the world since ever***), but Hex-Rays offers the IDA Free and an affordable paid version **named IDA Home:**

- **IDA Free:** https://hex-rays.com/blog/announcing-version-7-6-for-ida-freeware/
- **IDA Home:** https://hex-rays.com/ida-pro/#main-differences-between-ida-editions

No doubts, you can use **Ghidra** to disassemble, decompile and debug any code just in case you're more comfortable with it: https://github.com/NationalSecurityAgency/ghidra/releases

You'll need a good debugger and certainly the best one is **x64dbg/x32dbg**, which you can download from the following website:

- **x64dbg:** https://x64dbg.com/#start

Additionally, there're tons of **x64dbg plugins** that could be installed to extend x64dbg/x32dbg functionality and are quite recommended during dynamic analysis (mainly to avoid anti-debugging techniques used by malware), so few of them that you could like to install are shown below:

- **ScyllaHide:** it's an advanced anti-debug library that hooks several functions to hide the debugging activity from malware: https://github.com/x64dbg/ScyllaHide/releases

- **Labeless:** it's a quite recommended plugin that provides two key-features for reversers:

    - Label, function name, global variable and comment synchronization between x64dbg and IDA Pro.
    - Dynamic dumping of regions from memory for a debugged process, which will be useful, for example, for dumping the binary after its **task API resolving and/or string decoding**.
    - ***Labeless plugin*** can be downloaded from: https://github.com/a1ext/labeless/releases

- **DbgChild:** it provides an automatic detection of child processes created by the debugged process and automatically attaches a new instance of x64dbg to if the the main process forks a new process, so saving your time in many opportunities. **DbgChild** is available from: https://github.com/David-Reguera-Garcia-Dreg/DbgChild

Other useful plugins exist, but let's wait for the appropriate moment to talk about them. On time: many available plugins don't have been regularly kept by their authors and maintainers, so they could not work anytime. Be careful!

All remaining tools will be shown during our analysis and many future articles.

Last, but not least, this article (and all the following ones) certainly will have mistakes that will be fixed and I'll release new PDF versions reflecting all fixes.

3. Malware analysis goals

No doubts, It's an interesting point: *what are we looking for while analyzing a binary?*

The question is relevant because there're many possible objectives and aspects to be regarded while analyzing a malware. Nonetheless, during a real-world investigation, there are other important areas as malware analysis and, of course, we should consider them in all moments of our analysis:

a. **Memory Analysis:** it's an extremely powerful technique, which has proved its unlimited value in the last 10 years and used as a first-approach method during investigations to understand the malware infection events, its consequences, side effects and makes possible to acquire tons of evidences that might be hard to collect from disk or any other source.

b. **Network Analysis:** it is a quite useful resource (*pcap files*, for example) to understand and detect non-authorized communication (C2 – command and control channel) through traffic analysis and makes artifact gathering (for example, binary files, malicious documents and Cobalt Strike beacons).

c. **Filesystem/Disk Analysis:** the last frontier of any investigation, where we can analyze and detect side effects of a adversary invasion, breaches, frauds, leaks and, of course, malware infections.

Once again, all of them are very important and must be used in all real world investigation. However, let's return to the key point: why should you learn about reverse engineering and, in special, malware analysis? Simple: through malware analysis you have the opportunity to learn from the source of the evil about intentions and objectives of the the adversary and not only its effects. In other words, you can learn techniques, tricks, evasion strategies and, if you're lucky, you'll can collect important artifacts to make the correct attribution (most of the time, it's a hard task) and, who knows, help to arrest the bad guys.

Therefore, before starting any reversing task, we should remember that there're many questions that we should to consider and ask to ourselves:

- Is the binary **packed**? If it's, so malware is using a well-known packer or a custom one?
- What's the **networking communication** technique/API set being used by malware? From available techniques such as ***Winsock2, Wininet, COM (Component Object Model)*** or something in a lower level such as ***WSK (Winsock Kernel)*** or even custom implemented technique, which is being used?
- Is there any **code injection** or **hooking technique being used**? Which one?
- What are the **anti-forensic techniques** used? Is there any **anti-debugging** technique? **Anti-disassembly**? **Anti-VM/Sandbox**?
- Is there any **API/DLL encoding**?
- Are **strings encrypted**?
- What **synchronization primitives** are being used by the malware? Sometimes they hide important anti-debugging techniques.

- What are **cryptography algorithms** being used by the malware?
- What **persistence methods** are being used by the threat: *Registry, services, tasks or kernel drivers*?
- Is there any **shellcode** being injected into a operating system process?
- Is there any **file system mini-filter driver** being installed by malware?
- If there's a **kernel drivers** being installed, is there any **callback** (a kind of modern hook) or **timer** being installed?

In this first article, we'll focus on only **two short objectives**:

a. **unpacking the malware**
b. **extracting and decrypting its C2's configuration data**

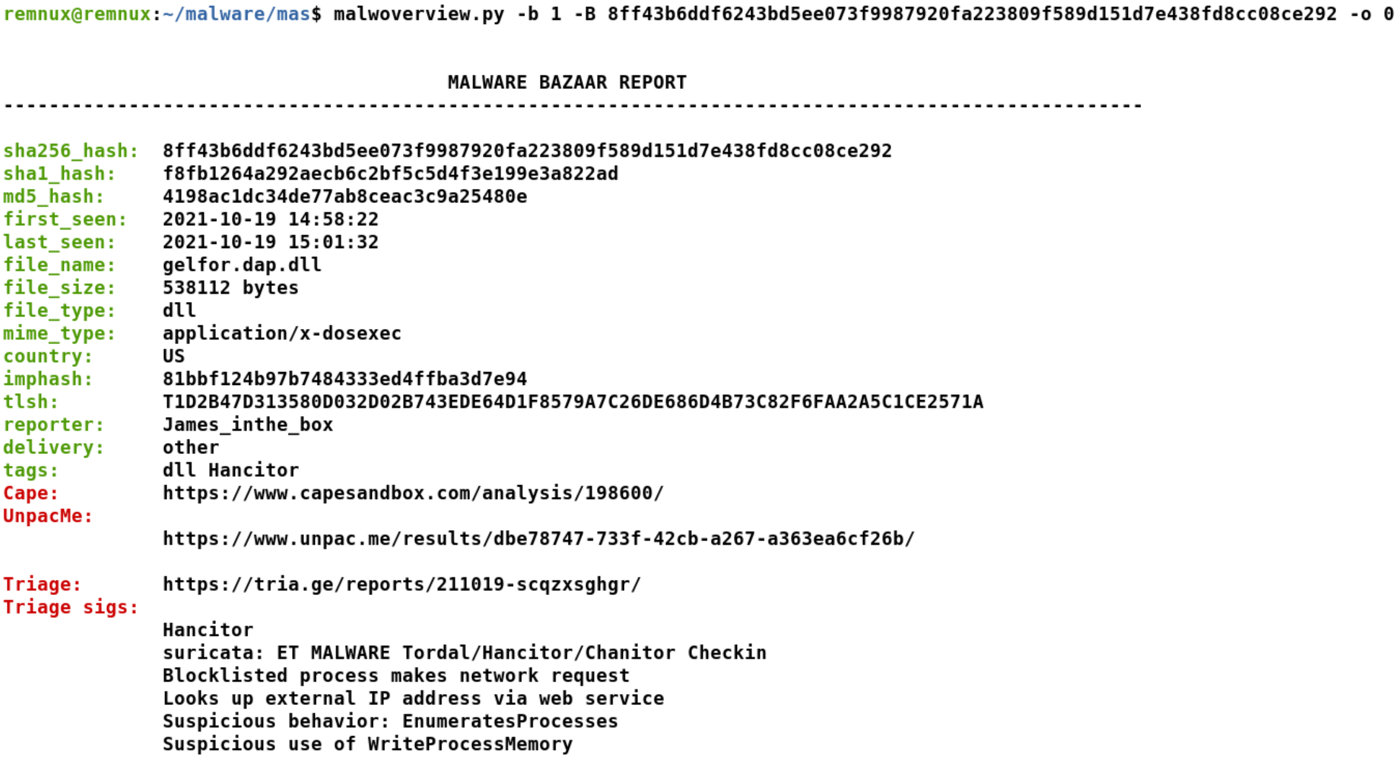
We're reviewing some well-known techniques for unpacking malware threats as well as different methods to extract C2's configuration data. Furthermore, I am going to provide a minimum background for some basic topics to help readers to be able to continue their own research about the mentioned topics.

4. Gathering initial information

The first sample has the following hash:

(SHA256) **8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292**

We're able to collect so much information from many endpoints such as **Malware Bazaar**, as shown in the figure below:



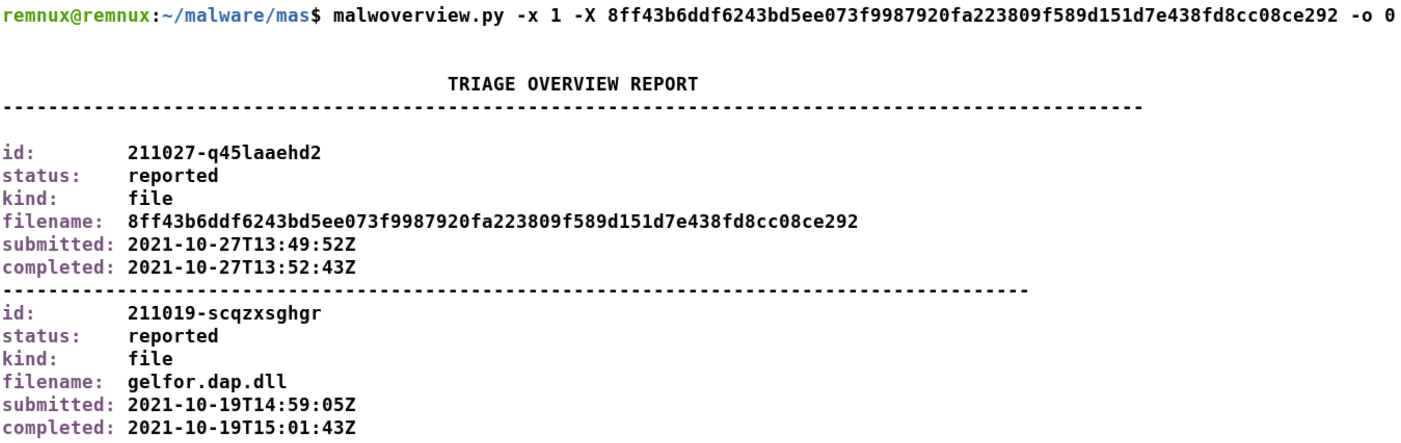
```
remnux@remnux:~/malware/mas$ malwoverview.py -b 1 -B 8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292 -o 0


                                       MALWARE BAZAAR REPORT
----------------------------------------------------------------------------------------------------

sha256_hash:  8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292
sha1_hash:    f8fb1264a292aecb6c2bf5c5d4f3e199e3a822ad
md5_hash:     4198ac1dc34de77ab8ceac3c9a25480e
first_seen:   2021-10-19 14:58:22
last_seen:    2021-10-19 15:01:32
file_name:    gelfor.dap.dll
file_size:    538112 bytes
file_type:    dll
mime_type:    application/x-dosexec
country:      US
imphash:      81bbf124b97b7484333ed4ffba3d7e94
tlsh:         T1D2B47D313580D032D02B743EDE64D1F8579A7C26DE686D4B73C82F6FAA2A5C1CE2571A
reporter:     James_inthe_box
delivery:     other
tags:         dll Hancitor
Cape:         https://www.capesandbox.com/analysis/198600/
UnpacMe:
              https://www.unpac.me/results/dbe78747-733f-42cb-a267-a363ea6cf26b/

Triage:       https://tria.ge/reports/211019-scqzxsghgr/
Triage sigs:
              Hancitor
              suricata: ET MALWARE Tordal/Hancitor/Chanitor Checkin
              Blocklisted process makes network request
              Looks up external IP address via web service
              Suspicious behavior: EnumeratesProcesses
              Suspicious use of WriteProcessMemory
```

[Figure 1]

According to the **Figure 1**, we have some important information:

- The target malware seems to be from **Hancitor** family.
- It uses **EnumerateProcesses( )** function, so it could be interesting to understand whether any special reason for that (code injection, for example);
- **WriteProcessMemory( )** is triggered as usually we have seen in **unpacking procedures** and **code injection**, so no news is a good news here.

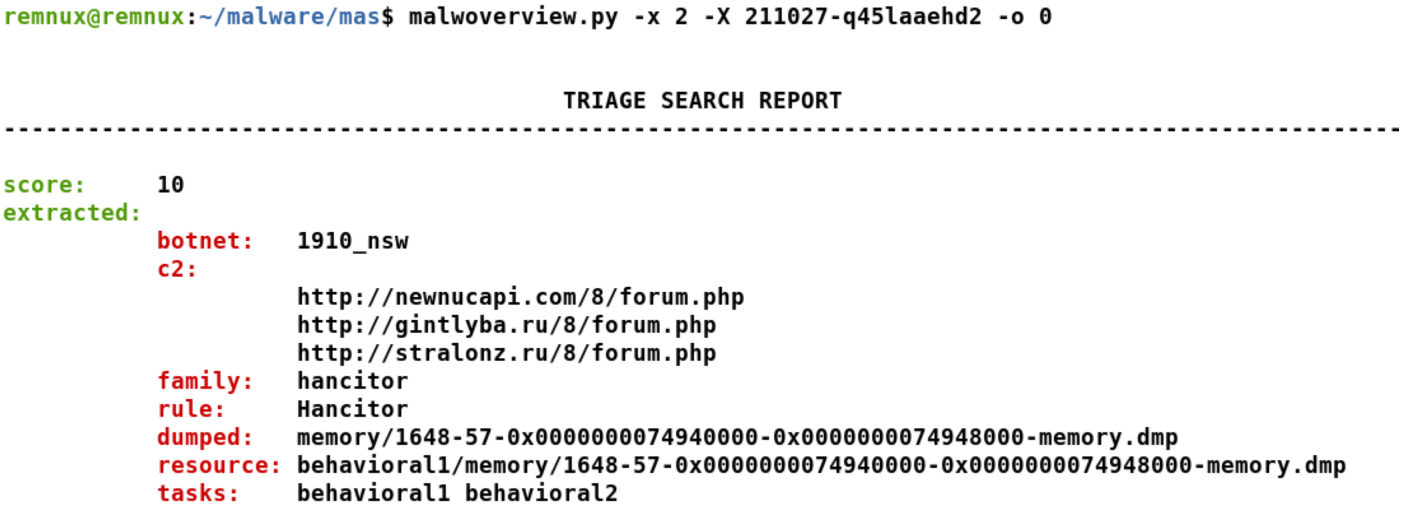
Extending our data acquisition, we can check the sample on **Triage** for collecting further information:



```
remnux@remnux:~/malware/mas$ malwoverview.py -x 1 -X 8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292 -o 0


                                        TRIAGE OVERVIEW REPORT
----------------------------------------------------------------------------------------------------

id:         211027-q45laaehd2
status:     reported
kind:       file
filename:   8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292
submitted:  2021-10-27T13:49:52Z
completed:  2021-10-27T13:52:43Z
------------------------------------------------------------------------------------------
id:         211019-scqzxsghgr
status:     reported
kind:       file
filename:   gelfor.dap.dll
submitted:  2021-10-19T14:59:05Z
completed:  2021-10-19T15:01:43Z
```

[Figure 2]

There're other tasks id related to this sample, but let's to focus on the first one only. Details about the first task id can be shown by executing the following command (the output was truncated):



```
remnux@remnux:~/malware/mas$ malwoverview.py -x 2 -X 211027-q45laaehd2 -o 0


                                        TRIAGE SEARCH REPORT
----------------------------------------------------------------------------------------------

score:      10
extracted:
            botnet:    1910_nsw
            c2:
                       http://newnucapi.com/8/forum.php
                       http://gintlyba.ru/8/forum.php
                       http://stralonz.ru/8/forum.php
            family:    hancitor
            rule:      Hancitor
            dumped:    memory/1648-57-0x0000000074940000-0x0000000074948000-memory.dmp
            resource:  behavioral1/memory/1648-57-0x0000000074940000-0x0000000074948000-memory.dmp
            tasks:     behavioral1 behavioral2
```

[Figure 3]

Of course, we could obtain additional information about the sample, but it's enough for now because we already have possible C2 (URLs).

Our next step is to find out whether this sample is packed (as most of the malware threats) or not. Furthermore, if it's packed, so we will learn how to unpack it using a debugger like x64dbg.

Nonetheless, let's review few concepts about unpacking malware on Windows systems.

5. Unpacking Concepts Review

Every single time I've heard someone talking about unpacking it seems impressions convert to the same conclusion: it might be not so easy. Of course, as we mentioned previously, unpacking a sample is likely the first step before possible string decryption and API/DLL resolving, for example, but we need to start from somewhere and with a goal.

There's a long list of reasons and aspects associated to motivations about packing a malicious code:

- It **makes the malicious code "hidden" from AV**. Of course, it isn't so hidden, but it's a soft evasion technique that make analyst's life a bit harder and, eventually, cause some problems to defenses.
- Packed sample **doesn't reveal the actual goals** of the actual malware.
- It could be **difficult unpacking it dynamically** due many **anti-analysis techniques** (**anti-debugger and anti-vm tricks**) to be circumvented.
- Malware usually packs valuable code in **several layers** using customized routines.
- Eventually, the whole malware or only the unpacking code might be **polymorphic**.

There are a lot of old well-known packers which we have procedures to unpack the code generated hidden by them, but most of malware authors have been used **customized packers** to turn code undetectable under security defenses monitoring. Additionally, there are some **special packers (as known as protector)** such as **Themida, Arxan, VMProtect, Agile .NET** and many others that usually virtualize their instructions and implement all kind of **anti-forensic and obfuscation techniques**, where few of characteristics are presented below:

- They have been used on **64-bit binaries**.
- **The IAT (Import Address Table) might have been removed** or, at maximum, there could be only one imported function.
- As usual, most **strings are encrypted**.
- **Memory integrity is checked and protected**, so it isn't possible to dump a clean executable from memory because original instructions are not completely decoded there.
- **Instructions are virtualized** and, surprisingly, translated to **RISC instructions**.
- These **virtualized instructions are encrypted** on memory.
- The **obfuscation is stack-based**, so it quite difficult to handle virtualized code using static approach.
- **Most of virtualized code is polymorphic**, so there are many virtual instructions referring to the same original instruction.
- There're thousand lines of **fake "push" instructions** and, of course, many of them contains **dead and useless code.**
- These protectors implement **code reordering using unconditional jumps**.
- All these modern packers use **code flattening**, many **anti-debugging** and **anti-vm techniques**.
- **Not all x64 instructions are virtualized**, so you will find a binary code containing a mix of virtualized and not virtualized (native) instructions.
- Most of time, **prologues and epilogues of functions are not virtualized**.
- Original **code section could be "splitted" and/or scattered around the program**, so instructions and data would be mixed.
- **Instructions referring to imported function might be zeroed or even replaced by NOP**, so in this case these "**references" will be restored dynamically**. Sometimes these same references aren't

zeroed, but **replaced by jump instructions using RVA to the same import address**, as well known as "IAT obfuscation".

- As used in shellcodes and common malware, **API names are hashed**.
- The **translation from native register to virtualized register is usually one-to-one, but not always**. Furthermore, there is a context switch component that is responsible for transferring registers and flag information into the virtual machine context.
- **Virtual machines handlers come from data blocks**.
- Many native **APIs are redirected to stub code that forwards the call**.
- Obfuscation techniques such as **constant unfolding, pattern-based obfuscation, control indirection, inline functions, code duplication** and mainly **opaque predicate** are used.

Before and during the unpacking task, there're many observations and questions that we could think about:

- Is the malware **really packed**?
- What are the **evidences of having a packed code**?
- Does the malware perform **self-injection** or **remote injection**?
- Does the malware perform **self-overwriting**?
- **Where is the payload being written**?
- **How** the payload is **going to be executed**?
- What are **evidenced of having an unpacked code** after the unpacking procedure?
- Are there **additional packed layers**?

The first point of the list above rises a key question: **how do we know whether a malware is really packed?**

There isn't an easy and definitive answer to this question, but eventually a set of two or more evidences could indicate that sample is packed:

- The binary sample has **few imported DLLs and functions**.
- There are many **obfuscated strings**.
- Existence of **specific system calls**.
- **Non-standard section names**.
- **Non-common executable binary sections** (only .text/.code section should be executable)
- **Unexpected writable sections**.
- **High entropy sections** (usually above 7.0, but not always – this is a weak indicator).
- **Substantial difference between the raw size and the virtual size of a section**.
- **Zero-sized sections**.
- **Missing APIs related to network communication**.
- **Lack of essential APIs for the malware functionalities (Crypt* functions in a ransomware, for example).**
- **Unusual file format and headers**.
- **Entry-point pointing to other section than .text/.code section**.
- Significant size of **resource section** (.rsrc section) followed by **LoadResource( )** function in the code.
- Presence of an **overlay**.
- Opening it up on IDA Pro and observing a big amount of **data or unexplored code on colored bar**.

It's very relevant and suitable to highlight one point: **the occurrence of only one characteristic from the list above doesn't determine that the malware is packed**. Thus, it's quite important to consider two or more of them. Furthermore, there are further observations to be considered:

- Most samples resolve dynamically their APIs using **LoadLibrary( )** followed by **GetProcAddress( )**, for example (**except on reflective code injection cases**).
- **Network APIs** also could be **dynamically resolved**.
- **Malformed headers** might be a bit **difficult to detect** at the first analysis.
- **Big resource section might not be relevant** because it might contain only GUI artifacts and digital certificates.
- There might be a **mix of encrypted/obfuscated strings and plain text strings**, so making a bit harder to decide whether the binary is or not packed.

The unpacking procedure using a debugger might bring a list of **challenges to be understood and bypassed**:

- **Anti-debugging techniques** (**time checking, CPUID, heap checking, debugging flag checking, NtSetInformationThread( ),** and so on), so it's recommended to use an **anti-debugger plugin such as ScyllaHide** (https://github.com/x64dbg/ScyllaHide) on **x64dbg/x32dbg** or even **StrongOD on OllyDbg** (there're some repositories containing OllyDbg and all associated useful plugins already built-in. Use Google for finding them).
- **Anti-VM tricks** checking for VMware, VirtualBox, Hyper-V and Qemu artifacts, for example.
- **Filename, hostname and account checking** (avoid using the hash as filename).
- **Available disk size on virtual machine** (it's recommended 100 GB, at least)
- **Number of processors on the testing virtual machine** (two or more would be suitable)
- **Uptime** (try to keep a virtual machine snapshot with uptime above 20 minutes).
- Many **non-sense calls** (result is not used any longer) and **non-existing APIs (fake APIs)**.
- **Exception handlers** being used as anti-debugging technique.
- **Software breakpoints being cleared** and **registers (DR#) being manipulated** (anti-breakpoint techniques)
- **Hash functions using typical algorithms** (for example, crc32, conti, add_ror13,…) being used.
- **Malicious code checking for well-known tools such Process Hacker, Process Explorer, Process Monitor** and so on (it's recommended to rename these executable binaries before using them).

Unfortunately, **anti-VM tricks** and **anti-debugger techniques cannot be always handled by plugins and we will have to manage to bypass them using the debugger**. In this case, we have an interesting possibility of using a different debugger like **WinDbg to manage some malware threats expecting for ring 3 debuggers only and not kernel debuggers (a recent case is the GuLoader malware).**

Even during or after unpacking procedures, **we could need to fix the resulting binary** because one or more of following issues:

- The **DOS/PE header could have been destroyed** on the memory **or modified** by a compress library.

- In many cases, when you extract a binary from memory, **you need to clean it up because there's some garbage before its DOS header (MZ signature) and PE header.**

- The **entry-point (EP)** could have been **zeroed or wrong**.

- The unpacked binary might have its **Import table destroyed** due to the fact it has been dumped and its address refers to virtual addresses (**mapped version** instead of **unmapped version**), so showing **unaligned sections** or **none section**.

- **Base address is wrong**.

- **PE format's field presents some inconsistence**.

- It could be **hard to determine the OEP (Original Entry Point)** , which usually appears after a transition from unpacker code using an indirect call (**call [eax]** or **jmp [eax]**, for example). Additionally, **existence of non-resolved APIs could be an evidence of the malicious code hasn't reached the OEP yet.** On time: **OEP is the entry point (EP) of an executable before it being packed**. After it has been packed, a new EP is associated to packer itself.

- **Mutexes** being used as a kind of "**unlock key" between two unpacking layers**. In this case, the second stage of unpacking doesn't happen without the first stage has happened, and if it's happened, so the mutex existence is confirmed.

- The code might be **executing self-overwriting**.

- The **first stage** of unpacked code **doesn't run from any directory**, but only from a specific one.

- **You can have extracted a decoy binary**. In many real cases, malware authors packs one or more useless executables as decoy to consume time of the analyst. Thus**, it would be wise not believe you've unpacked the correct binary from memory at first attempt**.

This list of issues is very limited and there're endless other possible side effects on unpacked binaries. Of course, distinct solutions for each one of these presented issues exist and they will be explained and given examples in the next articles of this series. Anyway, few approaches for handling some issues are:

- **Copy a good PE header from another executable** (or from the own malware sample) and **align sections** considering whether the unpacked binary is **unmapped (.text section usually starts on 0x400)** or **mapped (.text section usually starts on 0x1000).**

- **Align sections of an unpacked binary** (mapped addressing) **by fixing its respective Raw Address and Raw size**. This action usually **fixes the Import Table** and makes possible to visualize imported functions without any issues. Pay attention to possible "traps": **some unpacked binaries don't show its Import Table until you've aligned their sections**. However, **other malware threats don't have any function in the Import Table even after you having unpacked the binary**, so it doesn't mean you made any mistake, but it does that the **malware resolve all its APIs dynamically**.

- **Reconstruct the IAT** and **forcing the OEP** (Original Entry Point).

- If you're facing problems in finding the OEP, so remember that **OEP likely comes after the IAT has been resolved**. In this case**, one of possible approaches would be to check whether IAT is already resolved (check for Intermodular Calls on x64dbg or OllyDbg)** or **setting a breakpoint on a critical API that would be executed during a key operation** of malware (**CryptoAcquireContext( )** in ransomware threats, for example) because certainly IAT will be resolved when execution reaches these critical APIs . Afterwards, the suggestion is **looking for unconditional jumps to specific memory addresses or even indirect calls (call [eax], for example**). Another interesting approach would be **using the graphical visualization of a debugger ("g" on x64dbg)** and **check for these transition points (indirect calls or unconditional jumps for memory addresses) at the last "code blocks"**. Finally, a specialized tool might help you to find out the OEP. As you've noticed, there isn't a single approach to do this.

- **Adjust the base address** to match with the segment's base address dumped from memory.

- **To detect malware performing self-overwriting**, we could try to **set a breakpoint on the .text/.code section**. In this case, we could choose to trigger this breakpoint during code writing or execution.

- In two-stage unpacking cases, the **first unpacked binary might be a DLL**. Therefore, depending on the context, **it might be useful to convert the DLL binary to executable**, and there're many ways to accomplish this task, **but my favorite method is editing the PE header to alter the Characteristics field and make the entry of the exported function as an entry-point**.

To visualize, handle and fix most of issues after unpacking a binary you can use the following well-know tools such as:

- **PEBear** is an excellent tool **written by Aleksandra Doniec (a.k.a Hasherezade)** that's used to visualize details of a PE Header and fix many binary issues. You can download this tool from: https://github.com/hasherezade/pe-bear-releases

- **Pestudio** is a great tool **written by Marc Ochsenmeier** and it's mainly used to triage and collect different information of a potential malware. The tool (free and paid versions) are available here: https://www.winitor.com/features

- **CFF Explorer**, which makes part of Explorer Suite, it's an well-known PE Editor that is used to visualize and fix PE headers. The Explorer Suite can be downloaded from: https://ntcore.com/?page_id=388

- **pe_unmapper** is another tool **written by Aleksandra Doniec (a.k.a Hasherezade)** that can be used for converting a PE binary from mapped version to unmapped version, so fixing all PE alignment issues. This tool can be downloaded from: https://github.com/hasherezade/libpeconv/tree/master/pe_unmapper

- **Scylla** is an amazing x86/x64 Import Reconstructor that is already embedded in x64dbg. If you need the standalone version, so you can download it from: https://github.com/NtQuery/Scylla

- **HxD** is an excellent hex-editor that we could be used, for example, to check and fix PE headers manually. It can be downloaded from: https://mh-nexus.de/en/hxd/

- **XVI32 Hex Editor** is another interesting hex-editor that is great to clean up dumped memory regions to isolate the unpacked binary. XVI32 Hex Editor can be downloaded from: http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm

Once again, remember that unpacking is only the first obstacle during malware analysis, and many other hard challenges such as string de-obfuscation, API resolving, C2 configuration extraction, C2 emulation and other topics are also in out list. This project will cover several unpacking situations and all of these mentioned tasks in the next articles.

Now we have a minimal knowledge about unpacking process, issues and solutions, it's time to review different code injection techniques, which could help you to have a better comprehension about unpacking.

6. Code Injection Review

Code injection is a supported operation on Window systems and, of course, it a quite useful evasion method due to the fact that a malware is able to inject (write) a malicious code into a memory region (some people use the term "segment") of the process itself (self-injection) or a remote one (remote injection), and this payload will be executed on the target context as whether it made part of it and without leaving many evidences. Furthermore, the source process (malware) can cleanly terminate itself while the malicious payload continue being running in a supposedly good process (for example, explorer.exe and svchost.exe). At end, it's a stealth approach for evading security defenses.

It's quite interesting to figure out that a long list of mitigations and protections such as **Code Integrity Guard, Extension Point Disable Policy, Control Flow Guard, Code Integrity Guard, Dynamic Code Restriction and Arbitrary Code Guard** (a kind of update of Dynamic Code Restriction) exist since Windows 8.1 (mainly Windows 10 and 11) and it isn't so easy to perform code injection on these Windows versions without being detected and prevented. Further information about these mitigation and protection can be read on: https://docs.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10 , https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/customize-exploit-protection and https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/windows-10-memory-protection-features/ba-p/259046.

There are excellent public documents explaining several code injection techniques, but at a summarized way, the main **code injection techniques** are the following ones:

- **DLL Injection:** this old technique is used to force a process to load a DLL. Main potentially involved APIs: **OpenProcess( ), VirtualAllocEx( ), WriteProcessMemory** and **CreateRemoteThreat | NtCreateThread( ) | RtlCreateUserThread( ).**

- **PE Injection:** in this technique a malicious code is written and, consequently, forced to be executed in a remote process or even in the own process (self-injection). Main related APIs: **OpenThread( ), SuspendThread( ), VirtualAllocEx( ), WriteProcessMemory( ), SetThreatContext( )** and **ResumeThreat( ) | NtResumeThread( ).**

- **Reflective Injection:** this technique is similar to PE Injection, but the malicious code avoid using **LoadLibrary( ) and CreateRemoteThread( )**, for example. There're many interesting derivations of this method and, one of them (also used on **Cobalt Strike**) is accomplished by the following APIs: **CreateFileMapping( ), Nt/MapViewOfFile( ), OpenProcess( ), memcpy( )** and **Nt/MapViewOfSection( )**. At end, code on remote process can be executed by calling **OpenProcess( ), CreateThread( ), NtQueueApcThread( ), CreateRemoteThread( )** or **RtlCreateUserThread( ).** It's interesting to note that a variant could use **VirtualQueryEx( )** and **ReadProcessMemory( )** too.

- **APC Injection:** this code injection technique allows a program to execute code in a specific thread by attaching to an APC queue. The injected code will be executed by the thread when it exits of alertable state (originated by calls such as **SleepEx( ), SignalObjectAndWait( ), MsgWaitForMultipleObjectsEx( ), WaitForMultipleObjectsEx( ),** or **WaitForSingleObjectEx( )**). Therefore, it's common to also see APIs such as **CreateToolhelp32Snapshot(), Process32First( ), Process32Next( ), Thread32First( ), Thread32Next( ), QueueUserAPC( )** and **KeInitializeAPC( )** involved into this technique.

- **Hollowing or Process Replacement:** this technique, in a nutshell, is used by the malware to "drain out" the entire content of a process and insert into it a malicious content. Some involved APIs are **CreateProcess( ), NtQueryProcessInformation( ), GetModuleHandle( ), Zw/NtUnmapViewOfSection( ), VirtualAllocEx( ), WriteProcessMemory( ), GetThreadContext( ), SetThreadContext ( )** and **ResumeThread( ).**

- **AtomBombing:** this technique is an a variant of the previous technique (APC injection) and works by splitting the malicious payload into separated strings, creating an Atom to each given string, copying them into a RW segment (using **GlobalGetAtomName( )** and **NtQueueApcThread( )**) and setting the context by using **NtSetContextThread( ).** Therefore, a list of further APIs are **OpenThread( ), GlobalAddAtom( ), GlobalGetAtomName( )** and **QueueUserAPC( ).**

- **Process Doppelgänging:** this technique could be handled as a kind of evolution of Process Hollowing. The key difference between this both techniques is that while Process Hollowing replaces the process's content (image) before it being resumed, Process Doppelgänging is able to replace the image before the process even being created by overwriting the target image with a malicious one before it being loaded. The key concept here is that NTFS operations are performed within transactions, so either all these operations inside a transactions are committed together or none of them are committed. In the meanwhile, the malicious image only exists and it's visible inside the transaction and it isn't visible to any other process. Therefore, the malicious image is loaded into memory and the malware drops the malicious payload from file system (by rollbacking the transaction) as the file never had existed previously. Some APIs are involved in this technique:

CreateTransaction( ), CreateFileTransaction( ), NtCreateSection, NtCreateProcessEx( ), NtQueryInformationProcess( ), NtCreateThreadEx( ) and **RollbackTransaction( ).**

- **Process Herpaderping:** this technique is similar to Process Doppelgänging, but there's a subtle difference in its procedure. Process Herpaderping is based on that fact that security defenses usually monitor process creation by registering a callback routine on the kernel side using **PsSetCreateProcessNotifyRoutineEx( )** or during driver's **DispatchCleanup routines (IRP_MJ_CLEANUP)**, which it is invoked after a thread being created. That's the key issue: if the an adversary create and map a process and, afterwards, this adversary is able to modify the file image and then create the thread, so security products are able to detect such a malicious payload. Nonetheless, this checking order can be comprised whether the adversary is able to create malicious binary on disk, open a handle to it, map it as an image section using **NtCreateSection function** (and including the **SEC_IMAGE flag**), create a process using the section handle (**NtCreateProcesEx()**), modify the file content to not sounds like malicious and create a thread (**NtCreateThreadEx()**) using this "good image". That the point: when the thread is created, the process callback is triggered and the content of the file (good one) on disk is checked, so security defenses believes that everything is fine because image on disk is not harmful, but the true malicious is on memory. In other words, security defenses could not be effective to detect such image on disk that is different from image on memory. Few APIs used for this technique: **CreateFile( ), NtCreateSection( ), NtCreateProcessEx( )** and **NtCreateThreadEx( ).**

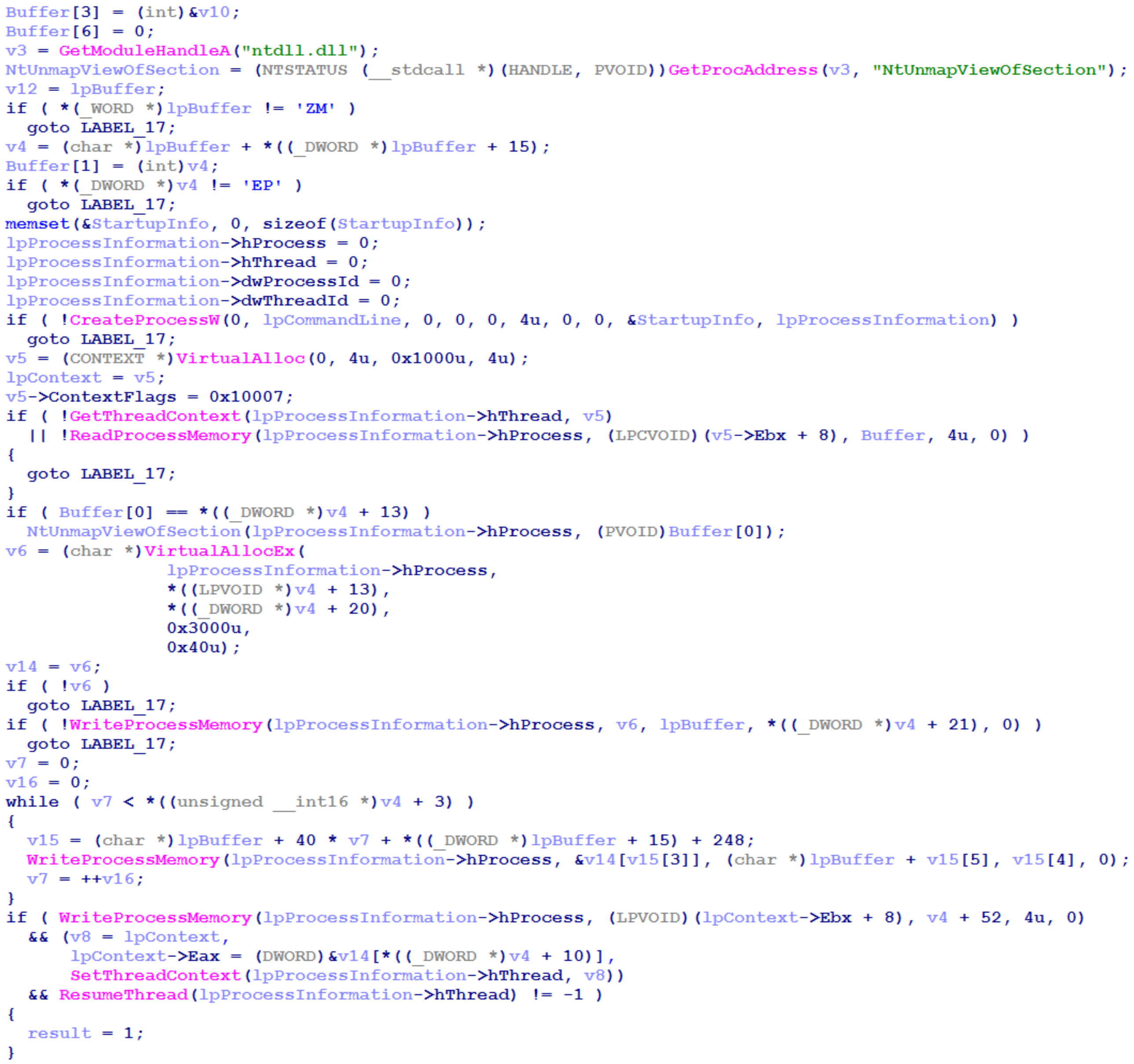
- **Hooking Injection:** to use this technique, we will see that functions involved with hooking activities such as **SetWindowsHookEx( )** and **PostThreadMessage( )** are used to inject a malicious DLL.

- **Extra Windows Memory Injection:** using this technique, malware threats injects code into the a process by using the *Extra Windows Memory (as known as EWM)*, whose size is up to 40 bytes and it's appended the instance of a class during the registration of windows classes. The trick is that the appended spaced is enough to store a pointer that might forward the execution to a malicious code. Some possible APIs involved to this technique are **FindWindowsA( ), GetWindowThreadProcessId( ), OpenProcess( ), VirtualAllocEx( ), WriteProcessMemory( ), SetWindowLongPtrA( )** and **SendNotify( ).**

- **Propagate Injection:** this technique has been used by malware threats such as RIG Exploit Kit and Smoke Loader to inject malicious code into explorer.exe process (medium integrity level) and other persistent ones, and it's based on the approach of enumerating (**EnumWindows( ) → EnumWindowsProc → EnumChildWindows( ) → EnumChildWindowsProc → EnumProps( ) → EnumPropsProc → GetProp**) windows implementing **SetWindowsSubclass( )** (this further information on https://docs.microsoft.com/en-us/windows/win32/api/commctrl/nf-commctrl-setwindowsubclass). As you could remember, this function install a windows subclass callback and, as you know, callbacks are interpreted as hooking methods in the security world. How does it works? Once subclassed windows are found (checking **UxSubclassInfo** and/or **CC32SubclassInfo**, which provide the subclass header), it's possible to preserve the old windows procedure, but we can also assign a new one to the window by updating **CallArray** field. When an event to the target process is sent then the new procedure is called and, afterwards, the old one is also called (keeping the previous and expected behavior). Therefore, a malware inserts a malicious payload (shellcode) into the memory and updates subclass procedure using **SetPropA( )**. When this new property is invoked (through a windows message) , the execution is forwarded to the payload. Some Windows APIs involved to this technique are **FindWindow( ), FindWindowEx( ), GetProp( ), GetWindowThreadProcessId( ), OpenProcess( ), ReadProcessMemory( ), VirtualAllocEx( ), WriteProcessMemory( ), SetProp( )** and **PostMessage( ).**

This short and quick review about code injection techniques will be useful to understand how malware try to keep undetected and also indirectly will help you to understand unpacking techniques.

A quite usual example of a code injection sequence from malware threats is shown below (a decompiled output from IDA Pro) and, certainly, you'll be able to identify the technique used through information presented previously in this section:



```
Buffer[3] = (int)&v10;
Buffer[6] = 0;
v3 = GetModuleHandleA("ntdll.dll");
NtUnmapViewOfSection = (NTSTATUS (__stdcall *)(HANDLE, PVOID))GetProcAddress(v3, "NtUnmapViewOfSection");
v12 = lpBuffer;
if ( *(_WORD *)lpBuffer != 'ZM' )
  goto LABEL_17;
v4 = (char *)lpBuffer + *((_DWORD *)lpBuffer + 15);
Buffer[1] = (int)v4;
if ( *(_DWORD *)v4 != 'EP' )
  goto LABEL_17;
memset(&StartupInfo, 0, sizeof(StartupInfo));
lpProcessInformation->hProcess = 0;
lpProcessInformation->hThread = 0;
lpProcessInformation->dwProcessId = 0;
lpProcessInformation->dwThreadId = 0;
if ( !CreateProcessW(0, lpCommandLine, 0, 0, 0, 4u, 0, 0, &StartupInfo, lpProcessInformation) )
  goto LABEL_17;
v5 = (CONTEXT *)VirtualAlloc(0, 4u, 0x1000u, 4u);
lpContext = v5;
v5->ContextFlags = 0x10007;
if ( !GetThreadContext(lpProcessInformation->hThread, v5)
  || !ReadProcessMemory(lpProcessInformation->hProcess, (LPCVOID)(v5->Ebx + 8), Buffer, 4u, 0) )
{
  goto LABEL_17;
}
if ( Buffer[0] == *((_DWORD *)v4 + 13) )
  NtUnmapViewOfSection(lpProcessInformation->hProcess, (PVOID)Buffer[0]);
v6 = (char *)VirtualAllocEx(
               lpProcessInformation->hProcess,
               *((LPVOID *)v4 + 13),
               *((_DWORD *)v4 + 20),
               0x3000u,
               0x40u);
v14 = v6;
if ( !v6 )
  goto LABEL_17;
if ( !WriteProcessMemory(lpProcessInformation->hProcess, v6, lpBuffer, *((_DWORD *)v4 + 21), 0) )
  goto LABEL_17;
v7 = 0;
v16 = 0;
while ( v7 < *((unsigned __int16 *)v4 + 3) )
{
  v15 = (char *)lpBuffer + 40 * v7 + *((_DWORD *)lpBuffer + 15) + 248;
  WriteProcessMemory(lpProcessInformation->hProcess, &v14[v15[3]], (char *)lpBuffer + v15[5], v15[4], 0);
  v7 = ++v16;
}
if ( WriteProcessMemory(lpProcessInformation->hProcess, (LPVOID)(lpContext->Ebx + 8), v4 + 52, 4u, 0)
  && (v8 = lpContext,
      lpContext->Eax = (DWORD)&v14[*((_DWORD *)v4 + 10)],
      SetThreadContext(lpProcessInformation->hThread, v8))
  && ResumeThread(lpProcessInformation->hThread) != -1 )
{
  result = 1;
}
```

[Figure 4]

7. Unpacking Methods

It's quite complicated to classify and, mainly, describe unpacking techniques, but in a general way there're few methods to unpack a malware sample such as using a debugger, an automated tool, a web service or even writing its own unpacking code to accomplish the task statically. The chosen methods depends on specific contexts and situations.

a. Debugger + breakpoint on specific functions

This is the most known method and consist on loading the malware into a debugger and setting up software breakpoints on well-known APIs, which most of them are related to memory management and manipulation, and looking for executables and/or shellcode to be extracted from the memory. Using x64dbg/x32dbg (**[ctrl]+g** or **bp <function>** on its CLI) is really simple to insert software breakpoints on the following APIs:

- **CreateProcessInternalW( )**
- **VirtualAlloc( )**
- **VirtualAllocEx( )**
- **VirtualProtect( ) | ZwProtectVirtualMemory( )**
- **WriteProcessMemory( ) | NtWriteProcessMemory( )**
- **ResumeThread( ) | NtResumeThread( )**
- **CryptDecrypt( ) | RtlDecompressBuffer( )**
- **NtCreateSection( ) + MapViewOfSection( ) | ZwMapViewOfSection( )**
- **UnmapViewOfSection( ) | ZwUnmapViewOfSection( )**
- **NtWriteVirtualMemory( )**
- **NtReadVirtualMemory( )**

During the unpacking procedure we might face some issues (for example, anti-debugging techniques being used by the malware) and other side effects. Therefore, some notes before and after unpacking could be useful:

- Set up breakpoints **after malware has reached its entry point** (after the system breakpoint).

- As mentioned previously, it's recommended to use an anti-debugging plugin and, in few cases, to ignore all exceptions from **0x00000000 to 0xFFFFFFFF range** (on x64dbg, go to **Options → Preferences → Exceptions** to include this range).

- Sometimes ignoring exceptions could be a bad idea because malware could be them to call the unpacking procedure. Additionally (and out of the context in this article) **there are threats that use interruptions and exceptions to call APIs**.

- Learning about all listed **APIs and their respective arguments by using MSDN** is a key knowledge to unpack malware threats successfully.

- If you're using **VirtualAlloc( )**, it's recommended to setup the **breakpoint on its exit point** (ret 10). Additionally, sometimes it is easier to follow the allocated content on dump by setting a **write memory breakpoint**.

- In some cases, the malware extracts its payload onto memory, but it **destroys the PE Header**, so you'll have **reconstruct the entire header,** though it's simple procedure using a hex editor like HxD.

- The extracted payload might be in mapped or unmapped format. If it's in mapped format, so probably the Import table is messed up and you need to fix them by **realigning sections headers** manually through PEBear (favorite method) or using a tool like **pe_unmapper**. You might need to **fix the base address and the entry point** whether it's zeroed.

- **To reconstruct a destroyed IAT it's recommended to use Scylla** (embedded on x64dbg). It will be necessary to enter the OEP and one of methods to find it is by looking for code transitions given by instructions such **as jmp eax, call eax, call [eax],** and so on.

- Few unpacked malware samples don't have any function in the IAT, so there're two possibilities: **either sections are misaligned (mapped version) or the unpacked malware resolves all its functions dynamically**.

- Using the **"g" hotkey on x64dbg** might be useful for visualizing the code in blocks and finding possible transitions to OEP.

- **Another good alternative to find OEP** is through code instrumentation like **PIN** (https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html).

- Tools like **tiny_tracer** (https://github.com/hasherezade/tiny_tracer) use PIN to perform instrumentation easier and can be used to learn about functions being called by the malware (quite useful for unpacking and learning about anti-analysis techniques) and also to find possible OEP.

- In many opportunities, the **unpacked code could be only the first stage** of a malware, so it's necessary to repeat steps to unpack the next stages.

- Few malware sample perform **self-overwriting**, so you could have to set a breakpoint on the **.text section** to detect the unpacked binary execution.

- Depending on the extracted binary (a shellcode, for example), **it might not be able to run out of a specific process context, so it'd be necessary to inject it into a running process** (for example, explorer.exe) to perform further analysis.

- How can you check whether the extracted malware might be the final one? There isn't a definitive answer and few indications might be found by looking for network functions from DLLs such as **WS2_32.dll** (Winsock) and **Wininet.dll**, **plain text strings**, **crypto functions** (mainly whether malware is an ransomware), and many other evidences. It's a good approach to **open up the extracted code on IDA Pro mainly after having re-aligned sections and/or reconstructed the IAT**.

b. Debugger + break on DLL loading

This an old and simple technique to unpack malware by stopping the debugger on each DLL loaded and examining the memory mapping for potentially extracted PE format files on the memory (pay attention: don't focus only on RWX segments because many malwares extracts its payload in RW regions and soon before transferring the execution context to the extracted executable they change the region's permissions to RWX by using **VirtualProtect( )**) . No doubts it can consume some time, but It continues being efficient in many cases. Common debuggers (**x64dbg, OllyDbg** and **Immunity Debugger**) have a configuration option to break on each DLL loading. On x64dbg this option is in **Options → Preferences → Events** and mark **DLL load.** On **OllyDbg** you can go to **Options → Debugging Options → Events** and mark "**Break on new module (DLL)**".

c. Automated method

A malware analyst can use tools to automate the unpacking procedure. **Aleksandra Doniec (Hasherezade)** has provided excellent tools to attend this objective:

- **hollows-hunter:** https://github.com/hasherezade/hollows_hunter/releases
- **pe-sieve:** https://github.com/hasherezade/pe-sieve/releases
- **mal_unpack:** https://github.com/hasherezade/mal_unpack/releases

Her tools has a similar approach to each other, so you should run the malware in an isolated virtual machine and execute the appropriate command, which I show some syntax examples below that can be used for a quick approach, though all of these tools contain useful options and it's worth to check them:

- **hollow_hunter.exe /pname <filename> /loop /imp**
- **mal_unpack.exe /exe <filename> /timeout <timeout: ms>**
- **pe-sieve64.exe /pid <process ID>**
- **pe-sieve64.exe /pid <process ID> /dmode 3 /imp 3**

The unpacked binaries with some additional information are saved into a directory created by the tool.

d. Process Hacker

Another trivial (and limited) way to extract binaries from memory is through **Process Hacker** by double-clicking on the running process, going to **"Memory" tab**, looking for **interesting regions/base addresses (RWX)**, double-clicking it and pressing "Save" button. Of course, it's easier finding the malicious binary/payload in case of self-injection. In case of remote injection you'll need to reverse the malware to understand the target process to be inject or make an "educated guess" and look for the injected code on well-known targets like explorer.exe or svchost.exe, for example. Once again, it's a limited and simple approach, but sometimes can save time.

e. Using an public/paid Internet service

You can use an Internet service as the amazing **Unpacme** (https://www.unpac.me/#/), which offers an automated unpacking service. There're a free and public plan (10 submissions per month) and other paid plans that are quite interesting for researchers and companies. Furthermore, it offers an API set to interface your customized application with the Unpacme service (https://api.unpac.me/).

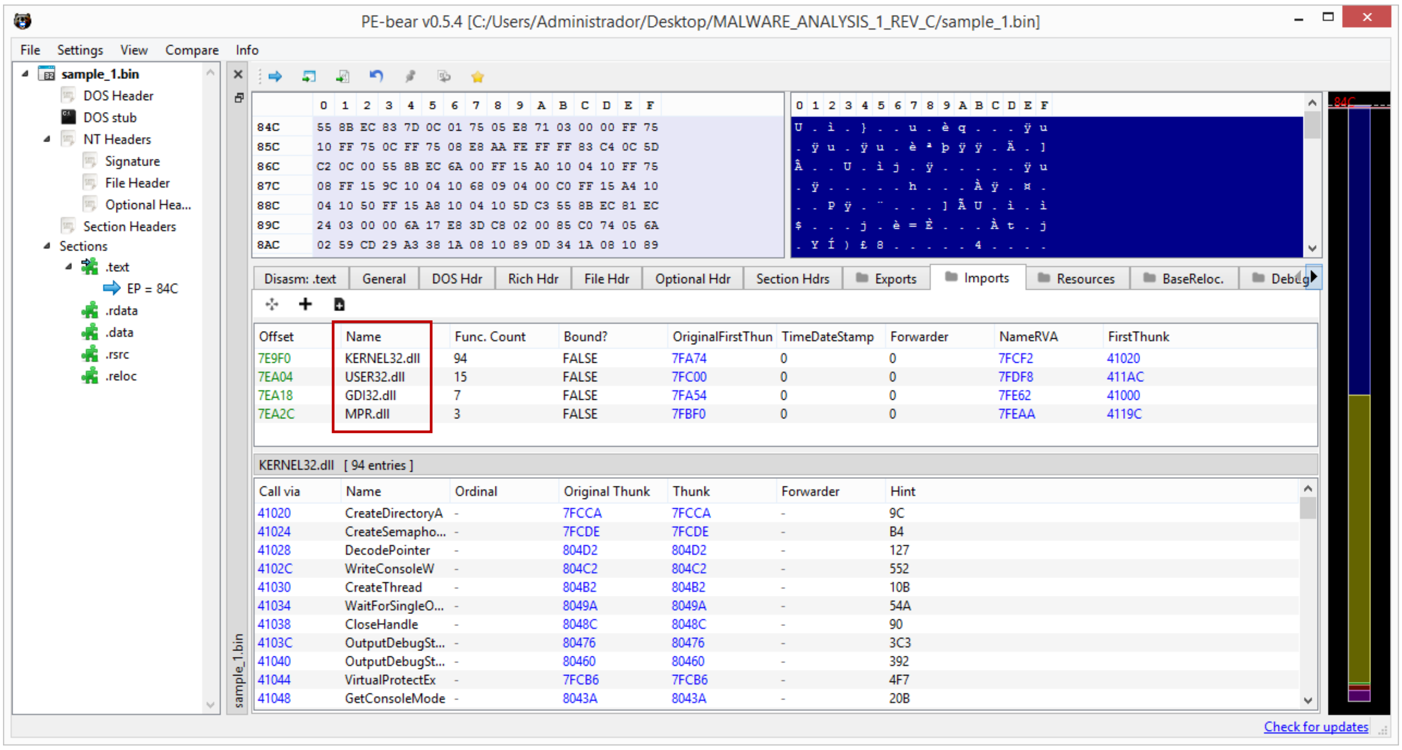
f. Writing an unpacker code

Although this approach sounds being time consuming, it's quite usual **writing Python code** to accomplish unpacking mainly in shellcode cases or while handling a case which a malware threads use several anti-vm and anti-debugging techniques. In addition, we have an advantage to automate the unpacking process while handling similar malware cases.

8. Unpacking the binary

Now we made a fast review about fundamentals of malware analysis, let's start our analysis. As I've already explained previously, I picked up this sample because it's quite simple and, in my opinion, it will be useful to start our series of articles. Remember this sample has the following SHA256 hash: **8ff43b6ddf6243bd5ee073f9987920fa223809f589d151d7e438fd8cc08ce292.**

Checking it using **PEBear** could be useful to collect first valuable information from malware:



```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 1 2 3 4 5 6 7 8 9 A B C D E F
84C  55 8B EC 83 7D 0C 01 75 05 E8 71 03 00 00 FF 75   U . ì . } . . u . è q . . . ÿ u
85C  10 FF 75 0C FF 75 08 E8 AA FE FF FF 83 C4 0C 5D   . ÿ u . ÿ u . è ª þ ÿ ÿ . Ä . ]
86C  C2 0C 00 55 8B EC 6A 00 FF 15 A0 10 04 10 FF 75   Â . . U . ì j . ÿ . . . . . ÿ u
87C  08 FF 15 9C 10 04 10 68 09 04 00 C0 FF 15 A4 10   . ÿ . . . . . h . . . À ÿ . ¤ .
88C  04 10 50 FF 15 A8 10 04 10 5D C3 55 8B EC 81 EC   . . P ÿ . ¨ . . . ] Ã U . ì . ì
89C  24 03 00 00 6A 17 E8 3D C8 02 00 85 C0 74 05 6A   $ . . . j . è = È . . . À t . j
8AC  02 59 CD 29 A3 38 1A 08 10 89 0D 34 1A 08 10 89   . Y Í ) £ 8 . . . . . 4 . . . .
```

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
7E9F0	KERNEL32.dll	94	FALSE	7FA74	0	0	7FCF2	41020
7EA04	USER32.dll	15	FALSE	7FC00	0	0	7FDF8	411AC
7EA18	GDI32.dll	7	FALSE	7FA54	0	0	7FE62	41000
7EA2C	MPR.dll	3	FALSE	7FBF0	0	0	7FEAA	4119C

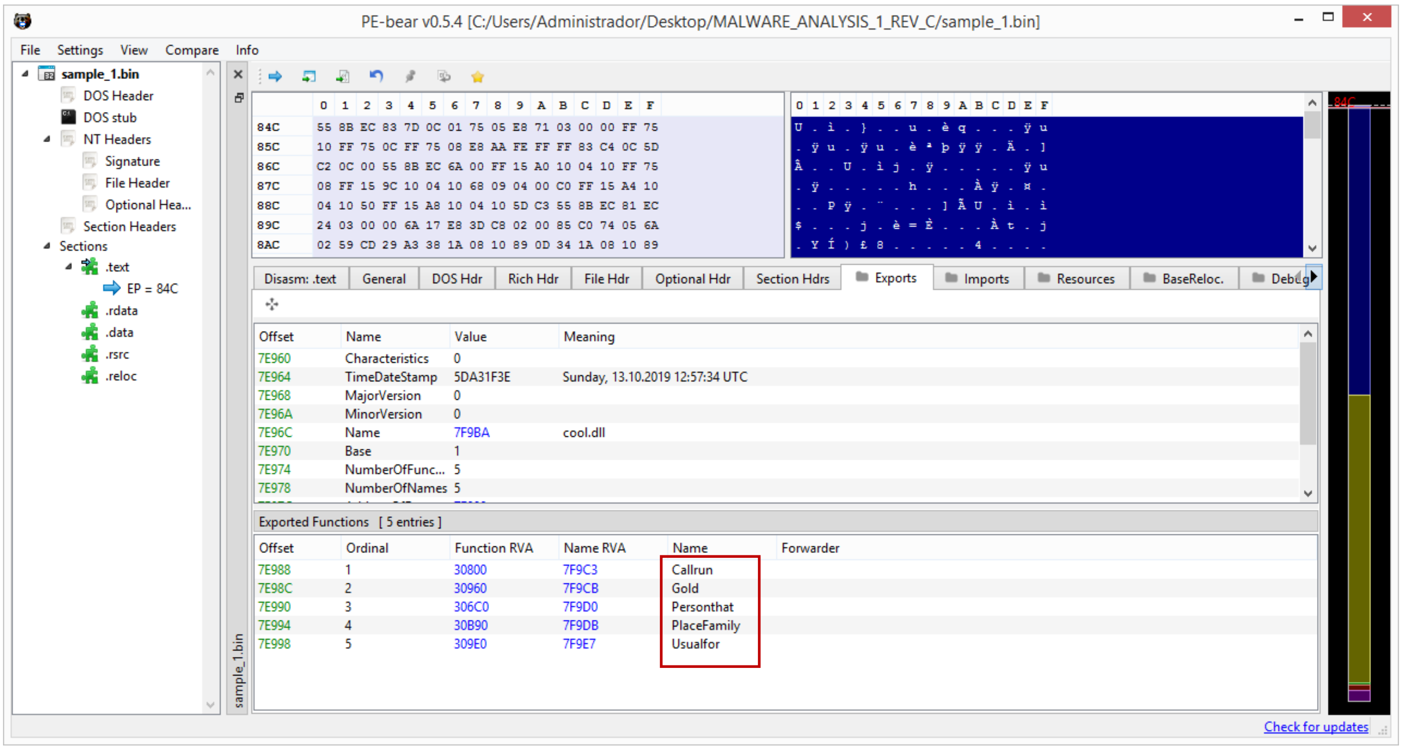
KERNEL32.dll [94 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
41020	CreateDirectoryA	-	7FCCA	7FCCA	-	9C
41024	CreateSemapho...	-	7FCDE	7FCDE	-	B4
41028	DecodePointer	-	804D2	804D2	-	127
4102C	WriteConsoleW	-	804C2	804C2	-	552
41030	CreateThread	-	804B2	804B2	-	10B
41034	WaitForSingleO...	-	8049A	8049A	-	54A
41038	CloseHandle	-	8048C	8048C	-	90
4103C	OutputDebugSt...	-	80476	80476	-	3C3
41040	OutputDebugSt...	-	80460	80460	-	392
41044	VirtualProtectEx	-	7FCB6	7FCB6	-	4F7
41048	GetConsoleMode	-	8043A	8043A	-	20B

[Figure 5]

As we can see**, there isn't any DLL (and functions) at IAT directly related to network communication, crypto or something really interesting**. Therefore, it's a first indication that our sample might be packed.

As this malware sample is a DLL, so we can learn about its **exported functions** because we'll use them to run the malware on the x64dbg/x32dbg:



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0 1 2 3 4 5 6 7 8 9 A B C D E F
84C   55 8B EC 83 7D 0C 01 75 05 E8 71 03 00 00 FF 75   U . ì . } . . u . è q . . . ÿ u
85C   10 FF 75 0C FF 75 08 E8 AA FE FF FF 83 C4 0C 5D   . ÿ u . ÿ u . è ª þ ÿ ÿ . Ä . ]
86C   C2 0C 00 55 8B EC 6A 00 FF 15 A0 10 04 10 FF 75   Â . . U . ì j . ÿ . . . . . ÿ u
87C   08 FF 15 9C 10 04 10 68 09 04 00 C0 FF 15 A4 10   . ÿ . . . . . h . . . À ÿ . ¤ .
88C   04 10 50 FF 15 A8 10 04 10 5D C3 55 8B EC 81 EC   . . P ÿ . ¨ . . . ] Ã U . ì . ì
89C   24 03 00 00 6A 17 E8 3D C8 02 00 85 C0 74 05 6A   $ . . . j . è = È . . . À t . j
8AC   02 59 CD 29 A3 38 1A 08 10 89 0D 34 1A 08 10 89   . Y Í ) £ 8 . . . . . . 4 . . .
```

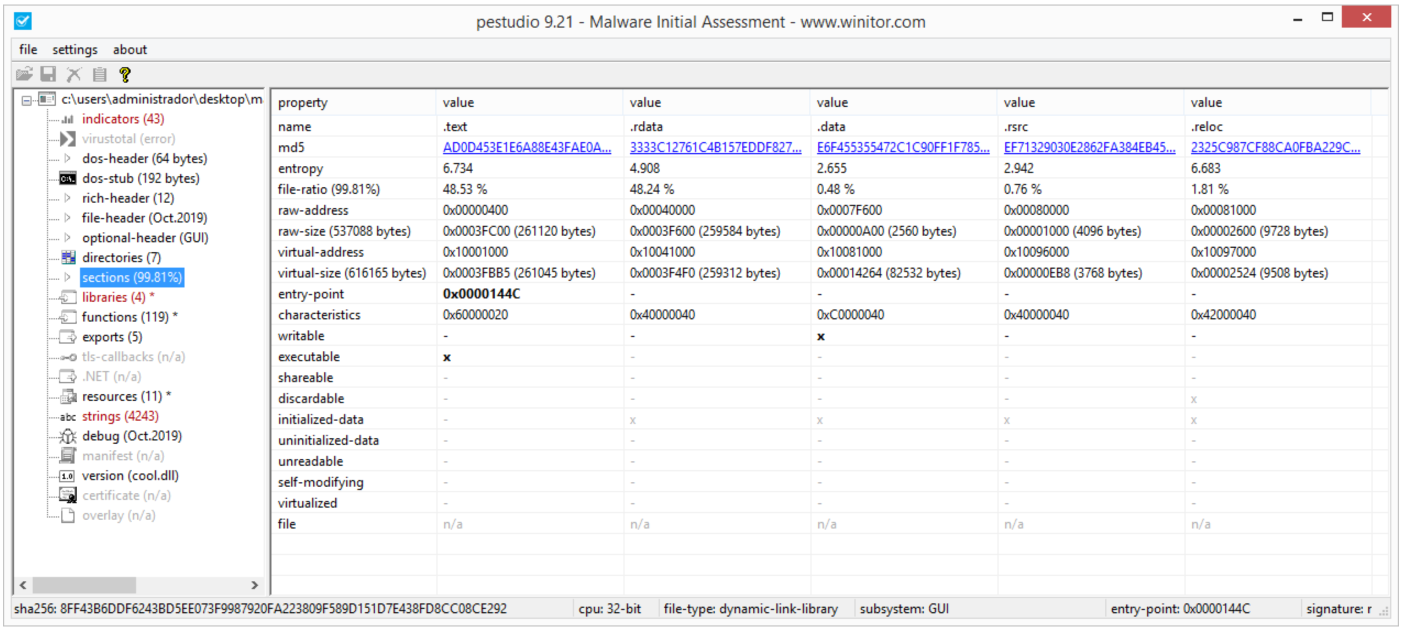
Offset	Name	Value	Meaning
7E960	Characteristics	0	
7E964	TimeDateStamp	5DA31F3E	Sunday, 13.10.2019 12:57:34 UTC
7E968	MajorVersion	0	
7E96A	MinorVersion	0	
7E96C	Name	7F9BA	cool.dll
7E970	Base	1	
7E974	NumberOfFunc...	5	
7E978	NumberOfNames	5	

Exported Functions [5 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
7E988	1	30800	7F9C3	Callrun	
7E98C	2	30960	7F9CB	Gold	
7E990	3	306C0	7F9D0	Personthat	
7E994	4	30B90	7F9DB	PlaceFamily	
7E998	5	309E0	7F9E7	Usualfor	

[Figure 6]

Using **pestudio tool** we are able to figure out a meaning difference between raw size and virtual size on .data section, which is also marked as "writable". It's an additional indication that our sample might be packed, as we expected.



property	value	value	value	value	value
name	.text	.rdata	.data	.rsrc	.reloc
md5	AD0D453E1E6A88E43FAE0A...	3333C12761C4B157EDDF827...	E6F455355472C1C90FF1F785...	EF71329030E2862FA384EB45...	2325C987CF88CA0FBA229C...
entropy	6.734	4.908	2.655	2.942	6.683
file-ratio (99.81%)	48.53 %	48.24 %	0.48 %	0.76 %	1.81 %
raw-address	0x00000400	0x00040000	0x0007F600	0x00080000	0x00081000
raw-size (537088 bytes)	0x0003FC00 (261120 bytes)	0x0003F600 (259584 bytes)	0x00000A00 (2560 bytes)	0x00001000 (4096 bytes)	0x00002600 (9728 bytes)
virtual-address	0x10001000	0x10041000	0x10081000	0x10096000	0x10097000
virtual-size (616165 bytes)	0x0003FBB5 (261045 bytes)	0x0003F4F0 (259312 bytes)	0x00014264 (82532 bytes)	0x00000EB8 (3768 bytes)	0x00002524 (9508 bytes)
entry-point	**0x0000144C**	-	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040	0x42000040
writable	-	-	**x**	-	-
executable	**x**	-	-	-	-
shareable	-	-	-	-	-
discardable	-	-	-	-	x
initialized-data	-	x	x	x	x
uninitialized-data	-	-	-	-	-
unreadable	-	-	-	-	-
self-modifying	-	-	-	-	-
virtualized	-	-	-	-	-
file	n/a	n/a	n/a	n/a	n/a

sha256: 8FF43B6DDF6243BD5EE073F9987920FA223809F589D151D7E438FD8CC08CE292 cpu: 32-bit file-type: dynamic-link-library subsystem: GUI entry-point: 0x0000144C signature: r

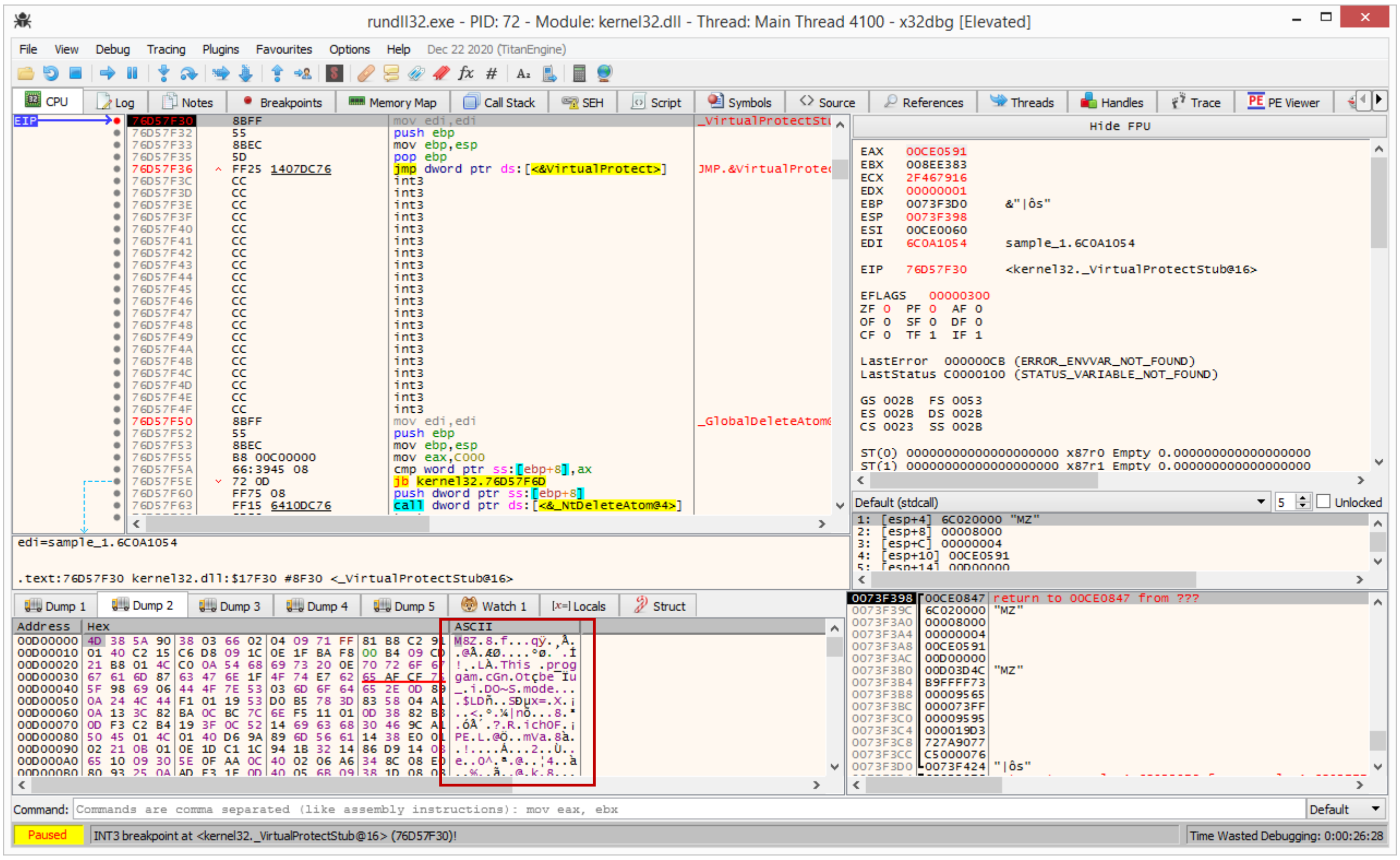
[Figure 7]

Our malware sample has five exported functions and, without analyzing it on IDA Pro, it's hard to guess what they really do. However, we could try the first one named "**Callrun**" that, apparently, it's a good bet. Therefore, using **x32dbg** (this binary is 32-bit), we can run it by open the **rundll32.exe** and changing the command line (**File → Change Command Line**) to **"C:\Windows\SysWOW64\rundll32.exe" C:\Users\Administrador\Desktop\sample_1.bin,#1**.

After changing it, restart/reload the debugging session and set up the following breakpoints on few classical functions (you can do it using **CTRL+G** or even the **x32dbg** command line interface) **after you have reached the entry point**:

- **VirtualAlloc** (on its exit point)
- **VirtualProtect**
- **ResumeThread**

Run **(F9)** and, after hitting the first breakpoint (on **VirtualAlloc( )**), right click on **EAX** and pick **"Follow In Dump".** The second hit will be in the same region and on the third hit you can right-click on the EAX and choose "Follow in Dump 2". If you wanted (not necessary here), you might go to the **"Dump 2" tab**, select the first four hex bytes, **right click → Breakpoint → Memory, Write → Singleshot** (it could be useful in case where malware take a long time to write on that region). If everything goes right (I hope) you'll see an image similar to the following one:


rundll32.exe - PID: 72 - Module: kernel32.dll - Thread: Main Thread 4100 - x32dbg [Elevated]
File View Debug Tracing Plugins Favourites Options Help Dec 22 2020 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace PE Viewer
Hide FPU
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct
Default (stdcall) 5 Unlocked
Default

```
EIP → 76D57F30  8BFF             mov edi,edi                           _VirtualProtectStu
      76D57F32  55               push ebp
      76D57F33  8BEC             mov ebp,esp
      76D57F35  5D               pop ebp
      76D57F36  ^ FF25 1407DC76  jmp dword ptr ds:[<&VirtualProtect>]  JMP.&VirtualProte
      76D57F3C  CC               int3
      76D57F3D  CC               int3
      76D57F3E  CC               int3
      76D57F3F  CC               int3
      76D57F40  CC               int3
      76D57F41  CC               int3
      76D57F42  CC               int3
      76D57F43  CC               int3
      76D57F44  CC               int3
      76D57F45  CC               int3
      76D57F46  CC               int3
      76D57F47  CC               int3
      76D57F48  CC               int3
      76D57F49  CC               int3
      76D57F4A  CC               int3
      76D57F4B  CC               int3
      76D57F4C  CC               int3
      76D57F4D  CC               int3
      76D57F4E  CC               int3
      76D57F4F  CC               int3
      76D57F50  8BFF             mov edi,edi                           _GlobalDeleteAtom@
      76D57F52  55               push ebp
      76D57F53  8BEC             mov ebp,esp
      76D57F55  B8 00C00000      mov eax,C000
      76D57F5A  66:3945 08       cmp word ptr ss:[ebp+8],ax
      76D57F5E  v 72 0D          jb kernel32.76D57F6D
      76D57F60  FF75 08          push dword ptr ss:[ebp+8]
      76D57F63  FF15 6410DC76    call dword ptr ds:[<&_NtDeleteAtom@4>]
```

```
edi=sample_1.6C0A1054

.text:76D57F30 kernel32.dll:$17F30 #8F30 <_VirtualProtectStub@16>
```

```
EAX    00CE0591
EBX    008EE383
ECX    2F467916
EDX    00000001
EBP    0073F3D0     &"|ôs"
ESP    0073F398
ESI    00CE0060
EDI    6C0A1054     sample_1.6C0A1054

EIP    76D57F30     <kernel32._VirtualProtectStub@16>

EFLAGS   00000300
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  000000CB (ERROR_ENVVAR_NOT_FOUND)
LastStatus C0000100 (STATUS_VARIABLE_NOT_FOUND)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

ST(0) 0000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 0000000000000000000 x87r1 Empty 0.000000000000000000
```

```
1: [esp+4] 6C020000 "MZ"
2: [esp+8] 00008000
3: [esp+C] 00000004
4: [esp+10] 00CE0591
5: [esp+14] 00D00000
```

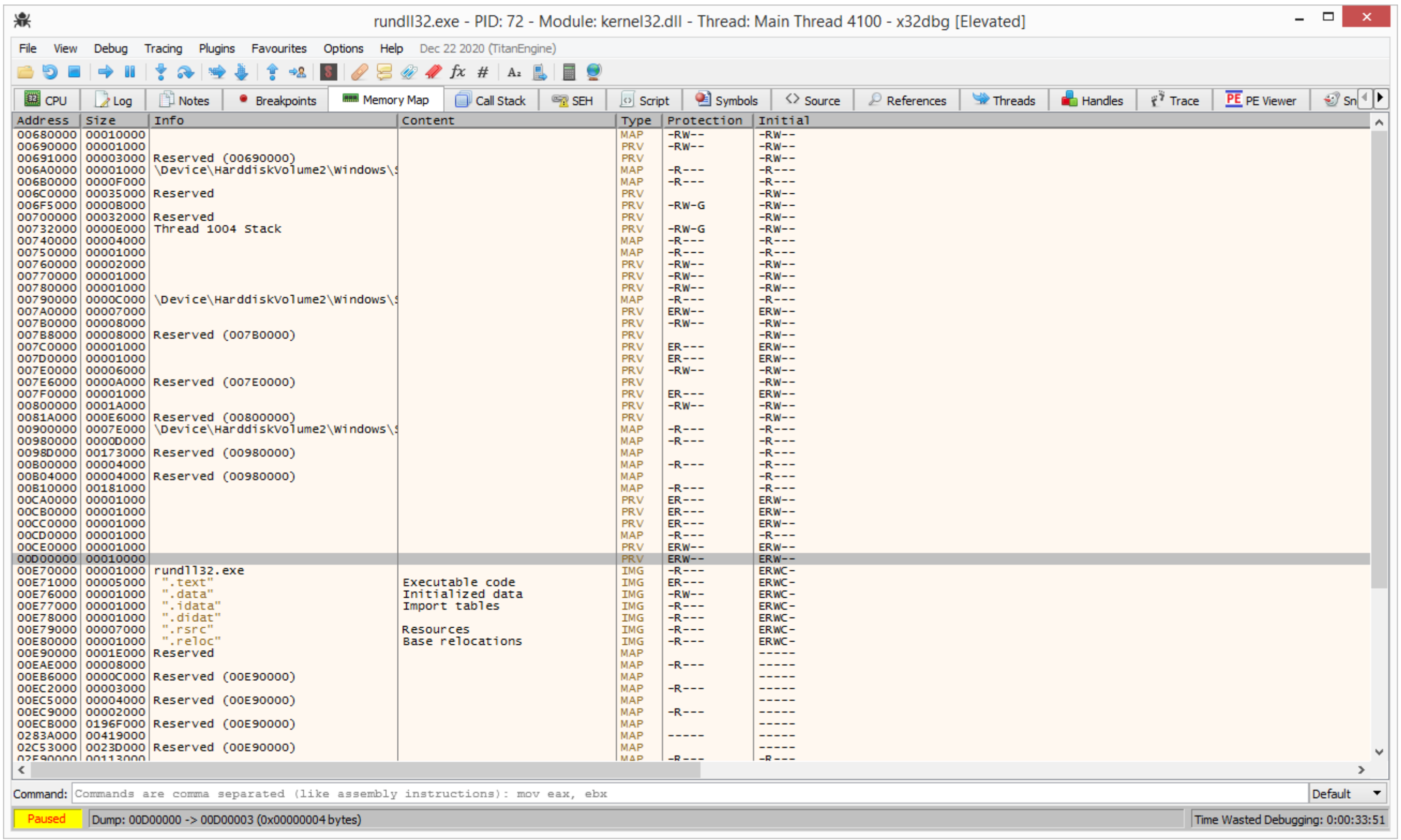
```
Address   Hex                                               ASCII
00D00000  4D 38 5A 90 38 03 66 02 04 09 71 FF 81 B8 C2 91  M8Z.8.f...qÿ.‚Â.
00D00010  01 40 C2 15 C6 D8 09 1C 0E 1F BA F8 00 B4 09 CD  .@Â.ÆØ....°ø. .Í
00D00020  21 B8 01 4C C0 0A 54 68 69 73 20 0E 70 72 6F 67  !..LÀ.This .prog
00D00030  67 61 6D 87 63 47 6E 1F 4F 74 E7 62 65 AF CF 75  gam.cGn.Otçbe¯Ïu
00D00040  5F 98 69 06 44 4F 7E 53 03 6D 6F 64 65 2E 0D 89  _.i.DO~S.mode...
00D00050  0A 24 4C 44 F1 01 19 53 D0 B5 78 3D 83 58 04 A1  .$LDñ..SÐµx=.X.¡
00D00060  0A 13 3C 82 BA 0C BC 7C 6E F5 11 01 0D 38 82 B8  ..<.º.¼|nõ...8.¸
00D00070  0D F3 C2 B4 19 3F 0C 52 14 69 63 68 30 46 9C A1  .óÂ´.?.R.ichOF.¡
00D00080  50 45 01 4C 01 40 D6 9A 89 6D 56 61 14 38 E0 01  PE.L.@Ö..mVa.8à.
00D00090  02 21 0B 01 0E 1D C1 1C 94 1B 32 14 86 D9 14 08  .!....Á...2..Ù..
00D000A0  65 10 09 30 5E 0F AA 0C 40 02 06 A6 34 8C 08 E0  e..0^.ª.@..¦4..à
00D000B0  80 93 25 0A AD E3 1F 0D 40 05 6B 09 38 1D 08 08  ..%..ã..@.k.8...
```

```
0073F398  00CE0847  return to 00CE0847 from ???
0073F39C  6C020000  "MZ"
0073F3A0  00008000
0073F3A4  00000004
0073F3A8  00CE0591
0073F3AC  00D00000
0073F3B0  00D03D4C  "MZ"
0073F3B4  B9FFFF73
0073F3B8  00009565
0073F3BC  000073FF
0073F3C0  00009595
0073F3C4  000019D3
0073F3C8  727A9077
0073F3CC  C5000076
0073F3D0  0073F424  "|ôs"
```

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused  INT3 breakpoint at <kernel32._VirtualProtectStub@16> (76D57F30)!   Time Wasted Debugging: 0:00:26:28
```

[Figure 8]

As you can see, first characters on ASCII representation are “**M8Z**”, which **suggests it’s using aPLib compression**. However, if you dump this region, you’ll find out the unpacked binary at the same region. Therefore, right-click on bytes from the **“Dump 2” tab** and choose **“Follow in Memory Map”** option:



rundll32.exe - PID: 72 - Module: kernel32.dll - Thread: Main Thread 4100 - x32dbg [Elevated]

File View Debug Tracing Plugins Favourites Options Help Dec 22 2020 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace PE Viewer Sn

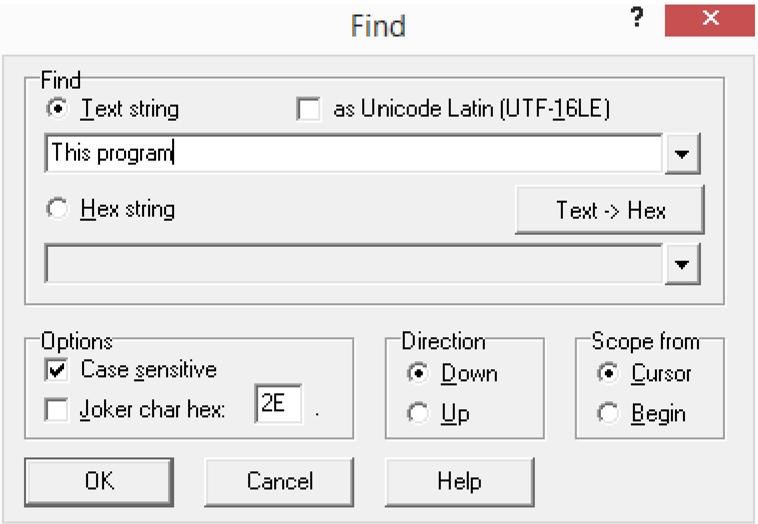
Address	Size	Info	Content	Type	Protection	Initial
00680000	00010000			MAP	-RW--	-RW--
00690000	00001000			PRV	-RW--	-RW--
00691000	00003000	Reserved (00690000)		PRV		-RW--
006A0000	00001000	\Device\HarddiskVolume2\Windows\S		MAP	-R---	-R---
006B0000	0000F000			MAP	-R---	-R---
006C0000	00035000	Reserved		PRV		-RW--
006F5000	0000B000			PRV	-RW-G	-RW--
00700000	00032000	Reserved		PRV		-RW--
00732000	0000E000	Thread 1004 Stack		PRV	-RW-G	-RW--
00740000	00004000			MAP	-R---	-R---
00750000	00001000			MAP	-R---	-R---
00760000	00002000			PRV	-RW--	-RW--
00770000	00001000			PRV	-RW--	-RW--
00780000	00001000			PRV	-RW--	-RW--
00790000	0000C000	\Device\HarddiskVolume2\Windows\S		MAP	-R---	-R---
007A0000	00007000			PRV	ERW--	ERW--
007B0000	00008000			PRV	-RW--	-RW--
007B8000	00008000	Reserved (007B0000)		PRV		-RW--
007C0000	00001000			PRV	ER---	ERW--
007D0000	00001000			PRV	ER---	ERW--
007E0000	00006000			PRV	-RW--	-RW--
007E6000	0000A000	Reserved (007E0000)		PRV		-RW--
007F0000	00001000			PRV	ER---	ERW--
00800000	0001A000			PRV	-RW--	-RW--
0081A000	000E6000	Reserved (00800000)		PRV		-RW--
00900000	0007E000	\Device\HarddiskVolume2\Windows\S		MAP	-R---	-R---
00980000	0000D000			MAP	-R---	-R---
0098D000	00173000	Reserved (00980000)		MAP		-R---
00B00000	00004000			MAP	-R---	-R---
00B04000	00004000	Reserved (00980000)		MAP		-R---
00B10000	00181000			MAP	-R---	-R---
00CA0000	00001000			PRV	ER---	ERW--
00CB0000	00001000			PRV	ER---	ERW--
00CC0000	00001000			PRV	ER---	ERW--
00CD0000	00001000			MAP	-R---	-R---
00CE0000	00001000			PRV	ERW--	ERW--
00D00000	00010000			PRV	ERW--	ERW--
00E70000	00001000	rundll32.exe		IMG	-R---	ERWC-
00E71000	00005000	".text"	Executable code	IMG	ER---	ERWC-
00E76000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
00E77000	00001000	".idata"	Import tables	IMG	-R---	ERWC-
00E78000	00001000	".didat"		IMG	-R---	ERWC-
00E79000	00007000	".rsrc"	Resources	IMG	-R---	ERWC-
00E80000	00001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00E90000	0001E000	Reserved		MAP		-----
00EAE000	00008000			MAP	-R---	-----
00EB6000	0000C000	Reserved (00E90000)		MAP		-----
00EC2000	00003000			MAP	-R---	-----
00EC5000	00004000	Reserved (00E90000)		MAP		-----
00EC9000	00002000			MAP	-R---	-----
00ECB000	0196F000	Reserved (00E90000)		MAP		-----
0283A000	00419000			MAP	-----	-----
02C53000	0023D000	Reserved (00E90000)		MAP		-----
02E90000	00113000			MAP	-R---	-R---

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

Paused Dump: 00D00000 -> 00D00003 (0x00000004 bytes) Time Wasted Debugging: 0:00:33:51

[Figure 9]

On the gray-highlighted base address, right click it and choose **“Dump Memory to File”** to save the memory region on the Desktop. For now, keep the debugger opened (you could need to fix an eventual destroyed IAT), and open the dumped file on the **XVI Editor**. Press **CTRL-F** and look for the string “**This program**” (pay attention: this search works in this case because the **PE Header wasn’t destroyed**):



Find

Find

● Text string □ as Unicode Latin (UTF-16LE)

This program

○ Hex string

Options

☑ Case sensitive

□ Joker char hex: 2E .

Direction

● Down

○ Up

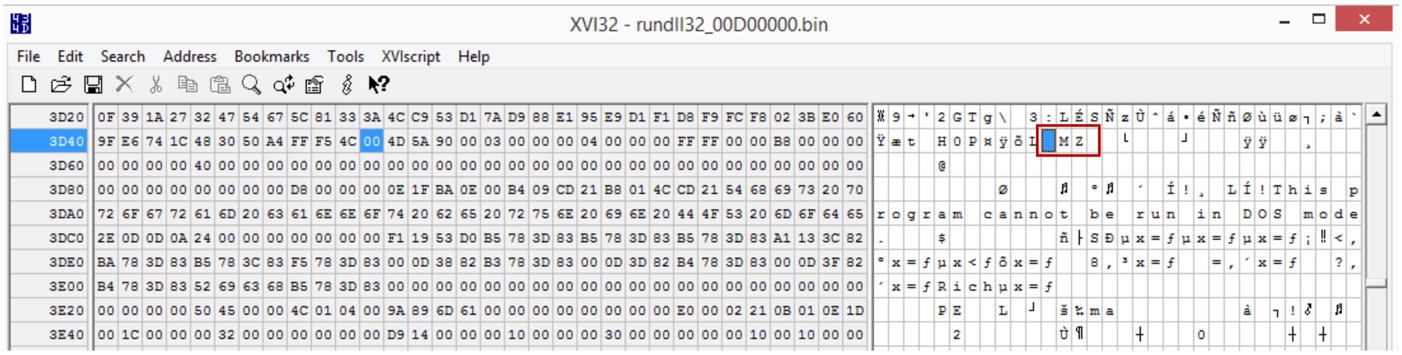
Scope from

● Cursor

○ Begin

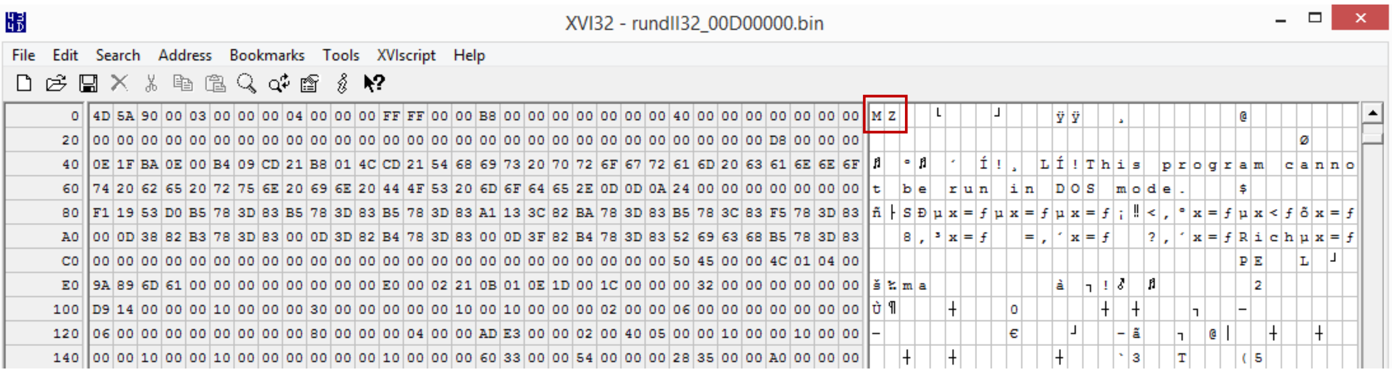
[Figure 10]

Once you found it, you should look for "MZ" characters on two or three lines before. Now, put the cursor at byte before the "MZ" mark, **go to Edit → Delete to cursor** as shown in the next couple of figures:



```
3D20  0F 39 1A 27 32 47 54 67 5C 81 33 3A 4C C9 53 D1 7A D9 88 E1 95 E9 D1 F1 D8 F9 FC F8 02 3B E0 60
3D40  9F E6 74 1C 48 30 50 A4 FF F5 4C 00 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00
3D60  00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3D80  00 00 00 00 00 00 00 00 D8 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70
3DA0  72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65
3DC0  2E 0D 0D 0A 24 00 00 00 00 00 00 00 F1 19 53 D0 B5 78 3D 83 B5 78 3D 83 B5 78 3D 83 A1 13 3C 82
3DE0  BA 78 3D 83 B5 78 3C 83 F5 78 3D 83 00 0D 38 82 B3 78 3D 83 00 0D 3D 82 B4 78 3D 83 00 0D 3F 82
3E00  B4 78 3D 83 52 69 63 68 B5 78 3D 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3E20  00 00 00 00 50 45 00 00 4C 01 04 00 9A 89 6D 61 00 00 00 00 00 00 00 00 E0 00 02 21 0B 01 0E 1D
3E40  00 1C 00 00 00 32 00 00 00 00 00 00 D9 14 00 00 00 10 00 00 00 30 00 00 00 00 00 10 00 10 00 00
```

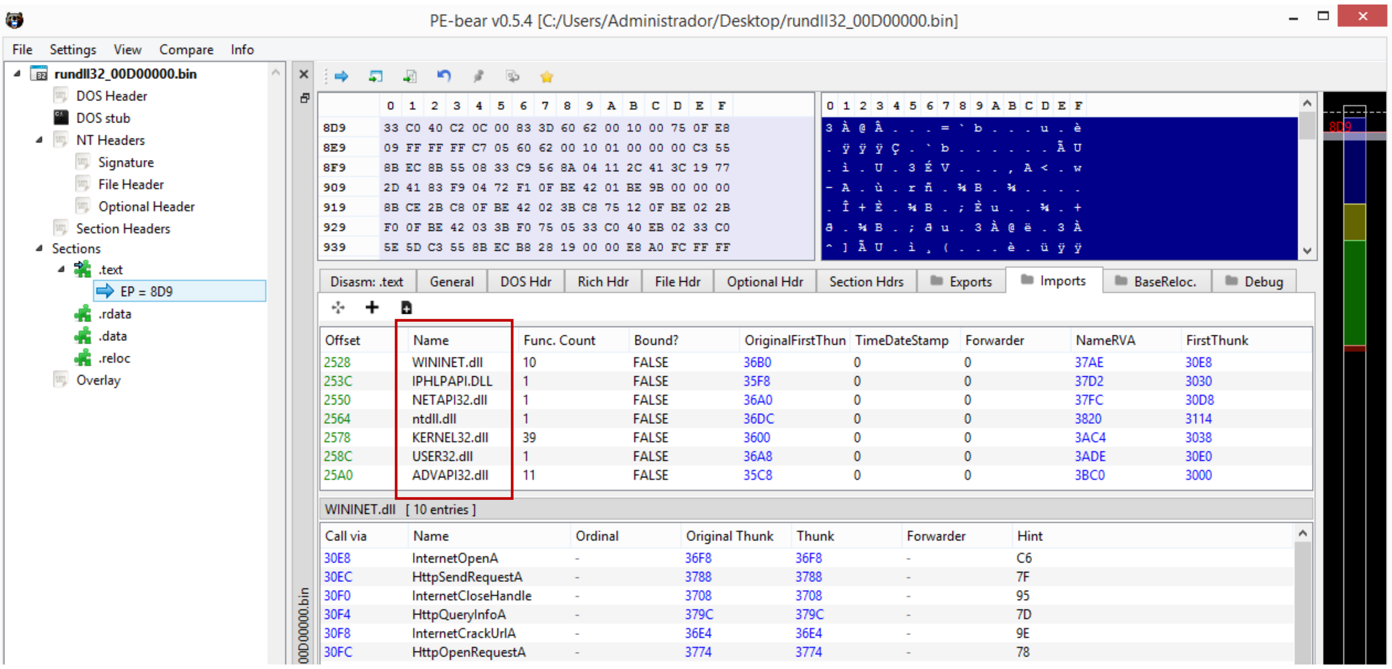
[Figure 11]



```
   0  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
  20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00
  40  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
  60  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
  80  F1 19 53 D0 B5 78 3D 83 B5 78 3D 83 B5 78 3D 83 A1 13 3C 82 BA 78 3D 83 B5 78 3C 83 F5 78 3D 83
  A0  00 0D 38 82 B3 78 3D 83 00 0D 3D 82 B4 78 3D 83 00 0D 3F 82 B4 78 3D 83 52 69 63 68 B5 78 3D 83
  C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00
  E0  9A 89 6D 61 00 00 00 00 00 00 00 00 E0 00 02 21 0B 01 0E 1D 00 1C 00 00 00 32 00 00 00 00 00 00
 100  D9 14 00 00 00 10 00 00 00 30 00 00 00 00 00 10 00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00
 120  06 00 00 00 00 00 00 00 00 80 00 00 00 04 00 00 AD E3 00 00 02 00 40 05 00 00 10 00 00 10 00 00
 140  00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 60 33 00 00 54 00 00 00 28 35 00 00 A0 00 00 00
```

[Figure 12]

Save it, open it on **PEBear** and go to **Imports tab** as shown below:



```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
8D9  33 C0 40 C2 0C 00 83 3D 60 62 00 10 00 75 0F E8
8E9  09 FF FF FF C7 05 60 62 00 10 01 00 00 00 C3 55
8F9  8B EC 8B 55 08 33 C9 56 8A 04 11 2C 41 3C 19 77
909  2D 41 83 F9 04 72 F1 0F BE 42 01 BE 9B 00 00 00
919  8B CE 2B C8 0F BE 42 02 3B C8 75 12 0F BE 02 2B
929  F0 0F BE 42 03 3B F0 75 05 33 C0 40 EB 02 33 C0
939  5E 5D C3 55 8B EC B8 28 19 00 00 E8 A0 FC FF FF
```

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
2528	WININET.dll	10	FALSE	36B0	0	0	37AE	30E8
253C	IPHLPAPI.DLL	1	FALSE	35F8	0	0	37D2	3030
2550	NETAPI32.dll	1	FALSE	36A0	0	0	37FC	30D8
2564	ntdll.dll	1	FALSE	36DC	0	0	3820	3114
2578	KERNEL32.dll	39	FALSE	3600	0	0	3AC4	3038
258C	USER32.dll	1	FALSE	36A8	0	0	3ADE	30E0
25A0	ADVAPI32.dll	11	FALSE	35C8	0	0	3BC0	3000

WININET.dll [10 entries]

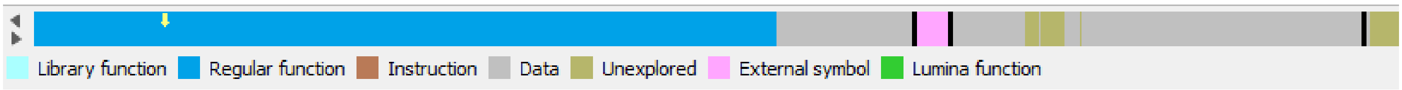
Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
30E8	InternetOpenA	-	36F8	36F8	-	C6
30EC	HttpSendRequestA	-	3788	3788	-	7F
30F0	InternetCloseHandle	-	3708	3708	-	95
30F4	HttpQueryInfoA	-	379C	379C	-	7D
30F8	InternetCrackUrlA	-	36E4	36E4	-	9E
30FC	HttpOpenRequestA	-	3774	3774	-	78

[Figure 13]

As you can confirm, the IAT is perfect, there is one DLL related to network communication (**WININET.dll**) and you also can see a slightly different **Entry Point (EP).** Therefore, it seems being our first stage unpacked (you should always assume that might have further packed stages and to repeat the same analysis unless you have an external information from other source or report). At this point, you can close the x32dbg because we won't need it anymore.

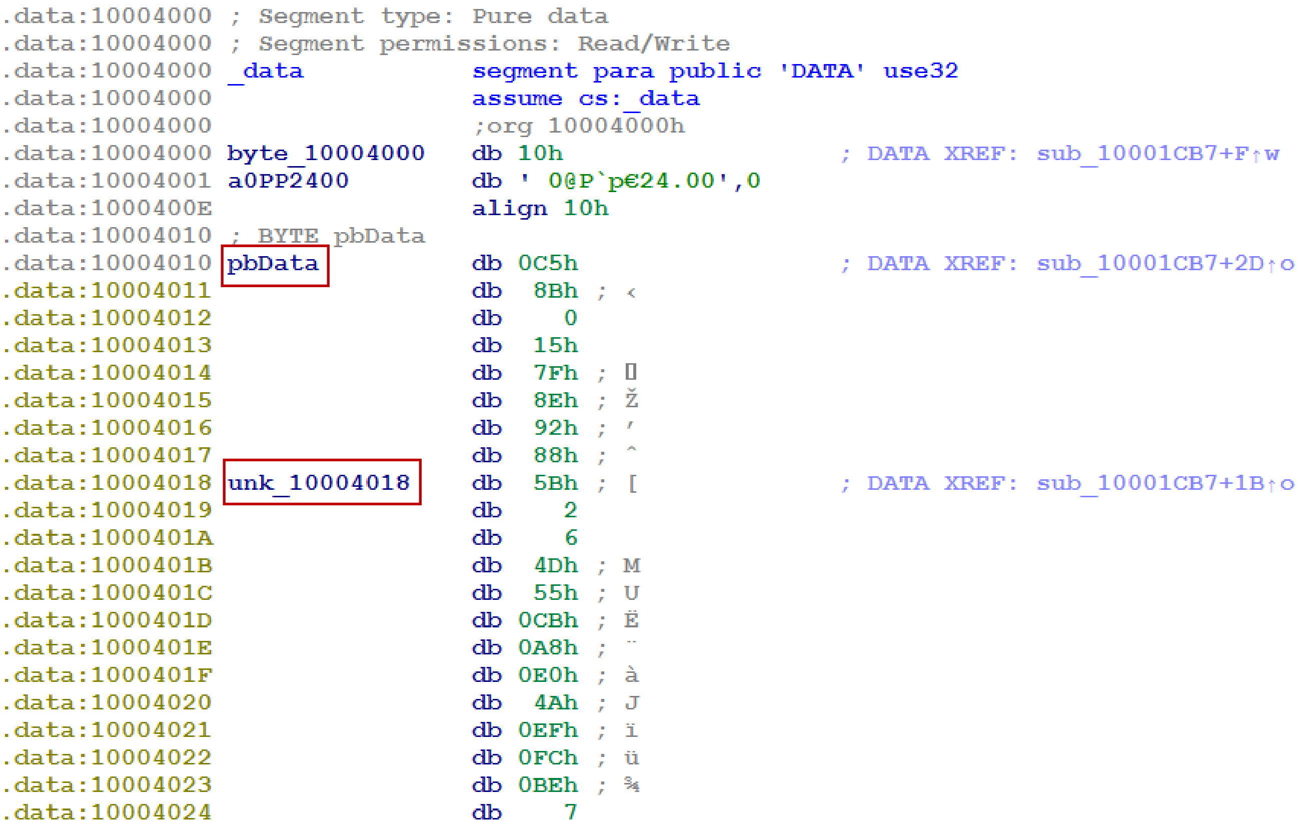
9. Reversing the decryption code

Now we have the unpacked binary, so let's open it up in IDA Pro. There're many ways to find encrypted configuration, but certainly one of easier (and a bit inaccurate) is by **looking for functions manipulating Data or Unexplored areas of color bars on IDA Pro** (actually, the unexplored are is much bigger than the shown below):



[Figure 14]

Clicking on the start of the Unexplored area above, you'll see the following code:



```
.data:10004000 ; Segment type: Pure data
.data:10004000 ; Segment permissions: Read/Write
.data:10004000 _data           segment para public 'DATA' use32
.data:10004000                 assume cs:_data
.data:10004000                 ;org 10004000h
.data:10004000 byte_10004000   db 10h                  ; DATA XREF: sub_10001CB7+F↑w
.data:10004001 a0PP2400        db ' 0@P`p€24.00',0
.data:1000400E                 align 10h
.data:10004010 ; BYTE pbData
.data:10004010 pbData          db 0C5h                 ; DATA XREF: sub_10001CB7+2D↑o
.data:10004011                 db  8Bh ; ‹
.data:10004012                 db    0
.data:10004013                 db  15h
.data:10004014                 db  7Fh ; 
.data:10004015                 db  8Eh ; Ž
.data:10004016                 db  92h ; ’
.data:10004017                 db  88h ; ˆ
.data:10004018 unk_10004018    db  5Bh ; [             ; DATA XREF: sub_10001CB7+1B↑o
.data:10004019                 db    2
.data:1000401A                 db    6
.data:1000401B                 db  4Dh ; M
.data:1000401C                 db  55h ; U
.data:1000401D                 db 0CBh ; Ë
.data:1000401E                 db 0A8h ; ¨
.data:1000401F                 db 0E0h ; à
.data:10004020                 db  4Ah ; J
.data:10004021                 db 0EFh ; ï
.data:10004022                 db 0FCh ; ü
.data:10004023                 db 0BEh ; ¾
.data:10004024                 db    7
```

[Figure 15]

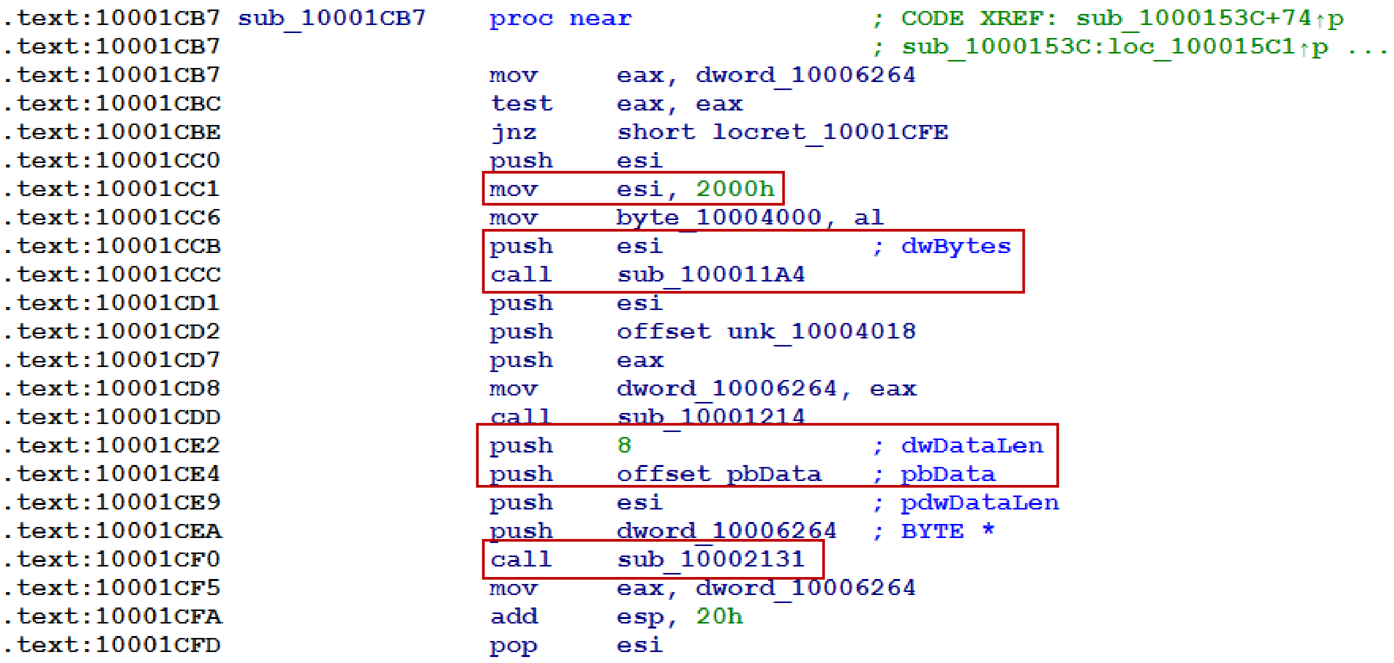
It's quite interesting to notice that we see cross references to specific addresses in this region:

- **byte_10004000** (16 bytes)
- **pbData** (8 bytes)
- **unk_10004018** (likely 0x2000 bytes)

From my experience analyzing malware, I already know that **pbData** is an important argument to a couple of APIs from Microsoft Crypto APIs, so it's an indicative that we are in the right track. Another pattern found in many well-known malware sample is the **structure key + encrypted data**, so even I don't have any further indication about this case, I could suppose that "pbData" is some key (length of 8 bytes), though sometimes it isn't the final key because malicious threats use **KDF (Key Derivation Functions)** to generate a definitive key from the provided password. Following our analysis, the **"unk10004018"** would be referring to the potentially encrypted data.

Although we're going well in the analysis, I mentioned previously this kind of "reverse thinking" is not precise because we don't actually know what the nature of the encrypted data that is stored at this address location. It would be better to analyze the malware and, from important references and functions that take these data, so we could have a better idea and context about data type involved here.

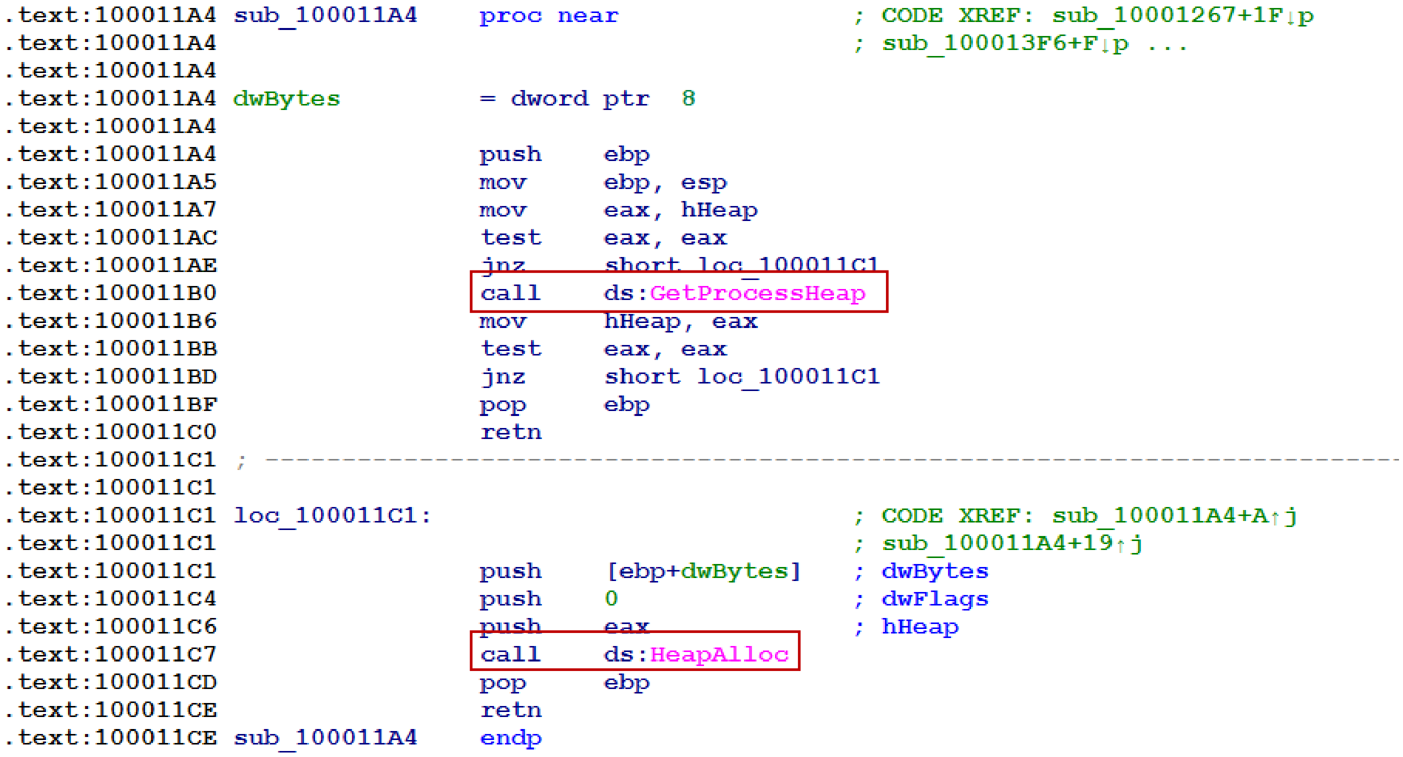
Following the cross-reference on **pbData** (**X hotkey**) we get to the subroutine **sub_10001CB7**. From this point, we clearly see our data (**pbData**) is being pushed onto the stack as argument to subroutine **sub_10002131**, as shown below:



```
.text:10001CB7 sub_10001CB7    proc near               ; CODE XREF: sub_1000153C+74↑p
.text:10001CB7                                         ; sub_1000153C:loc_100015C1↑p ...
.text:10001CB7                 mov     eax, dword_10006264
.text:10001CBC                 test    eax, eax
.text:10001CBE                 jnz     short locret_10001CFE
.text:10001CC0                 push    esi
.text:10001CC1                 mov     esi, 2000h
.text:10001CC6                 mov     byte_10004000, al
.text:10001CCB                 push    esi             ; dwBytes
.text:10001CCC                 call    sub_100011A4
.text:10001CD1                 push    esi
.text:10001CD2                 push    offset unk_10004018
.text:10001CD7                 push    eax
.text:10001CD8                 mov     dword_10006264, eax
.text:10001CDD                 call    sub_10001214
.text:10001CE2                 push    8               ; dwDataLen
.text:10001CE4                 push    offset pbData   ; pbData
.text:10001CE9                 push    esi             ; pdwDataLen
.text:10001CEA                 push    dword_10006264  ; BYTE *
.text:10001CF0                 call    sub_10002131
.text:10001CF5                 mov     eax, dword_10006264
.text:10001CFA                 add     esp, 20h
.text:10001CFD                 pop     esi
```

[Figure 16]

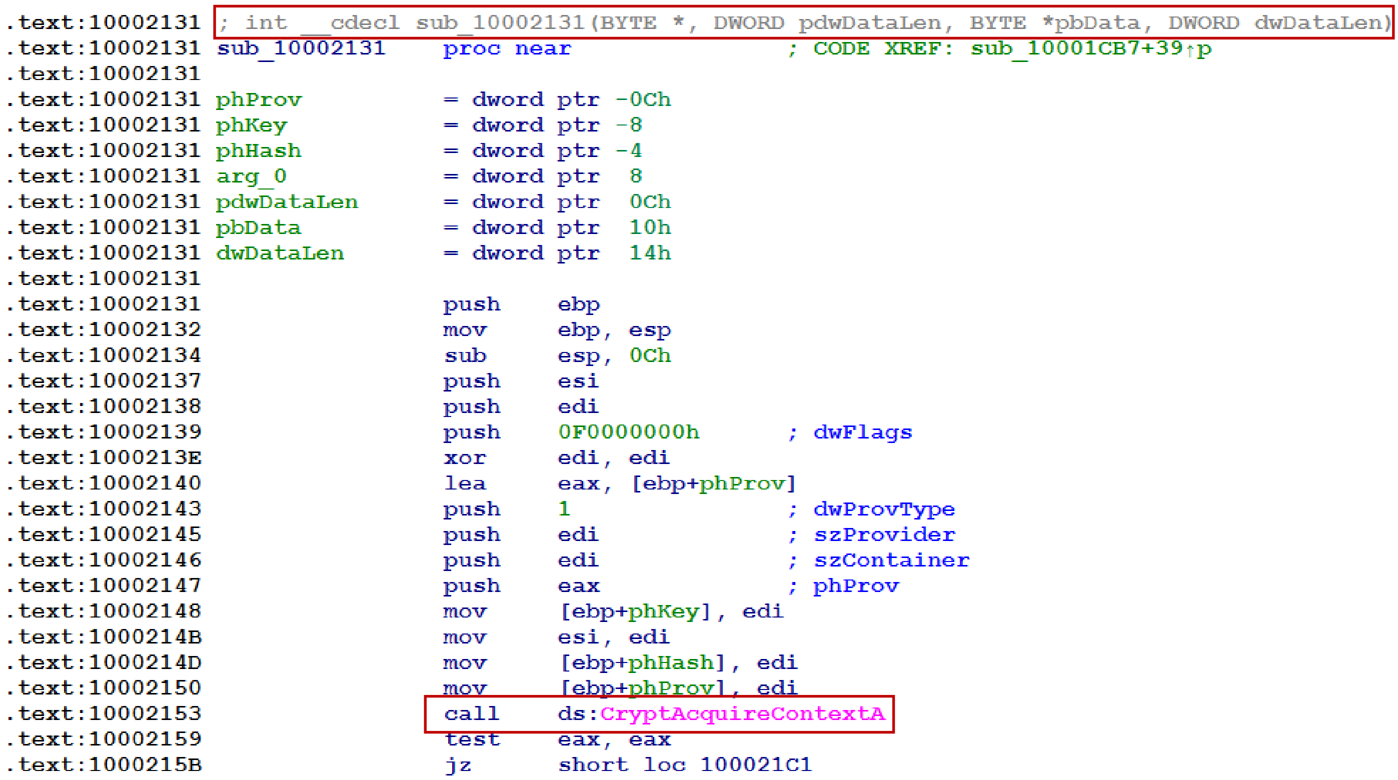
Observing other arguments and based on my previous experience, I know that other arguments being passed to **subroutine sub_10002131** are also involved with Cryptography. Please, we should also see that **dwBytes** (from esi register, which received the value of 0x2000) is being used as an argument to **subroutine sub_100011A4**, which only performs buffer allocation to receive a content, as shown below:



```
.text:100011A4 sub_100011A4    proc near               ; CODE XREF: sub_10001267+1F↓p
.text:100011A4                                         ; sub_100013F6+F↓p ...
.text:100011A4
.text:100011A4 dwBytes         = dword ptr  8
.text:100011A4
.text:100011A4                 push    ebp
.text:100011A5                 mov     ebp, esp
.text:100011A7                 mov     eax, hHeap
.text:100011AC                 test    eax, eax
.text:100011AE                 jnz     short loc_100011C1
.text:100011B0                 call    ds:GetProcessHeap
.text:100011B6                 mov     hHeap, eax
.text:100011BB                 test    eax, eax
.text:100011BD                 jnz     short loc_100011C1
.text:100011BF                 pop     ebp
.text:100011C0                 retn
.text:100011C1 ; ---------------------------------------------------------------------------
.text:100011C1
.text:100011C1 loc_100011C1:                           ; CODE XREF: sub_100011A4+A↑j
.text:100011C1                                         ; sub_100011A4+19↑j
.text:100011C1                 push    [ebp+dwBytes]   ; dwBytes
.text:100011C4                 push    0               ; dwFlags
.text:100011C6                 push    eax             ; hHeap
.text:100011C7                 call    ds:HeapAlloc
.text:100011CD                 pop     ebp
.text:100011CE                 retn
.text:100011CE sub_100011A4    endp
```

[Figure 17]

Analyzing the **subroutine sub_10002131** (a bit long) in several parts, we have the following code:

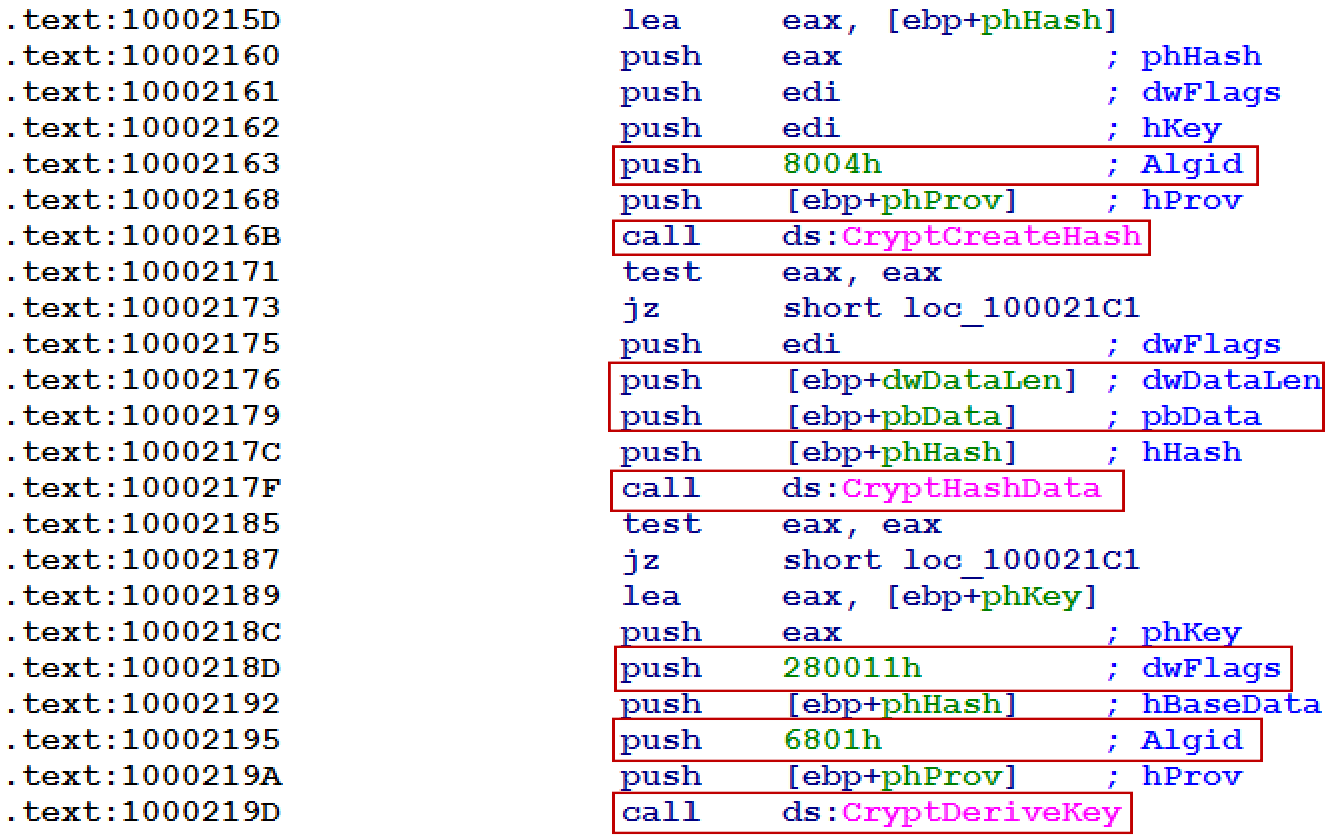


```
.text:10002131 ; int __cdecl sub_10002131(BYTE *, DWORD pdwDataLen, BYTE *pbData, DWORD dwDataLen)
.text:10002131 sub_10002131    proc near               ; CODE XREF: sub_10001CB7+39↑p
.text:10002131
.text:10002131 phProv          = dword ptr -0Ch
.text:10002131 phKey           = dword ptr -8
.text:10002131 phHash          = dword ptr -4
.text:10002131 arg_0           = dword ptr  8
.text:10002131 pdwDataLen      = dword ptr  0Ch
.text:10002131 pbData          = dword ptr  10h
.text:10002131 dwDataLen       = dword ptr  14h
.text:10002131
.text:10002131                 push    ebp
.text:10002132                 mov     ebp, esp
.text:10002134                 sub     esp, 0Ch
.text:10002137                 push    esi
.text:10002138                 push    edi
.text:10002139                 push    0F0000000h      ; dwFlags
.text:1000213E                 xor     edi, edi
.text:10002140                 lea     eax, [ebp+phProv]
.text:10002143                 push    1               ; dwProvType
.text:10002145                 push    edi             ; szProvider
.text:10002146                 push    edi             ; szContainer
.text:10002147                 push    eax             ; phProv
.text:10002148                 mov     [ebp+phKey], edi
.text:1000214B                 mov     esi, edi
.text:1000214D                 mov     [ebp+phHash], edi
.text:10002150                 mov     [ebp+phProv], edi
.text:10002153                 call    ds:CryptAcquireContextA
.text:10002159                 test    eax, eax
.text:1000215B                 jz      short loc_100021C1
```

[Figure 18]

Although **CryptAcquireContextA( )** is a deprecated function, it continue being very used by malware threats. This API is used **to acquire a handle to a key container with a CSP (Cryptographic Service Provider) and it's using a default key container name and a user default provider because both arguments are zero (from xor edi, edi).** Taking **dwFlags (0x0F0000000)** and searching this value on **wincrypt.h**, we know that it is referring to **CRYPT_VERIFYCONTEXT provider type**, which is commonly seen in applications using ephemeral keys or that don't need access to persisted private keys. Actually, as you will learn, this malware sample uses a kind of **key derivation**.

Next block of code reveals important information to our analysis:



```
.text:1000215D                  lea     eax, [ebp+phHash]
.text:10002160                  push    eax             ; phHash
.text:10002161                  push    edi             ; dwFlags
.text:10002162                  push    edi             ; hKey
.text:10002163                  push    8004h           ; Algid
.text:10002168                  push    [ebp+phProv]    ; hProv
.text:1000216B                  call    ds:CryptCreateHash
.text:10002171                  test    eax, eax
.text:10002173                  jz      short loc_100021C1
.text:10002175                  push    edi             ; dwFlags
.text:10002176                  push    [ebp+dwDataLen] ; dwDataLen
.text:10002179                  push    [ebp+pbData]    ; pbData
.text:1000217C                  push    [ebp+phHash]    ; hHash
.text:1000217F                  call    ds:CryptHashData
.text:10002185                  test    eax, eax
.text:10002187                  jz      short loc_100021C1
.text:10002189                  lea     eax, [ebp+phKey]
.text:1000218C                  push    eax             ; phKey
.text:1000218D                  push    280011h         ; dwFlags
.text:10002192                  push    [ebp+phHash]    ; hBaseData
.text:10002195                  push    6801h           ; Algid
.text:1000219A                  push    [ebp+phProv]    ; hProv
.text:1000219D                  call    ds:CryptDeriveKey
```

[Figure 19]

This block contains three APIs and we have to analyze them one by one:

a. **CryptCreateHash( ):**

This function creates a **CSP hash object** and only one of its arguments, **Algid**, is quite interesting and it set to **8004h**.

Searching on https://docs.microsoft.com/en-us/windows/win32/seccrypto/alg-id we are able to learn that **0x8004h means CALG_SHA1**, so this malware is manipulating **SHA1 hash (160 bits / 20 bytes).** The return of this function is a handle to the CSP hash object (saved into the **phHash argument**, which was initially zeroed) that is going to be used in the next function.

b. CryptHashData( ):

This function adds data (given by **pbData argument**) of an specific size (given by **dwDataLen argument**) into the hash object returned by **CryptCreateHash( ).** Both arguments are the third and fourth arguments of subroutine **sub_10002131** (Figure 18). Following them, these arguments comes from subroutine **sub_10001CB7** (Figure 16), which shows that the size is 8 bytes and refers to the possible key (**pbData**). At this point, **CryptHashData( ) function** is ingesting the possible key (**8 bytes**) into the hash object, so generating a **SHA1 hash**. In other words, the malware's code is generating a hash output from an entry and the returned handle from **CryptCreateHash( )** refers to the **CSP hash object,** which holds this hashed data.

c. CryptDeriveKey( ):

This function generates *session keys* derived from a given seed data value and, according to its definition, it guarantees that same sessions key will be generated using the same base data, so it's completely suitable for our case because we're decrypting a configuration data. There are two interesting arguments here: a**. Algid is 0x6801** and **dwFlags is 0x280011**, and we need to discuss about them.

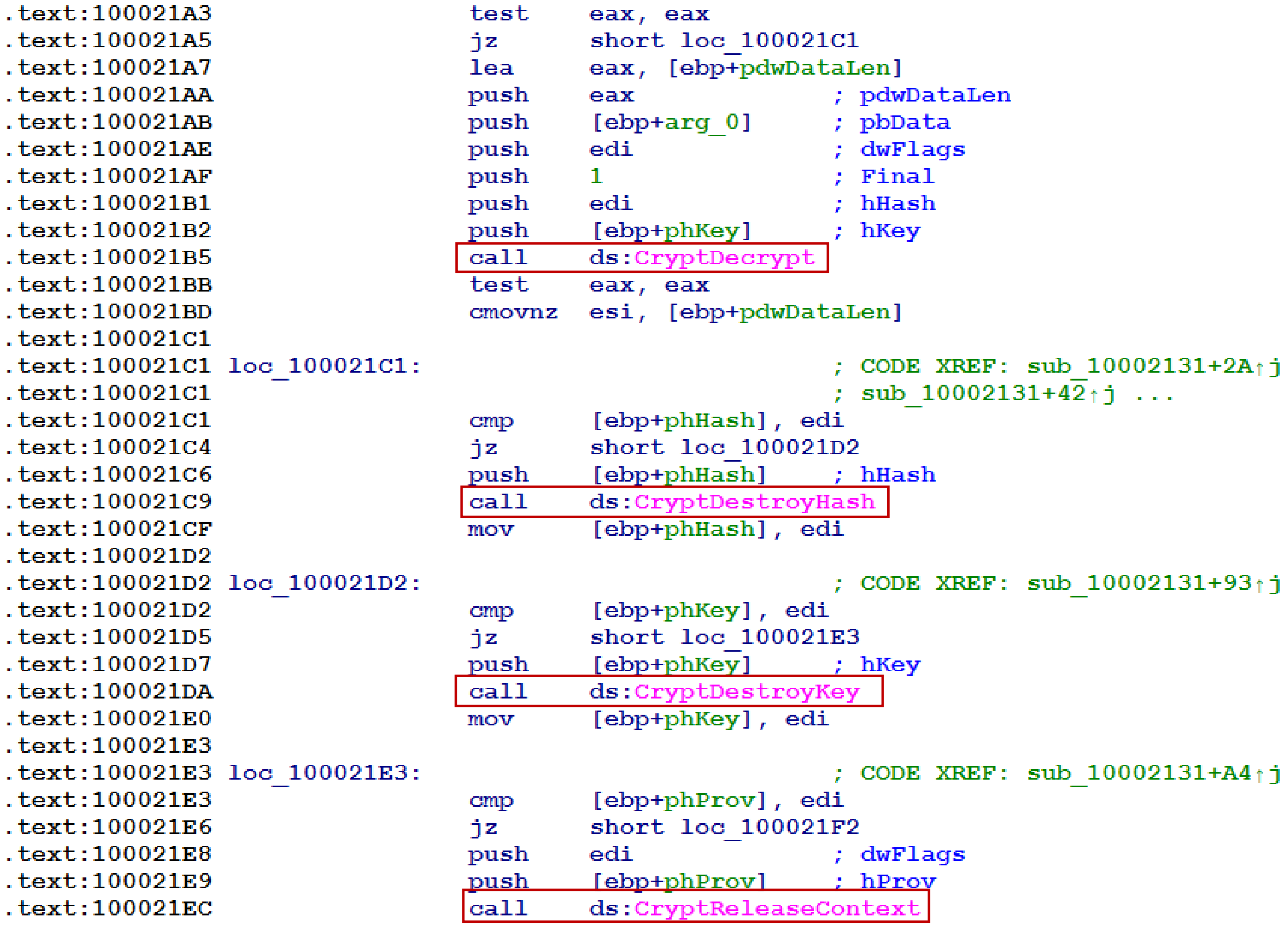
The second argument (**Algid == 0x6801**) identifies the symmetric encryption algorithm and according to the documentation (https://docs.microsoft.com/en-us/windows/win32/seccrypto/alg-id), **0x6801** means **CALG_RC4**.

The fourth argument (**dwFlags == 0x280011**) deserves further details. According to documentation (https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptderivekey): *"The **key size**, representing the length of the key modulus in bits, is set with the **upper 16 bits of this parameter**. Thus, if a 128-bit RC4 session key is to be generated, the value 0x00800000 is combined with any other dwFlags predefined value with a bitwise-OR operation".* Thus, **we know the RC4 key has 40 bits (0x28), so it has 5 bytes**. Another part from MSDN documentation tells that *"The lower 16 bits of this parameter can be zero or you can specify one or more of the following flags by using the bitwise-OR operator to combine them".* Taking the possible values from MSDN page and searching for them on **wincrypt.h file** (**ReacOS** provides us a sample: https://doxygen.reactos.org/d7/d4a/wincrypt_8h_source.html) we found that **0x11** means **CRYPT_NO_SALT** + **CRYPT_EXPORTABLE** (both self-explaining).

Therefore, it's really interesting to figure out that the malware's author is **using SHA1 as a KDF (Key Derivation Function)** to generate the final decryption key, which can be explained by the following sequence:

a) **pbData = C58B00157F8E9288** (Remember: to export data you should use **SHIFT-E**)
b) pbData → SHA1 (20 bytes)
c) **SHA1 → CryptHashData( ) → CryptDeriveKey( ) → RC4 key (5 bytes**)

Of course, there're much better **KDF** such as **Bcrypt, Scrypt and Argon2** that might be used in real applications, but we're sure that malware's author wasn't concerned to aspects of using or not a KDF **resistant to FPGA, ASIC and GPU attacks**.



```
.text:100021A3                test    eax, eax
.text:100021A5                jz      short loc_100021C1
.text:100021A7                lea     eax, [ebp+pdwDataLen]
.text:100021AA                push    eax             ; pdwDataLen
.text:100021AB                push    [ebp+arg_0]     ; pbData
.text:100021AE                push    edi             ; dwFlags
.text:100021AF                push    1               ; Final
.text:100021B1                push    edi             ; hHash
.text:100021B2                push    [ebp+phKey]     ; hKey
.text:100021B5                call    ds:CryptDecrypt
.text:100021BB                test    eax, eax
.text:100021BD                cmovnz  esi, [ebp+pdwDataLen]
.text:100021C1
.text:100021C1 loc_100021C1:                          ; CODE XREF: sub_10002131+2A↑j
.text:100021C1                                        ; sub_10002131+42↑j ...
.text:100021C1                cmp     [ebp+phHash], edi
.text:100021C4                jz      short loc_100021D2
.text:100021C6                push    [ebp+phHash]    ; hHash
.text:100021C9                call    ds:CryptDestroyHash
.text:100021CF                mov     [ebp+phHash], edi
.text:100021D2
.text:100021D2 loc_100021D2:                          ; CODE XREF: sub_10002131+93↑j
.text:100021D2                cmp     [ebp+phKey], edi
.text:100021D5                jz      short loc_100021E3
.text:100021D7                push    [ebp+phKey]     ; hKey
.text:100021DA                call    ds:CryptDestroyKey
.text:100021E0                mov     [ebp+phKey], edi
.text:100021E3
.text:100021E3 loc_100021E3:                          ; CODE XREF: sub_10002131+A4↑j
.text:100021E3                cmp     [ebp+phProv], edi
.text:100021E6                jz      short loc_100021F2
.text:100021E8                push    edi             ; dwFlags
.text:100021E9                push    [ebp+phProv]    ; hProv
.text:100021EC                call    ds:CryptReleaseContext
```

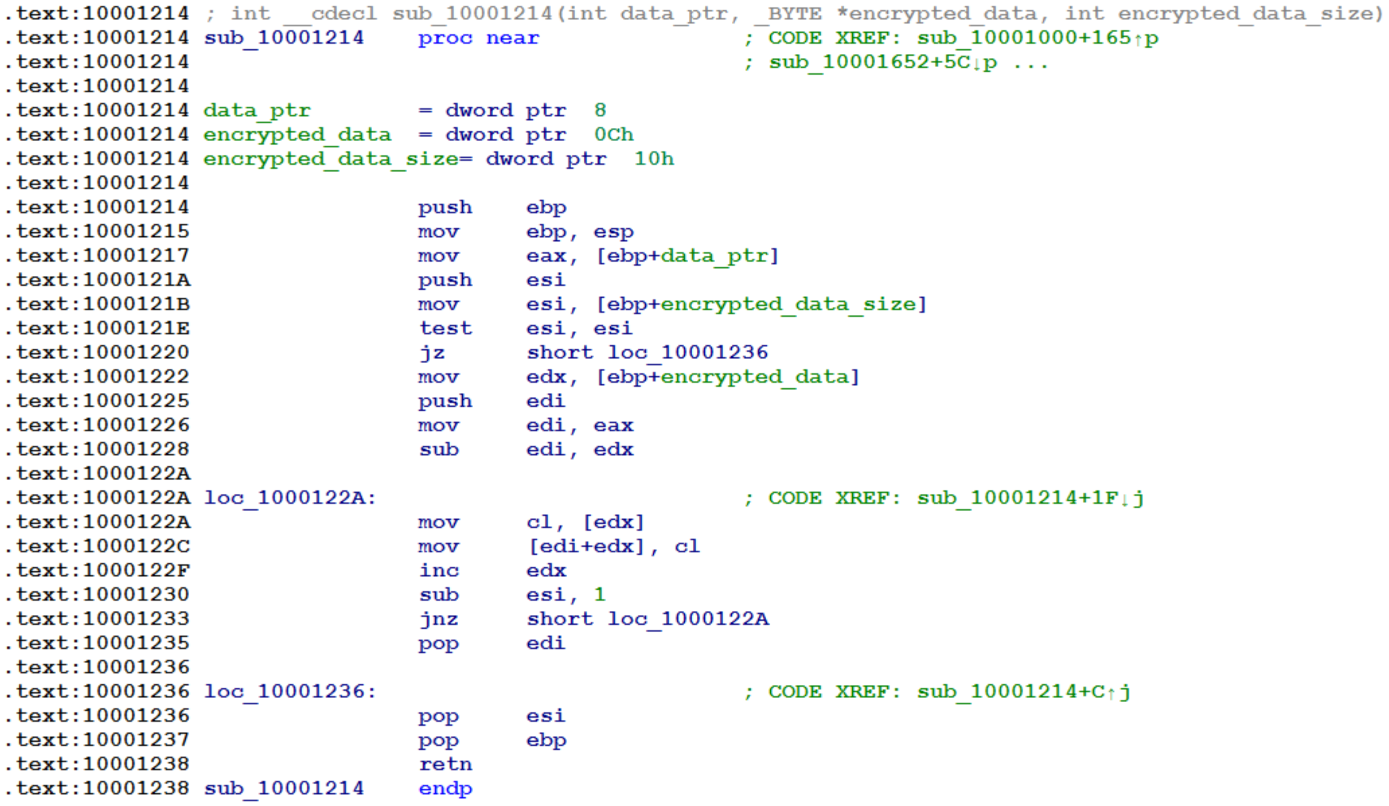
[Figure 20]

d. CryptDecrypt( )

This function is responsible for decrypting encrypted data by using a provided handle to the key (**hKey**). Other arguments provided are **hHash** (handle to the hash object resulting from **CryptCreateHash( )**), **Final** (value equal to 1 because is the last and unique section being decrypted), **pbData** (buffer containing the data to be decrypted) and **pbwDataLen** (pointer to a DWORD that indicates the **length of the pbData** buffer that, in this case, is **0x2000**).

Remember that **pbwDataLen (0x2000)** is the second argument of the function being analyzed (***sub_10002131(BYTE *rc4_encrypted_data, DWORD pdwDataLen, BYTE *pbData, DWORD dwDataLen)***). Therefore, the **final result (RC4 decrypted data)** is saved into the **pbData** and the respective size is saved into **pdwDataLen** argument.

Another interesting point is that the **pbData here is NOT the 8-byte key, but the encrypted data coming from unk_10004018 data reference that was transferred to dword_10006264 memory defined array within the subroutine sub_10001214**, which I've renamed its arguments (**"N" shortcut**), as shown in the next figure:



```
.text:10001214 ; int __cdecl sub_10001214(int data_ptr, _BYTE *encrypted_data, int encrypted_data_size)
.text:10001214 sub_10001214    proc near               ; CODE XREF: sub_10001000+165↑p
.text:10001214                                         ; sub_10001652+5C↓p ...
.text:10001214
.text:10001214 data_ptr        = dword ptr  8
.text:10001214 encrypted_data  = dword ptr  0Ch
.text:10001214 encrypted_data_size= dword ptr  10h
.text:10001214
.text:10001214                 push    ebp
.text:10001215                 mov     ebp, esp
.text:10001217                 mov     eax, [ebp+data_ptr]
.text:1000121A                 push    esi
.text:1000121B                 mov     esi, [ebp+encrypted_data_size]
.text:1000121E                 test    esi, esi
.text:10001220                 jz      short loc_10001236
.text:10001222                 mov     edx, [ebp+encrypted_data]
.text:10001225                 push    edi
.text:10001226                 mov     edi, eax
.text:10001228                 sub     edi, edx
.text:1000122A
.text:1000122A loc_1000122A:                           ; CODE XREF: sub_10001214+1F↓j
.text:1000122A                 mov     cl, [edx]
.text:1000122C                 mov     [edi+edx], cl
.text:1000122F                 inc     edx
.text:10001230                 sub     esi, 1
.text:10001233                 jnz     short loc_1000122A
.text:10001235                 pop     edi
.text:10001236
.text:10001236 loc_10001236:                           ; CODE XREF: sub_10001214+C↑j
.text:10001236                 pop     esi
.text:10001237                 pop     ebp
.text:10001238                 retn
.text:10001238 sub_10001214    endp
```

[Figure 21]

The remaining functions (**CryptDestroyHash, CryptDestroyKey and CryptReleaseContext**) have clear meanings and we don't need to explain them here.

Finally we have all necessary information that we are going to use in the next step to write a configuration extractor and decryptor:

- **initial key:** C58B00157F8E9288 (after first 16 bytes of .data section)
- **initial data address:** 0x10004018
- **data size:** 0x2000
- **hash algorithm:** SHA1 (20 bytes)
- **decryption algorithm:** RC4
- **RC4 key size:** 5 bytes

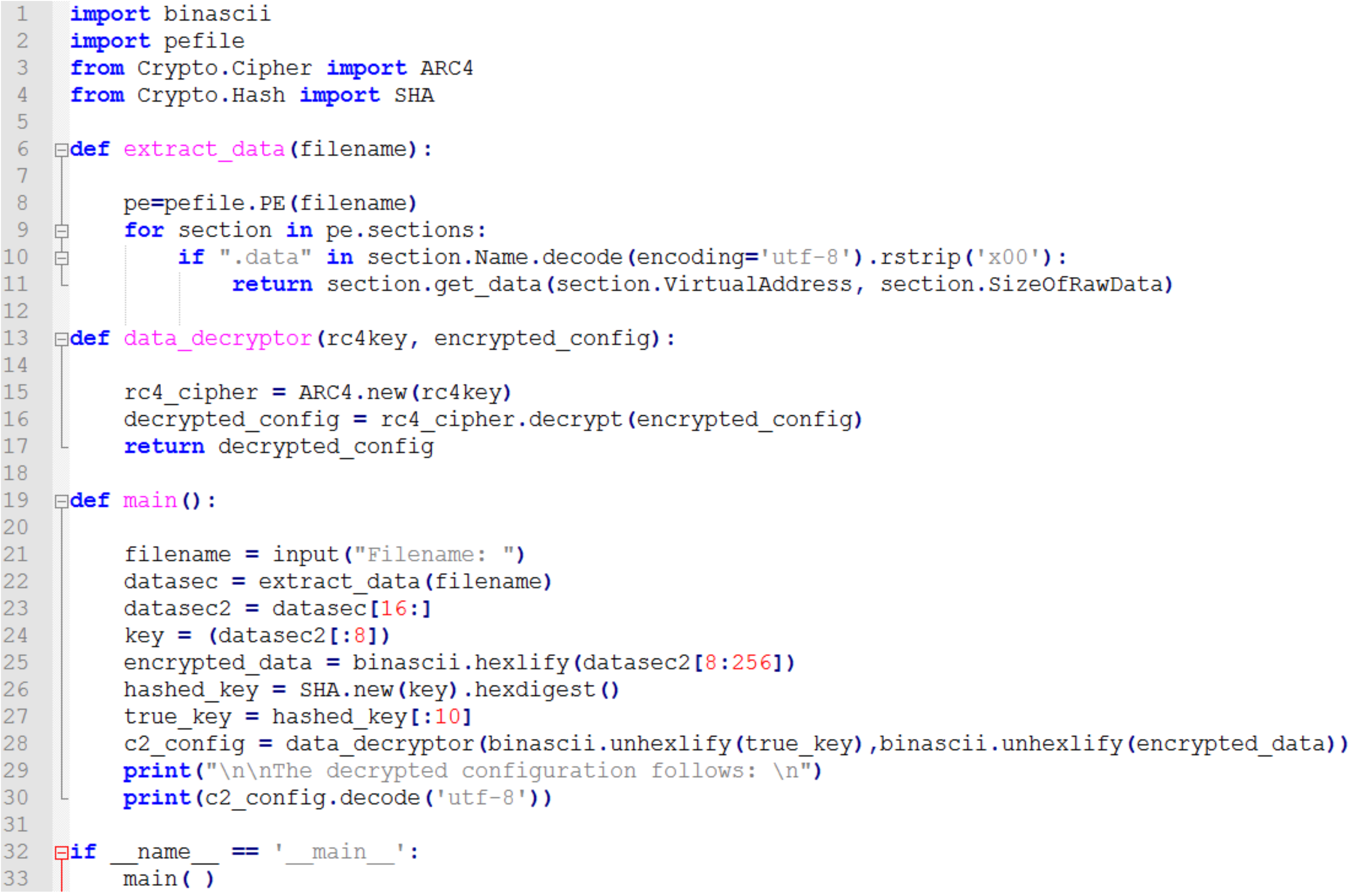
Let's proceed to the next section and write a C2 data configuration extractor and decryptor.

10. Writing a data configuration extractor

Writing a configuration extractor is a task that might seem complicated at first time, but this only a matter of learning the path once and, afterwards, you can take your own steps. Additionally, it isn't not a such thing (and not even new from last couple of years) as you could believe, and personally I've being doing it for many years.

Several languages might be used, but I chose Python 3, which have nice features to writing any decoder. I'll try not only to explain what each line does, but also show reasons for each decision.

My data C2 data decryptor code (named **hancitor_conf_extractor_1.py**) follows below:



```
1   import binascii
2   import pefile
3   from Crypto.Cipher import ARC4
4   from Crypto.Hash import SHA
5
6   def extract_data(filename):
7
8       pe=pefile.PE(filename)
9       for section in pe.sections:
10          if ".data" in section.Name.decode(encoding='utf-8').rstrip('x00'):
11              return section.get_data(section.VirtualAddress, section.SizeOfRawData)
12
13  def data_decryptor(rc4key, encrypted_config):
14
15      rc4_cipher = ARC4.new(rc4key)
16      decrypted_config = rc4_cipher.decrypt(encrypted_config)
17      return decrypted_config
18
19  def main():
20
21      filename = input("Filename: ")
22      datasec = extract_data(filename)
23      datasec2 = datasec[16:]
24      key = (datasec2[:8])
25      encrypted_data = binascii.hexlify(datasec2[8:256])
26      hashed_key = SHA.new(key).hexdigest()
27      true_key = hashed_key[:10]
28      c2_config = data_decryptor(binascii.unhexlify(true_key),binascii.unhexlify(encrypted_data))
29      print("\n\nThe decrypted configuration follows: \n")
30      print(c2_config.decode('utf-8'))
31
32  if __name__ == '__main__':
33      main( )
```

[Figure 22]

Let's try to explain line by line of the code, but not at the exact order of the Python 3 script:

a. **Lines 1, 2, 3 and 4** perform necessary imports because code is manipulating a PE file and this Python 3 script aims to decrypt data involved with RC4 and SHA1 algorithms, as well as it's needed to handle with transformations from binary to ascii, so **binascii package** is required too.

b. **On lines 21, 22, 23 and 24** we start the **main( )** definition and the code asks for user to enter the filename of an unpacked Hancitor binary which the encrypted data is going to extracted from. The extracted data is stored into the **datasec variable** as bytes.

c. The **extract_data( )** (**lines 6 to 13**) doesn't have anything new, but we should to highlight three points: i.) our **target section is the ".data"**, where are stored the key and encrypted data (check Figure 15); ii.) we are concerned to **Unicode formatting** and that's the reason of being using **decode(encoding='utf-8')**; iii.) the code is stripping possible '0x00' from section names (common while handling Unicode names); iv.) we're using **PE properties to delimit where the .data section starts and ends**.

d. **Line 25** (**datasec2 = datasec[16:]**) defines datasec2 variable that contains **key + all encrypted data except the first 16 bytes**, which is likely related to a campaign ID or something similar.

e. **Line 26** (**key = (datasec2[:8])**) defines the **key variable** that, as we learned previously, is composed by the **first 8 bytes** of datasec2.

f. **Line 27** (**encrypted_data = binascii.hexlify(datasec2[8:256]**) stores the **encrypted data configuration** into **encrypted_data**. Actually, as we learned from the reversed code, the buffer's size reserved by the binary code is **0x2000**, but as you'll see, collecting **248 bytes is enough**, but you could adjust it according to your needs. The **hexlify( )** translates binary data to hexadecimal format.

g. **Line 28** (**hashed_key = SHA.new(key).hexdigest()**) generates a **hexadecimal SHA1 hash**.

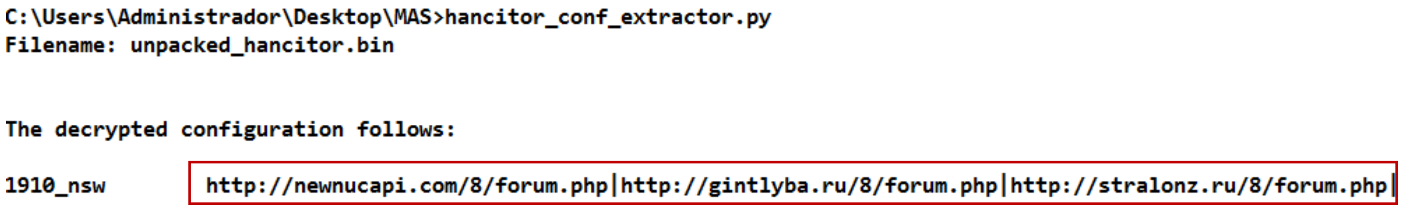
e. **Line 29** (**true_key = hashed_key[:10]**) collects **only the first 10 hexadecimal (5 bytes)** according to learned from **CryptDeriveKey( )** explanation on **page 27**.

f. **Line 30** (**data_decryptor(binascii.unhexlify(true_key),binascii.unhexlify(encrypted_data))**) calls the decryptor function and one of important facts is that **we must transform data from hexadecimal string to binary representation before using RC4 functions**.

g. The **data_decryptor( )** (**lines 15 to 19**) is quite simple and basically decrypts the encrypted data through **RC4 algorithm** using the given key (**true_key variable from line 29**).

h. Finally, **on line 32** (**print(c2_config.decode('utf-8'))**) the result is sent to terminal. Once again, you should note that before printing the decrypted data we need to decode it **assuming to be handling with a possible Unicode character set.**

Executing this Python 3 script against our unpacked Hancitor binary sample we have:

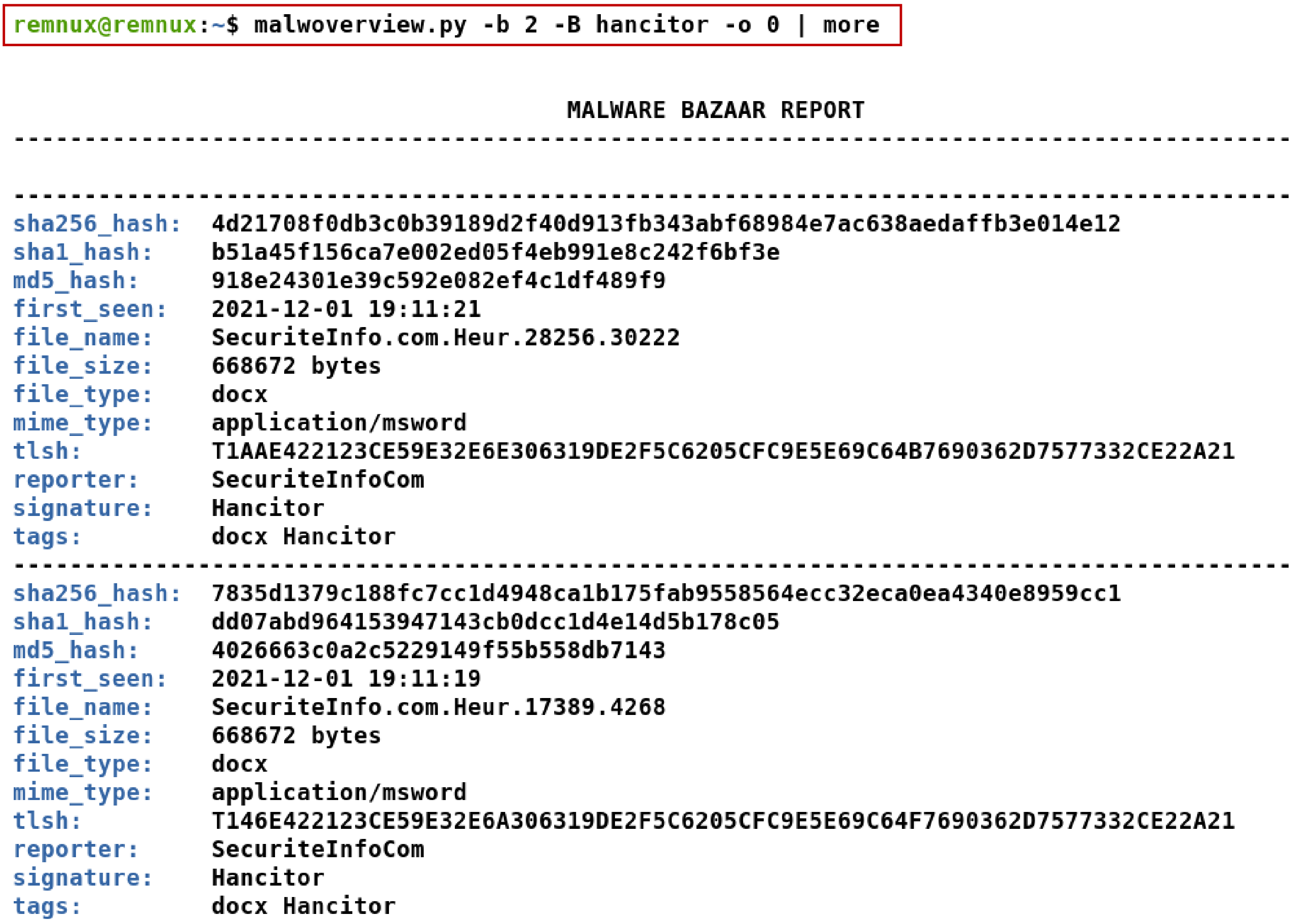


```
C:\Users\Administrador\Desktop\MAS>hancitor_conf_extractor.py
Filename: unpacked_hancitor.bin


The decrypted configuration follows:

1910_nsw         http://newnucapi.com/8/forum.php|http://gintlyba.ru/8/forum.php|http://stralonz.ru/8/forum.php|
```

[Figure 23]

That's great! We managed to **extract and decrypt the Hancitor C2 configuration** from the unpacked Hancitor binary. The advantage of writing a decryptor script is **that we can use it against all Hancitors sample that follow the same binary pattern.**

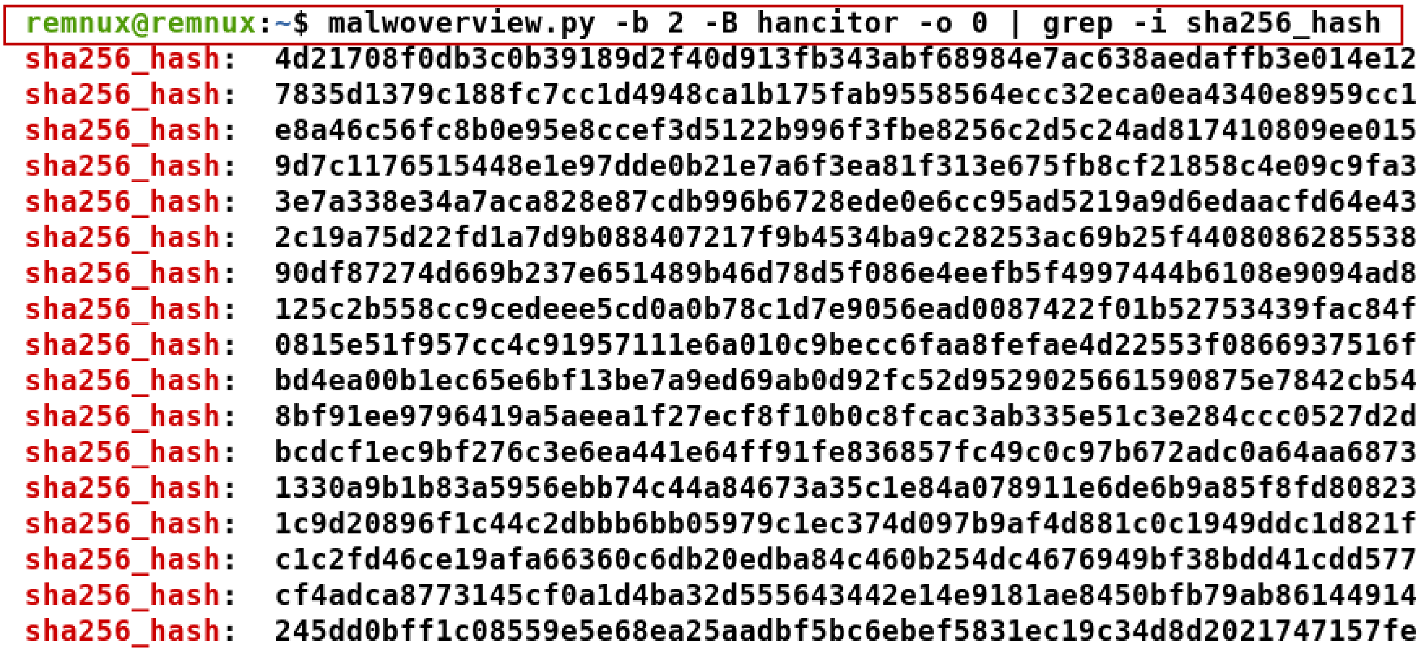
To confirm our script, let's look for another Hancitor sample, unpacking it, trying to extract and decrypt its C2 data configuration. **Malware Bazaar** offers an endless number of Hancitor samples and we can use **Malwoverview** to list and download them, as shown on the next figure:



```
remnux@remnux:~$ malwoverview.py -b 2 -B hancitor -o 0 | more

                                        MALWARE BAZAAR REPORT
------------------------------------------------------------------------------------------


------------------------------------------------------------------------------------------
sha256_hash:  4d21708f0db3c0b39189d2f40d913fb343abf68984e7ac638aedaffb3e014e12
sha1_hash:    b51a45f156ca7e002ed05f4eb991e8c242f6bf3e
md5_hash:     918e24301e39c592e082ef4c1df489f9
first_seen:   2021-12-01 19:11:21
file_name:    SecuriteInfo.com.Heur.28256.30222
file_size:    668672 bytes
file_type:    docx
mime_type:    application/msword
tlsh:         T1AAE422123CE59E32E6E306319DE2F5C6205CFC9E5E69C64B7690362D7577332CE22A21
reporter:     SecuriteInfoCom
signature:    Hancitor
tags:         docx Hancitor
------------------------------------------------------------------------------------------
sha256_hash:  7835d1379c188fc7cc1d4948ca1b175fab9558564ecc32eca0ea4340e8959cc1
sha1_hash:    dd07abd964153947143cb0dcc1d4e14d5b178c05
md5_hash:     4026663c0a2c5229149f55b558db7143
first_seen:   2021-12-01 19:11:19
file_name:    SecuriteInfo.com.Heur.17389.4268
file_size:    668672 bytes
file_type:    docx
mime_type:    application/msword
tlsh:         T146E422123CE59E32E6A306319DE2F5C6205CFC9E5E69C64F7690362D7577332CE22A21
reporter:     SecuriteInfoCom
signature:    Hancitor
tags:         docx Hancitor
```

[Figure 24]

Collecting many possible **Hancitor hashes** is pretty easy, as shown below (*the listing has been truncated*):



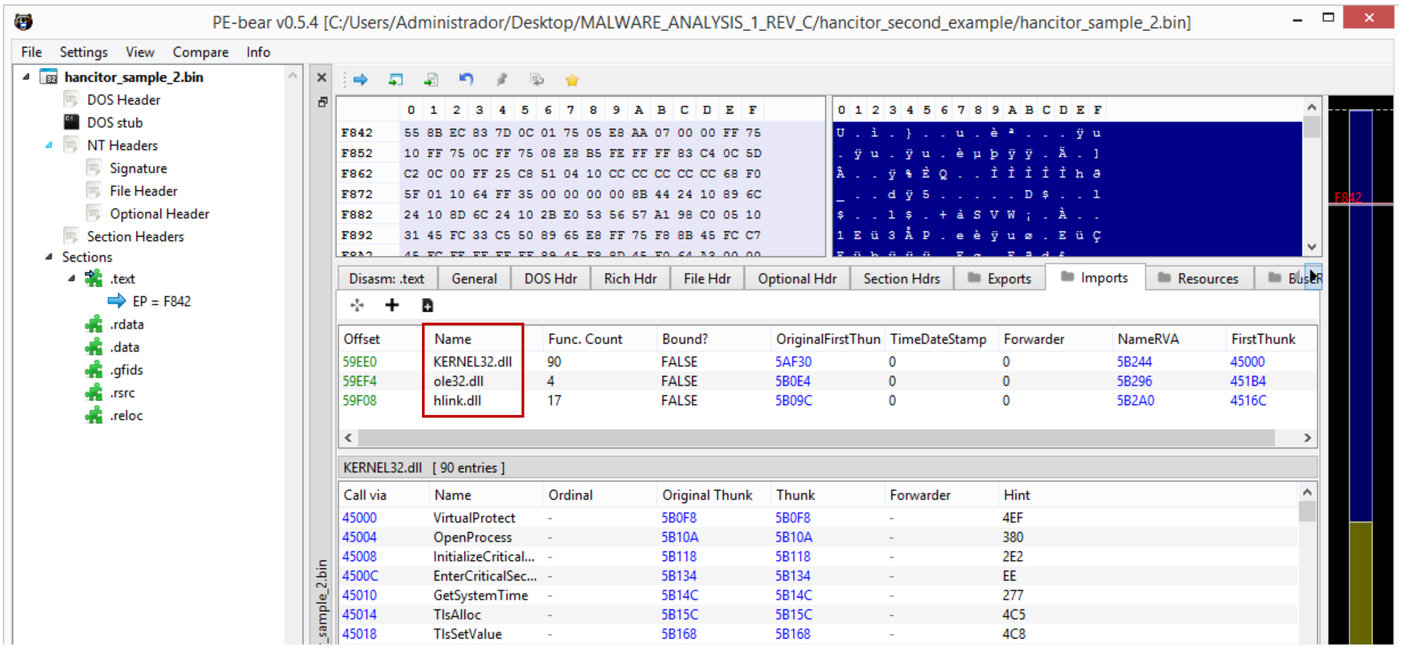
```
remnux@remnux:~$ malwoverview.py -b 2 -B hancitor -o 0 | grep -i sha256_hash
sha256_hash:  4d21708f0db3c0b39189d2f40d913fb343abf68984e7ac638aedaffb3e014e12
sha256_hash:  7835d1379c188fc7cc1d4948ca1b175fab9558564ecc32eca0ea4340e8959cc1
sha256_hash:  e8a46c56fc8b0e95e8ccef3d5122b996f3fbe8256c2d5c24ad817410809ee015
sha256_hash:  9d7c1176515448e1e97dde0b21e7a6f3ea81f313e675fb8cf21858c4e09c9fa3
sha256_hash:  3e7a338e34a7aca828e87cdb996b6728ede0e6cc95ad5219a9d6edaacfd64e43
sha256_hash:  2c19a75d22fd1a7d9b088407217f9b4534ba9c28253ac69b25f4408086285538
sha256_hash:  90df87274d669b237e651489b46d78d5f086e4eefb5f4997444b6108e9094ad8
sha256_hash:  125c2b558cc9cedeee5cd0a0b78c1d7e9056ead0087422f01b52753439fac84f
sha256_hash:  0815e51f957cc4c91957111e6a010c9becc6faa8fefae4d22553f0866937516f
sha256_hash:  bd4ea00b1ec65e6bf13be7a9ed69ab0d92fc52d9529025661590875e7842cb54
sha256_hash:  8bf91ee9796419a5aeea1f27ecf8f10b0c8fcac3ab335e51c3e284ccc0527d2d
sha256_hash:  bcdcf1ec9bf276c3e6ea441e64ff91fe836857fc49c0c97b672adc0a64aa6873
sha256_hash:  1330a9b1b83a5956ebb74c44a84673a35c1e84a078911e6de6b9a85f8fd80823
sha256_hash:  1c9d20896f1c44c2dbbb6bb05979c1ec374d097b9af4d881c0c1949ddc1d821f
sha256_hash:  c1c2fd46ce19afa66360c6db20edba84c460b254dc4676949bf38bdd41cdd577
sha256_hash:  cf4adca8773145cf0a1d4ba32d555643442e14e9181ae8450bfb79ab86144914
sha256_hash:  245dd0bff1c08559e5e68ea25aadbf5bc6ebef5831ec19c34d8d2021747157fe
```

[Figure 25]

Therefore, we can pick up one of these hashes, download the respective sample from **Malware Bazaar**, unzip it (password: **infected**) and load it into **PE Bear** (**Figures 26** and **27**):

```
remnux@remnux:~/malware/mas$ malwoverview.py -b 5 -B ede4c71a7d7a09d4da1860cbfec4a0a02104b510eb359883e
3276a018f39ead8 -o 0

                                      MALWARE BAZAAR REPORT

----------------------------------------------------------------------------------------------------

SAMPLE SAVED!

remnux@remnux:~/malware/mas$ 7z e ede4c71a7d7a09d4da1860cbfec4a0a02104b510eb359883e3276a018f39ead8.zip


7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,2 CPUs Intel(R) Core(TM) i7-1087
5H CPU @ 2.30GHz (A0652),ASM,AES-NI)

Scanning the drive for archives:
1 file, 210062 bytes (206 KiB)

Extracting archive: ede4c71a7d7a09d4da1860cbfec4a0a02104b510eb359883e3276a018f39ead8.zip
--
Path = ede4c71a7d7a09d4da1860cbfec4a0a02104b510eb359883e3276a018f39ead8.zip
Type = zip
Physical Size = 210062


Enter password (will not be echoed):
Everything is Ok

Size:       388608
Compressed: 210062
```

[Figure 26]



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
F842  55 8B EC 83 7D 0C 01 75 05 E8 AA 07 00 00 FF 75
F852  10 FF 75 0C FF 75 08 E8 B5 FE FF FF 83 C4 0C 5D
F862  C2 0C 00 FF 25 C8 51 04 10 CC CC CC CC CC 68 F0
F872  5F 01 10 64 FF 35 00 00 00 00 8B 44 24 10 89 6C
F882  24 10 8D 6C 24 10 2B E0 53 56 57 A1 98 C0 05 10
F892  31 45 FC 33 C5 50 89 65 E8 FF 75 F8 8B 45 FC C7
```

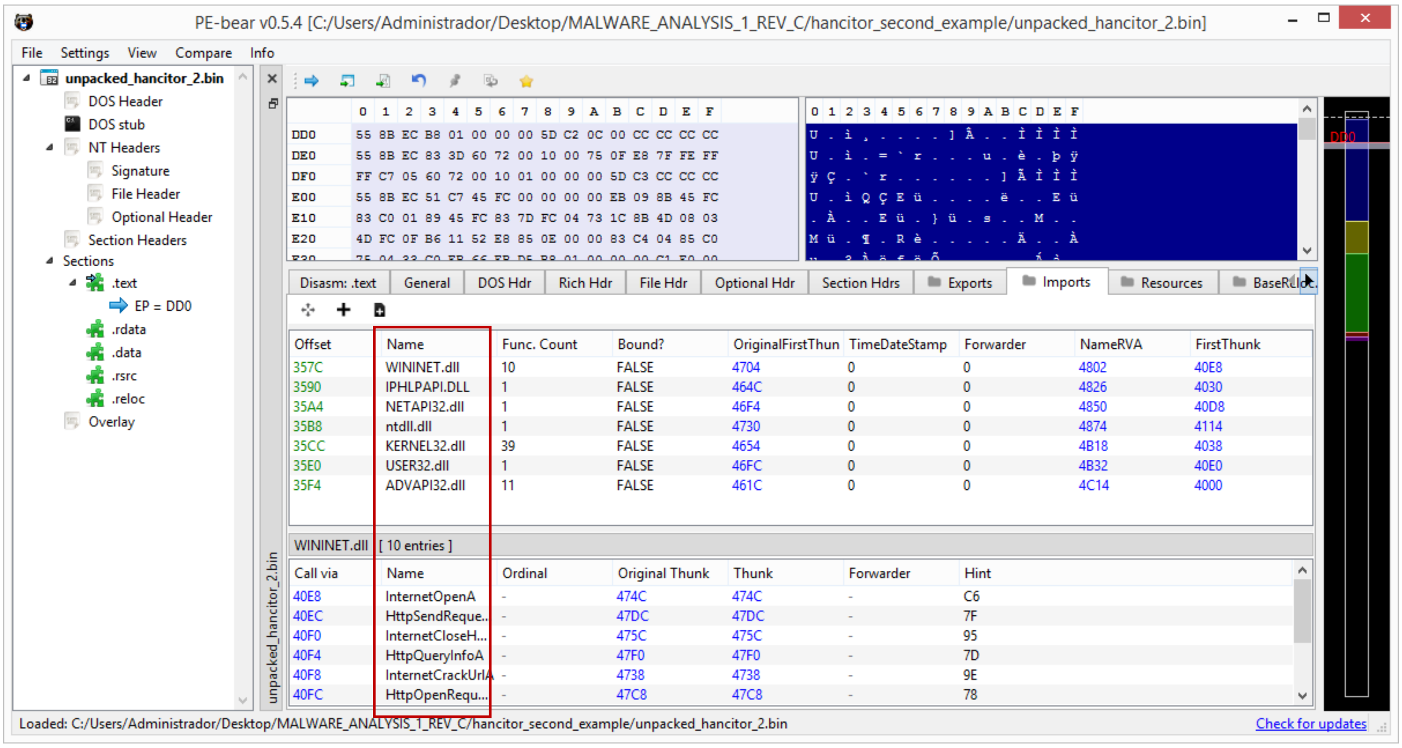
Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
59EE0	KERNEL32.dll	90	FALSE	5AF30	0	0	5B244	45000
59EF4	ole32.dll	4	FALSE	5B0E4	0	0	5B296	451B4
59F08	hlink.dll	17	FALSE	5B09C	0	0	5B2A0	4516C

KERNEL32.dll [90 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
45000	VirtualProtect	-	5B0F8	5B0F8	-	4EF
45004	OpenProcess	-	5B10A	5B10A	-	380
45008	InitializeCritical...	-	5B118	5B118	-	2E2
4500C	EnterCriticalSec...	-	5B134	5B134	-	EE
45010	GetSystemTime	-	5B14C	5B14C	-	277
45014	TlsAlloc	-	5B15C	5B15C	-	4C5
45018	TlsSetValue	-	5B168	5B168	-	4C8

[Figure 27]

After unpacking this sample using the same method and breakpoints (on **x32dbg**) as shown previously, we should verify it on **PE Bear** as shown below:



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     0 1 2 3 4 5 6 7 8 9 A B C D E F
DD0   55 8B EC B8 01 00 00 00 5D C2 0C 00 CC CC CC CC    U . ì , . . . . ] Â . . Ì Ì Ì Ì
DE0   55 8B EC 83 3D 60 72 00 10 00 75 0F E8 7F FE FF    U . ì . = ` r . . . u . è . þ ÿ
DF0   FF C7 05 60 72 00 10 01 00 00 00 5D C3 CC CC CC    ÿ Ç . ` r . . . . . . ] Ã Ì Ì Ì
E00   55 8B EC 51 C7 45 FC 00 00 00 00 EB 09 8B 45 FC    U . ì Q Ç E ü . . . . ë . . E ü
E10   83 C0 01 89 45 FC 83 7D FC 04 73 1C 8B 4D 08 03    . À . . E ü . } ü . s . . M . .
E20   4D FC 0F B6 11 52 E8 85 0E 00 00 83 C4 04 85 C0    M ü . ¶ . R è . . . . . Ä . . À
```

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
357C	WININET.dll	10	FALSE	4704	0	0	4802	40E8
3590	IPHLPAPI.DLL	1	FALSE	464C	0	0	4826	4030
35A4	NETAPI32.dll	1	FALSE	46F4	0	0	4850	40D8
35B8	ntdll.dll	1	FALSE	4730	0	0	4874	4114
35CC	KERNEL32.dll	39	FALSE	4654	0	0	4B18	4038
35E0	USER32.dll	1	FALSE	46FC	0	0	4B32	40E0
35F4	ADVAPI32.dll	11	FALSE	461C	0	0	4C14	4000

WININET.dll [10 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
40E8	InternetOpenA	-	474C	474C	-	C6
40EC	HttpSendReque...	-	47DC	47DC	-	7F
40F0	InternetCloseH...	-	475C	475C	-	95
40F4	HttpQueryInfoA	-	47F0	47F0	-	7D
40F8	InternetCrackUrlA	-	4738	4738	-	9E
40FC	HttpOpenRequ...	-	47C8	47C8	-	78

Loaded: C:/Users/Administrador/Desktop/MALWARE_ANALYSIS_1_REV_C/hancitor_second_example/unpacked_hancitor_2.bin

Check for updates

[Figure 28]

Using our C2 data decryptor script against this second unpacked Hancitor we have:

```
C:\Users\Administrador\Desktop\MAS>hancitor_conf_extractor.py
Filename: unpacked_hancitor_2.bin


The decrypted configuration follows:

0710_pkrdv      http://strictence.com/8/forum.php|http://wimberels.ru/8/forum.php|http://cithernista.ru/8/forum.php|
```

[Figure 29]

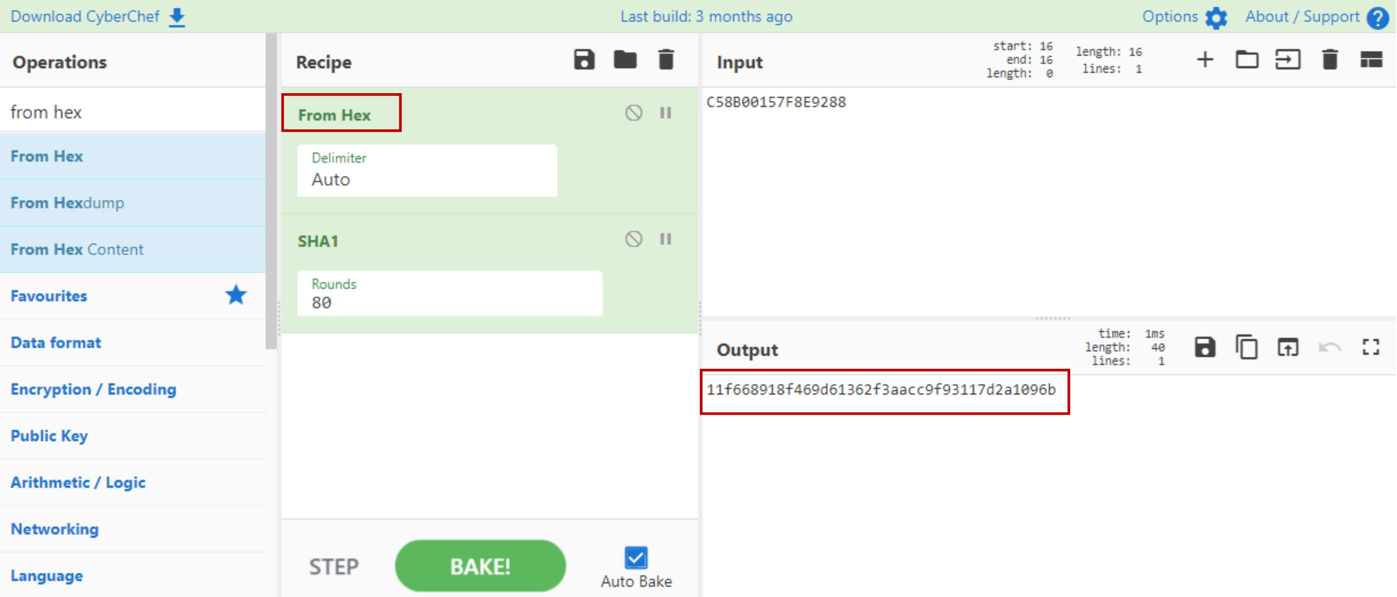
As we expected, everything has worked well again and we have a good Hancitor script to extract and decrypt its C2 configuration data.

There're two other methods that could have used **to extract the C2 data configuration from Hancitor**:

- **Cyber Chef:** https://gchq.github.io/CyberChef/
- Through a debugger (**x32dbg**)

Both approaches are great, but they are understood as "manual" methods and we would need to work on one sample a time. Using a debugger is quite simple because it's enough **to set a breakpoint on CryptDecrypt( )** and, **once hit, execute until its exit point and check (Follow in Dump) its output parameter (the 5th argument) on stack.** Certainly, the reader knows how to perform these steps.

To use CyberChef, we need to select the bytes of the key and export it using **SHIFT+E (Edit → Export Data)** and copy this data into the **Input area**. Pick up **From Hex** recipe (because exported data are in hexadecimal format) then the **SHA1 recipe** because we want a **SHA1 hash** from the **Input**, as shown in the Figure 30:



from hex

From Hex
Delimiter: Auto

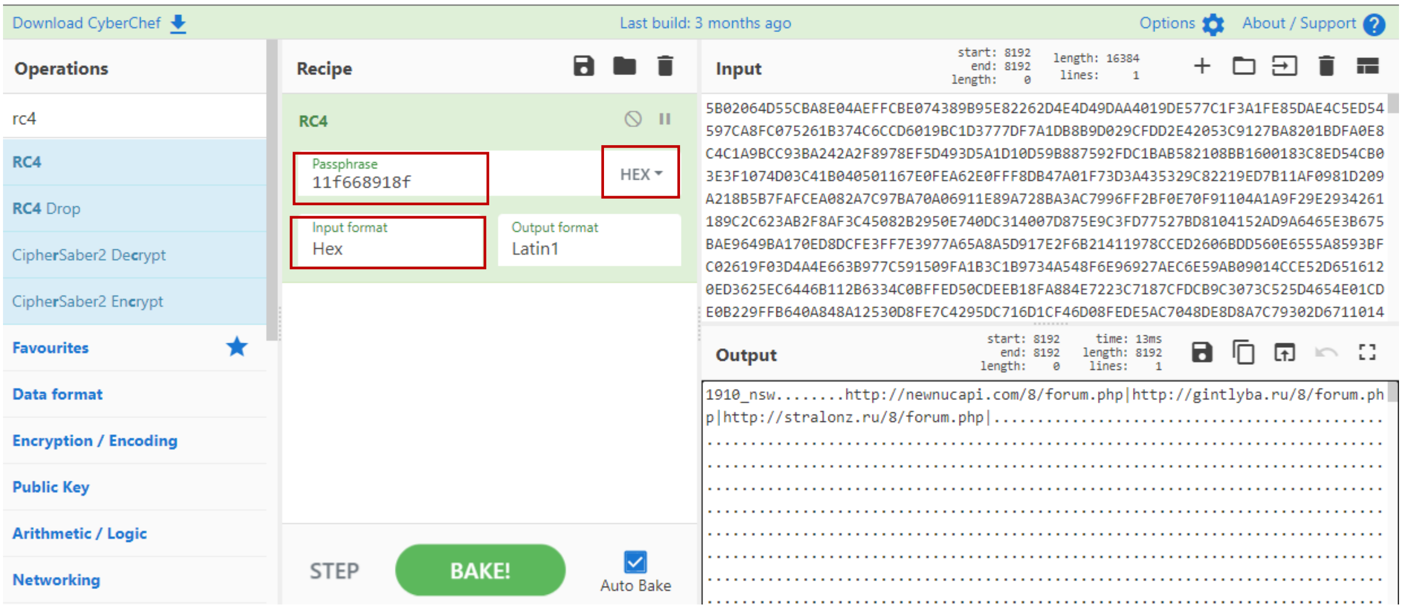
SHA1
Rounds: 80

```
start: 16   length: 16
  end: 16    lines: 1
length: 0
C58B00157F8E9288
```

```
  time: 1ms
length: 40
 lines: 1
11f668918f469d61362f3aacc9f93117d2a1096b
```

[Figure 30]

To the encrypted data (gathered from **0x10004018),** repeat the same procedure by exporting it (**SHIFT+E**) and copying it into the **Input area**. Drag **RC4 recipe** into **Recipe area** and pay attention to few points: a.) the passphrase is composed by the **first 10 hexadecimal digits** (**5 bytes**) from **SHA1 output** (as we learned from **CryptDeriveKey( )**); b.) **the Passphrase is in hex format**; c.) the **Input format** is also in hexadecimal format. Finally, we got the same result of our Python 3 script as shown in the Figure 31 below:



rc4

RC4
Passphrase: 11f668918f   HEX
Input format: Hex
Output format: Latin1

```
start: 8192   length: 16384
  end: 8192    lines: 1
length: 0
5B02064D55CBA8E04AEFFCBE074389B95E82262D4E4D49DAA4019DE577C1F3A1FE85DAE4C5ED54
597CA8FC075261B374C6CCD6019BC1D3777DF7A1DB8B9D029CFDD2E42053C9127BA8201BDFA0E8
C4C1A9BCC93BA242A2F8978EF5D493D5A1D10D59B887592FDC1BAB582108BB1600183C8ED54CB0
3E3F1074D03C41B040501167E0FEA62E0FFF8DB47A01F73D3A435329C82219ED7B11AF0981D209
A218B5B7FAFCEA082A7C97BA70A06911E89A728BA3AC7996FF2BF0E70F91104A1A9F29E2934261
189C2C623AB2F8AF3C45082B2950E740DC314007D875E9C3FD77527BD8104152AD9A6465E3B675
BAE9649BA170ED8DCFE3FF7E3977A65A8A5D917E2F6B21411978CCED2606BDD560E6555A8593BF
C02619F03D4A4E663B977C591509FA1B3C1B9734A548F6E96927AEC6E59AB09014CCE52D651612
0ED3625EC6446B112B6334C0BFFED50CDEEB18FA884E7223C7187CFDCB9C3073C525D4654E01CD
E0B229FFB640A848A12530D8FE7C4295DC716D1CF46D08FEDE5AC7048DE8D8A7C79302D6711014
```

```
start: 8192     time: 13ms
  end: 8192   length: 8192
length: 0      lines: 1
1910_nsw........http://newnucapi.com/8/forum.php|http://gintlyba.ru/8/forum.ph
p|http://stralonz.ru/8/forum.php|.............................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
```

[Figure 31]

11. Conclusion

In this first article I showed how to extract and decrypt the C2 data configuration from Hancitor malware by writing a Python 3 script. Additionally, I presented several concepts and foundations such as code injection and unpacking that will be useful in next articles of this series. Anyway, maybe it's quite relevant to highlight few points here:

a. I preferred this simple malware sample due my final purpose that's to help other professionals to take their own steps on malware analysis and, to start a series of article, certainly it was quite useful.

b. For now, I have hidden a lot reversing engineering details about this malware on propose because the initial goal was focusing only on C2 data configuration extraction and decryption in this article.

c. Once again, and unlike what many beginners in reverse engineering might think about, data extraction/decryption is not something new (not even close), and I've been doing it from many years, so it's a great topic to start with. Furthermore, getting C2 configuration is one of the main goals while analyzing a malware (few other ones are infection's vector, persistence, evasion and network communication, for example).

d. I chose Python 3 as script language because I think it's easier to understand and most of security researchers know well about it. No doubts, we could write programs in C or Golang and, eventually we can use them in next articles.

e. We will study several samples and contexts in the next articles and review topics such as **COM, unpacking, code injection, C2 emulation, .NET reversing, anti-analysis techniques, API resolving, string decryption, IDC/IDA Python, IDA AppCall** and so on, so let's take it one step at a time.

Don't forget: this is a live document and I will update it soon I find mistakes and errors.

I have been working with reverse engineering for over a decade, I'd like having started this series previously, but it wasn't possible, unfortunately. Therefore, now my plan is to write a book about malware analysis to contribute to the security community and continue this series. Let's see what will happen.

Just in case you want to keep in touch, my public contact information follows below:

- **Twitter:** @ale_sp_brazil
- **LinkedIn:** https://www.linkedin.com/in/aleborges
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

Alexandre Borges